Session Overhead Reduction in Adaptive Streaming

A Technical Paper prepared for SCTE•ISBE by

Alexander Giladi
Fellow
Comcast
1899 Wynkoop St., Denver CO
+1 (215) 581-7118
alex_giladi@comcast.com

Table of Contents

List of Figures

List of Tables

1. Introduction

In typical linear adaptive streaming deployments, much thought is given to bitrate optimization, in order to maximize a viewer's quality of experience. The non-negligible overhead of the manifest traffic is, however, typically overlooked. A manifest contains information essential for streaming a video asset, identifying the contents of the stream and the location of where constituent components, like URLs, can be found. A manifest which is frequently refreshed can significantly impact bandwidth consumption and number of requests made to Content Delivery Networks (CDNs). This is true for both Apple® HLS and MPEG DASH streaming systems. While the "manifest bloat" problem is endemic for both systems, this paper concentrates on MPEG DASH and DASH-specific tools. In the case of MPEG DASH, there are several major sources contributing to manifest growth, such as the number of included content periods and the sheer volume of DRM license information. Frequent requests further compound the burden of what is essentially "manifest bloat." Altogether the manifest overhead can easily reach 250Kbps and go past 2Mbps in some pathologic cases. These numbers are well beyond the typical low-rate video bitrate.

There are several different approaches to minimizing the manifest overhead. Some are as simple as using HTTP lossless compression algorithms, such as gzip, or the more exotic Brotli [11]. Current DASH practices, such as predictive templates, events, and the timeline extension process, provide an orthogonal approach. Operational experience with large-scale, DASH-based linear programming resulted in a set of new manifest reduction tools that were consequently added into the recent version of MPEG DASH. For example, a recently introduced MPD patching mechanism dramatically reduces the manifest overhead by only sending updates when possible. Several other additions reduce the size of the DASH manifest, by reducing the "bloat" due to license acquisition information in multi-DRM content and segment losses.

Adaptive streaming over HTTP emerged as the mainstream method of delivering video IP-based networks roughly a decade ago as a response to the challenges of network unpredictability and device heterogeneity. Adaptive streaming technology is what powers over-the-top (OTT) services such as Netflix, YouTube, Hulu, Disney+, Peacock, and HBO Max, among others. In aggregate, adaptive streaming techniques reach hundreds of millions of viewers on a dizzying variety of devices.

Adaptive streaming is by no means a new technology. C. Gonkin et. al. [1] wrote the one of the earliest papers on adaptive streaming describing the Real Networks implementation of the approach. The technology became mainstream much later, after the introduction of the first version of Apple HTTP Live Streaming [6] (HLS) in 2009. Several additional streaming systems, such as Microsoft's HTTP Smooth Streaming (HSS) and Adobe's HTTP Dynamic Streaming (HDS) emerged at around the same time and became viable alternatives. Two years later, MPEG Dynamic Adaptive Streaming over HTTP [2] (DASH) became an international standard. Today, HLS and DASH are responsible for the vast majority of adaptive streaming deployments. DASH also made its way into broadcast as a part of Advanced Television System Committee (ATSC) 3.0 and Europe's Hybrid Broadcast Broadband TV (HbbTV).

As opposed to traditional broadcast, cable, and IPTV systems, which push content to receivers, adaptive streaming is a "pull" system, where a streaming client makes an autonomous decision about what to download based on current network conditions and device capabilities. The content is encoded in multiple bitrates referred to as *representations*. Each representation is encoded as series of short *segments* (playable pieces of video, typically 2-10 seconds each). Representations which have the same media content (e.g. video at different bitrates) and have aligned segment boundaries are grouped into *adaptation sets*. The complete asset (e.g. a movie, a pre-recorded or live show) is broken into one or more independent *periods*, which cover a uniform period of time within a presentation. While video on demand (VOD) assets without advertisements consist of a single period, linear channels typically contain multiple periods. An example of such a multi-period asset can be a linear channel where the first period covers entertainment content, the next three cover three different advertisements, and the fifth covers the continuation of the entertainment content.

HLS is similar. Its manifest is a collection of media playlists listing segment URLs, and a master playlist that references the media playlists and lists their properties. The media playlists are conceptually identical to representations, while the master playlist combines the roles of both period and adaptation sets.

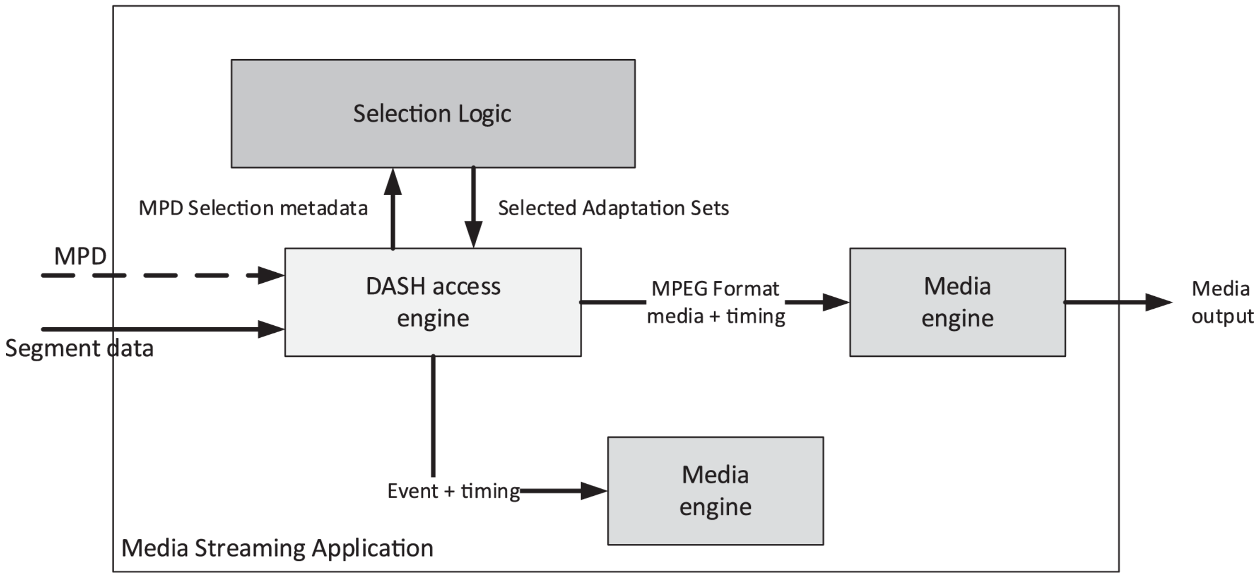

Figure 1: Anatomy of DASH-based streaming application[2]

A DASH streaming session starts with downloading a Media Presentation Description (MPD), which is an XML document containing information about media segments, their timing, and the inter-relationships between them. After parsing the MPD, the client selects the representations it sees fit, given its capabilities and network conditions. It starts downloading the media segments from the selected representations. The segments are typically kept in a player buffer, and eventually decoded and rendered. The client also continuously estimates available bandwidth and monitors buffer fullness, and given these it re-evaluates its representation selection.

A. C. Begen et al [8] provide an excellent introduction to the concepts of adaptive streaming. A much later work by A. Bentaleb et al. [9] provides an overview of modern rate adaptation techniques.

2. Session overhead and HTTP compression

Any streaming session starts with downloading the manifest. In cases of linear content, this manifest is periodically updated. HLS downloads the master playlist at the beginning of the session, and refreshes the media playlists per each segment [6]. While DASH has several tools for avoiding unnecessary MPD requests, many naïve implementations do not use them, and end up implementing an HLS-like client, where new MPD is requested prior to each segment request.

MPDs can be quite “chatty”. Table 1 below lists linear channel streaming bitrate overheads as measured with several production-grade MPDs, and quantifies the corresponding bandwidth overhead associated with 2-sec segment durations. This 2-sec duration implies an MPD request every 2 seconds (i.e. per each segment).. Two-second segment durations are both a common practice and a recommendation (see e.g. S. Lederer [10]).

Table 1: MPD overhead

MPD	MPD Size (bytes)	Bandwidth (kbps)
A	61904	247.62
B	326933	1307.73
C	425867	1703.468
D	552219	2208.88

We are seeing nearly 250Kbps of overhead with a single-period MPD A, and quickly exceed 1 Mbps as the number of periods in the MPD grows. Using the HLS bitrate ladder described in [7], a 1.3 Mbps overhead translates into the difference between 360p and 540p resolutions. This is hardly negligible. Beyond pure traffic, the size of the MPD also translates into parsing time and memory footprint, both functions of number of XML elements.

HTTP/1.1 allows compression of the body of the HTTP response. This is achieved using transfer coding and happens between the endpoints of the protocol, thus it is mostly transparent to the application. Gzip compression is mandatory in HTTP/1.1, while the newer and widely supported Brotli compression [11] provides noticeably better results. Session-related traffic overhead shrinks dramatically – up to 98% – if HTTP compression is used. – this is illustrated in Table 2 . Even with the most efficient Brotli compression, however, the bitrate overhead is still 50 Kbps for an 18-period MPD D, and nearly 250 Kbps with the ubiquitously supported gzip.

Table 2: MPD size (in bytes) with and without HTTP compression

MPD	Uncompressed	Gzip	Brotli
A	61904	7096	5598
B	326933	37322	5589
C	425867	17266	6583
D	552219	16885	10975

We can see that application of HTTP compression is essential to make MPD traffic overhead manageable and reduce the start-up time (due to a much faster MPD download).

The HTTP compression approach only addresses the size of the MPD in bytes on the wire and on the CDN. There are additional aspects which depend on the number of XML elements in the MPD: the memory footprint and the MPD parsing time. These need to be addressed using DASH-specific techniques, which are described in the next section.

3. Reducing traffic overhead

This section reviews DASH-specific approaches which can be used to reduce the MPD size, number of XML elements, and parsing time. First we discuss MPD patching, a very powerful tool introduced in the latest amendment to MPEG DASH. Patches provide an extremely significant improvement in traffic overhead. Other techniques are needed to remove unneeded XML elements from the actual XML document, and are instrumental in reducing MPD parsing time and its memory footprint.

3.1. MPD patch

In the vast majority of cases the change between the previous and the current MPD is minimal. For example, a new segment may have been added in all representations, and the oldest segment may have been removed. This means that the change affects a minimal number of XML elements. Sending only the difference across consecutive MPDs, as updates, is undoubtedly more efficient than downloading full MPDs. This technique can be far more efficient than the generic application of HTTP compression.

The MPD patching framework was first introduced in 3GP-DASH [13] as MPD Delta. The delta format was line oriented, where operands (insert, remove, replace) applied to lines of text. This approach has a major weakness – it implicitly assumes existence of an actual MPD file on the client. This is often not the case, as clients often store the MPD in an in-memory structure, which may be Document Object Model (DOM) or a custom data structure. Secondly, line-oriented syntax is ill-suited for the case where XML document can be regenerated by different entities – whitespace differences can result in patch application errors. Lastly, the MPD Delta syntax ignored version mismatches – e.g. when a modification made to an MPD which differs from the one used for the creation of the delta file, which may render the MPD constructed in the client memory invalid.

The more recent MPEG DASH patching framework takes a slightly different approach: it operates on XML elements and not lines, and addresses elements using XPath as opposed to line numbers. The syntax of the MPD patch is a very restricted subset of the IETF XML patch framework [15]. The restricted syntax creates a system where there typically is only one way of addressing each specific element, and a mandatory check for the client MPD version precedes each patch application. Amendment 1 [3] to the 4th edition of MPEG DASH allows explicit requests for MPD patches as an alternative to requesting a full MPD.

MPD patches change the way the client operates: instead of a simple repeated request to an MPD URL, the client alternates between full MPD and patch requests. When the client starts the streaming session, it downloads an MPD. This downloaded MPD provides URLs for both the next MPD and the next MPD patch. If the client decides to download the patch, the patch process will first validate the MPD version, in order to avoid a mismatch. If the download is unsuccessful, or the patch process fails, the client will request the full MPD as an update.

The results of patch application are strikingly better than those achieved by plain HTTP compression: a single patch for the MPDs listed in Table 2 is 944 bytes uncompressed, and only 349 bytes if Brotli compression is applied.

Table 3: Traffic overhead in megabytes for 1-hr session

MPD	Naïve	Naïve + Brotli	Patch	Patch + Brotli
A	106.26	9.61	1.34	0.678
B	561.22	9.59	1.59	0.678
C	731.05	11.30	1.10	0.630
D	947.95	18.84	1.81	0.683

Table 3 shows the impact of using patch updates during a 1-hr streaming session. It assumes 2-sec segments and a period being added every 5 minutes. We can see that the effect of patching is more pronounced than the one of the most efficient HTTP compression alone. For example, in case of MPD A Brotli compression resulted in 90.96% reduction in traffic, while patching alone resulted in a 98.74% reduction. Combining both tools is most efficient, as also reduces the first MPD download by the same 90.96%, which results in a shorter start-up time.

Figure 2 below shows an example MPD which adds 7 new segments. As we can see, it contains the MPD identifier and its version information (publication time) in order to ensure the validity of the patching operation. Another method of ensuring the right patch is downloaded is embedding the version on the new (post-patch) MPD in the URL for the next patch. This example is adopted from [3].

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Patch
    xmlns="urn:mpeg:dash:schema:mpd-patch:2020"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:mpeg:dash:schema:mpd-patch:2020 DASH-MPD-
PATCH.xsd"
    mpdId="42"
    originalPublishTime="2020-05-13T05:34:06+00:00"
    publishTime="2020-05-13T05:34:28.601Z">

    <replace sel="/MPD/@publishTime">2020-05-13T05:34:28.601Z</replace>

    <replace sel="/MPD/PatchLocation[0]">
        <PatchLocation ttl="60">live-stream/patch.mpd?publishTime=2020-05-
13T05%3A34%3A28.601Z</PatchLocation>
    </replace>

    <add sel="/MPD/Period[@id='1']/AdaptationSet[@id='1']
              /SegmentTemplate/SegmentTimeline">
        <S d="360360" r="6" t="5494659049"/>
    </add>

    <add sel="/MPD/Period[@id='1']/AdaptationSet[@id='2']/
                         SegmentTemplate/SegmentTimeline">
        <S d="360960" t="5494660288"/>
        <S d="359040"/>
        <S d="360960" r="1"/>
        <S d="359040"/>
        <S d="360960" r="1"/>
    </add>

</Patch>
```

Figure 2: MPD Patch example

3.2. Segment Gap Signaling

There is no such a thing as 100% reliability in a complex content origination system. Encoders may fail, packagers may fail, networks may fail. As a result, having ideal continuous sequence of segments is hard to achieve over a long enough period of time. There will always be segment gaps – short periods where a segment was not generated by the transcoder.

Many things can go wrong. For example, when the encoder output is a MPEG-2 TS over UDP and a datagram is lost en route to the packager, the segment is lost as well. If the encoder output is a multicast per representation (a very common configuration), occasional packet loss will affect a single representation at a time.

Another example is an encoder failure – if an encoder fails, for some reason, and a redundant encoder is started and starts outputting segments, there may be a gap of one or more segments, between the last segment written to origin by the primary encoder, and the first segment written by the redundant encoder. When different representations are generated by different encoders, and an encoder fails, there is a gap only for a subset of the representations, and there will be a segment in one or more representations.

One of the advantages offered by DASH periods is that they are independent, and it is possible to vary the number or character of representations across periods. The main driver behind this design was advanced advertising. For example, the entertainment content may carry audio in English, French, and Spanish languages as well as an English narration (video description) audio track. Moreover, some languages will be available as both stereo and multichannel audio. An ad inserted into this content may only have English stereo. This ad will be represented as a separate period in DASH; pre-ad and post-ad periods would also be periods on their own.

In case a segment from a representation is missing, many implementations assume that the representation no longer exists in the presentation. This triggers creation of a new period without that "lost" representation. At some point in time the packager will again start generating segments for the "lost" representations, which will, in turn, trigger creation of a new period.

In the author's experience, occasional segment gaps resulted in MPDs with 100-300 periods, which made them both exceptionally large and non-trivial for a player. An alternative to this would be using the SegmentTimeline element, which allows gaps in presentation time. With that said, the use of SegmentTimeline in the case of per-representation gaps is also inefficient, as it will need to appear in every single representation -- as opposed to the common practice of including them only at the adaptation set level. For example, in the case of an unencrypted, 12-representation HLS bitrate ladder [7], and a 2-min MPD, the MPD size increased nearly ten-fold, form 11KB to 105KB.

All of this can be avoided at a low cost if a missing segment is explicitly identified. Both DASH and HLS recently allowed such signaling. In DASH this is achieved using a FailoverContent element, which indicates time gaps for which the representation has no segments. This translates to just a few of lines, with a single gap (i.e., one or more consecutive missing segment) corresponding to a single XML element. This has very little impact on either the MPD size or the number of XML elements it contains.

3.3. Efficient multi-DRM signaling

The overhead of DRM license inlining traffic can be significant, especially in context of linear channels with multi-period, multi-DRM MPDs and multiple video and audio options. The Common Encryption standard [4] and several DRMs define a way of including license acquisition information in the MPD. This is needed in order to start the license acquisition in parallel with the download of an initialization segment, as opposed to waiting for it and parsing this information out of a box contained in it. This inevitably grows the MPD size.

For example, consider a single-period MPD with stereo and multi-channel adaptation sets for both English and Spanish, video, and trick modes has 6 cenc:pssh elements. The number is multiplied by the number of DRMs – meaning that the same single-period MPD service with 3 DRMs contains 18 cenc:pssh elements, but only 3 of them are unique.

What makes the problem acute is that these elements are fairly large. For example, the 18-period 320Kb MPD B contains 171Kb worth of license acquisition data embedded in its ContentProtection element. The vast majority of this data is redundant.

The recent amendment to the 4th edition of MPEG DASH [3] introduced a referencing mechanism for ContentProtection elements. Unique ContentProtection elements (one for each DRM) are placed once, at the MPD level, and given unique IDs. They are then referenced by ContentProtection descriptors at adaptation set level, which results in 3 unique elements in the reference ContentProtection descriptors and 18 one-line dependent ContentProtection elements. This on its own dramatically reduces the MPD size. For example, the 320Kb MPD B is reduced to 86Kb.

Table 4 illustrates the results of referencing in MPDs A, B, and D, which have 3 DRMs. We can see that referencing results in a very significant reduction in uncompressed size of an MPD. As opposed to HTTP compression, this result translates directly into reduction in memory footprint. With that said, the result of applying lossless Brotli compression to an MPD with referencing is near-identical to applying same Brotli compression to the original MPD.

Table 4: MPD size (in bytes) with referencing and HTTP compression

MPD	Periods	Original	ContentProtection Referencing		
			Uncompressed	Gzip	Brotli
A	1	61904	18170	6447	5596
B	10	326933	88010	7350	5568
D	18	552219	155650	16128	11051

Referencing can also be used to reduce the size of an MPD patch carrying a new period, as the license acquisition information would be a significant part of the patch, and it typically does not change every period.

4. Reducing the number of HTTP requests

A naïve HLS-like implementation also implies that MPD traffic is responsible for a third of CDN requests. The number of CDN cache misses is lower for the MPDs, but the number of actual requests is still quite large – a third of the requests for an asset containing video and audio. The sheer number of HTTP GET requests is taxing the edge caches.

Several tools in the MPEG DASH specification are intended to reduce the number of HTTP requests for linear content.

4.1. Asynchronous MPD updates

DASH has an inband event mechanism largely modeled after SCTE 35 cue messages in MPEG-2 TS. DASH inband events are timed "blobs" of metadata embedded in an Event Message (`emsg`) box. This box resides in the beginning of an ISO-BMFF media segment, before the Movie Fragment (`moof`) box. In order to ensure that the event is received regardless of the representation, all representations within an adaptation set carry the same inband events. A client is expected to parse the very beginning of the incoming media segment, and in case it finds an `emsg` box, it is expected to pass it to the application or process itself.

One key event type defined in the DASH specification is the MPD Validity Expiration event. It lets the client know the time at which its MPD is going to expire. If the presence MPD Validity Expiration event is signaled in the MPD, the client does not need refresh the MPD until explicitly instructed by the MPD Validity Expiration embedded in a media segment. This is a very powerful feature when coupled with the features described in the next two sections – predictive templates and timeline extension.

4.2. Predictive templates

Templates are one of the major differences between DASH and HLS. In DASH, templates are mandatory for all DASH Live profiles intended for linear channels and events. A template is a string similar to the one used in printf functions in the C programming language. The template string has several predefined variables, two of which, $Number$ and $Time$, are of particular interest.

$Number$ stands for the number of the requested segment. This number is incremented for every segment, and URLs for every segment are derived by inserting the value of $Number$ into the template string. This way, given a more or less constant segment duration, there is no need to update the MPD, because future segment URLs can be derived along with an approximation of their availability window. The DASH-IF guidelines [5] allow segment duration to vary within ±50% of the target segment duration, however they limit the accumulated drift of any number of consecutive segments to the same 50% of a segment duration. For example, if we assume 2-sec segment duration, each segment can be between 1 and 3 seconds, but the cumulative duration of 1000 segments has to be between 1999 and 2001 seconds.

The $Time$ variable leverages the precise value of the start time of the segment. It is used with the SegmentTimeline element, which contains one or more S elements. Each S element represents a sequence of one or more consecutive segments of equal duration, and does this using the principle of run-length coding. For example, a single S element describes 42 consecutive 2-sec segments as a run of 42 segments with length (duration) of 2 seconds each. The start time of the first segment of the run, the run, and the common duration of the segments are all indicated in the S element. Note that the durations are precise -- if the 43rd segment is even one frame shorter or longer than that, it will be described in a separate S element.

A special run value (i.e., the value of the S@r attribute) of "-1" means "until further notice". This way there is no need for an MPD update as long as the segments are precisely identical – start time and duration allow calculation of the value of $Time$ and hence the derivation of a segment URL for a segment which has not yet been encoded. This has the same effect as the $Number$-based approach, but is easier to use and validate, since the times are precise and not an approximation with a ±50% tolerance.

Templates using the autoincrementing $Number$ or $Time$ with run of -1 let us predict URLs of future segments. This way no MPD update is required as long as segments are being made available in time – their URL and the time at which they can be requested can be calculated way before these segments are even created. This works fine if there is no change in the MPD beyond adding and removing new segments.

The likelihood of not having any other changes in a linear manifest is nil – for example, due to advanced advertising which adds new periods and new SCTE 35 events. This commonly leads to an HLS-like implementation where MPD request is issued for every segment. However, the MPD Validity Expiration event can be used to trigger an MPD update before a material change in the MPD, such as a new upcoming period. This approach reduces the number of MPD requests to the bare minimum – potentially, once per every period as opposed to once every 2-second segment. For example, an hour long MPD with an entertainment period, followed by an ad period, followed by the next entertainment period, may require just one MPD update (prior to the start of the ad period) as opposed to 1,800 requests for the traditional per-segment polling.

4.3. Timeline extension

The SegmentTimeline predictive mode is based on an assumption of frame-identical segment durations. Given US fractional frame rates, this is not always possible, because there is no reasonably short segment duration to align a 1024-sample AAC frames, 1536-sample E-AC-3 frames, and 29.97 fps video. As a result, audio segments typically have a duration pattern where periodically adding one longer or shorter segment prevents a drift from forming. This reduces the efficiency of $Time$-based addressing by limiting run values in audio adaptation sets.

A more efficient mode, commonly referred to as “timeline extension,” is documented in DASH-IF IOP [5] as “MPD and Segment-based Live Service Offering”. Given that each S element provides a time precisely matching the time in the Track Fragment Decode Time (`tfdt`) box of the media segment, it is possible to predict the URL of the next segment, having parsed the beginning of the current segment. This approach allows a reliance on event-driven, asynchronous MPD updates and results in the same performance as in the $Number$-based case, but with much higher precision.

Table 5: Request and traffic overhead of a 1-hr session with asynchronous updates

MPD	Naïve		Asynchronous updates		
	Traffic (MB)	GET requests	Uncompressed (MB)	Brotli (MB)	GET requests
A	106.26	1800	0.71	0.064	12
B	561.22	1800	3.74	0.064	12
C	731.05	1800	4.87	0.075	12
D	947.95	1800	6.32	0.126	12

Table 5 shows the effect of using asynchronous MPD updates (without any of the traffic-reducing techniques from section 3) in a scenario we used in Table 3 above: 1-hr session with 5-min periods and 2-sec segments. We further assume that an MPD Validity Expiration event is sent before the start of the period. This scenario results in a 99.33% reduction in HTTP GET requests and shows the best result in traffic reduction. With that said, the numbers below represent a “happy path”, and every missing segment in each currently playing representation will trigger an extra MPD request. DASH allows a “missing content segment” – a segment which contains no media data and only provides segment timing information. In case of timeline extension, such a segment will prevent unneeded MPD requests.

Note that asynchronous updates can be combined with MPD patches to get to even greater efficiencies.

5. Conclusion

In this paper we quantified the impact of “manifest bloat” specific to DASH and its MPD traffic and reviewed several methods of reducing it. While application of brotli compression reduces the traffic on its own by at least 90%, we recommend a combination of the proposed measures. Note that both the timeline extension and the

Measures such as ContentProtection referencing and gap signaling reduce the memory footprint and improve parsing performance by eliminating redundant elements in the MPD.

From a pragmatic standpoint, HTTP compression is the ultimate “low hanging fruit,” in that it carries a significant traffic impact at a fairly low cost. Gap signaling and ContentProtection referencing are relatively easy to implement and operate. MPD patches are also a new feature and requires an implementation effort on both the client and the packager side, with that said its benefits are significant enough to justify those efforts.

Abbreviations

3GP	Third Generation Partnership
AAC	Advanced Audio Coding
ATSC	Advanced Television Systems Committee
CDN	Content Delivery Network
DASH	Dynamic Adaptive Streaming over HTTP
DASH-IF	DASH Industry Forum
DOM	Document Object Model
DRM	Digital Rights Management
FPS	Frames per Second
HbbTV	Hybrid Broadcast Broadband TV
HDS	HTTP Dynamic Streaming (HDS) (Adobe)
HLS	HTTP Live Streaming (Apple)
HTTP	Hypertext Transfer Protocol
ISO-BMFF	ISO Base Media File Format (a.k.a. mp4)
IETF	Internet Engineering Task Force
MPD	Media Presentation Description
MPEG	Moving Pictures Experts Group
URL	Uniform Resource Locator
VOD	Video On Demand
XML	Extensible Markup Language

Bibliography & References

[1] G. J. Conklin, G. S. Greenbaum, K. O. Lillevold, A. F. Lippman and Y. A. Reznik, "Video coding for streaming media delivery on the Internet," in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 3, pp. 269-281, March 2001, doi: 10.1109/76.911155.
[2] ISO/IEC 23009-1:2019, Information technology — Dynamic adaptive streaming over HTTP (DASH) — Part 1: Media presentation description and segment formats, 4th edition.
[3] ISO/IEC SC29 WG11, Text of ISO/IEC 23009-1 4th edition Draft Amendment 1, CMAF support, events processing model and other extensions, available online at https://wg11.sc29.org/doc_end_user/current_document.php?id=74740&id_meeting=182
[4] ISO/IEC 23001-7:2016, Information technology — MPEG systems technologies — Part 7: Common encryption in ISO base media file format files, 3rd edition
[5] DASH-IF Guidelines for Implementation: DASH-IF Interoperability Points, November 2018, available online at https://dashif.org/docs/DASH-IF-IOP-v4.3.pdf
[6] R. Pantos, HTTP Live Streaming 2nd Edition, IETF I-D, available online at https://datatracker.ietf.org/doc/html/draft-pantos-hls-rfc8216bis-07
[7] Apple Inc., "HLS Authoring Specification for Apple Devices", available online at https://developer.apple.com/documentation/http_live_streaming/hls_authoring_specification_for_apple_devices
[8] Ali C. Begen, Tankut Akgul and Mark Baugher, "Watching video over the Web, part 1: streaming protocols," IEEE Internet Comput., vol. 15/2, pp. 54-63, Mar./Apr. 2011.

[9] A. Bentaleb, B. Taani, A. C. Begen, C. Timmerer and R. Zimmermann, "A Survey on Bitrate Adaptation Schemes for Streaming Media Over HTTP," in *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 562-585, Firstquarter 2019, doi: 10.1109/COMST.2018.2862938.
[10] S. Lederer, Optimal Adaptive Streaming Formats MPEG-DASH & HLS Segment Length, November 2015, available online at https://bitmovin.com/mpeg-dash-hls-segment-length/
[11] IETF RFC 7932, Brotli Compressed Data Format, July 2016
[12] IETF RFC 7232, Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, June 2014
[13] 3GPP TS 26.247 'Progressive Download and Dynamic Adaptive Streaming over HTTP (3GP-DASH) v. 2.0.0 (Release 10), June 2011.
[14] Zachary Cava, Scaling Live OTT with DASH: Techniques and Lessons Learned, Mile-High Video 2019, Denver CO, available at http://mile-high.video/files/mhv2019/pdf/day2/2_16_Cava.pdf
[15] IETF RFC 5261, An Extensible Markup Language (XML) Patch Operations Framework Utilizing XML Path Language (XPath) Selectors, September 2008