Chunked DASH in JavaScript

ROBERT ALNESJÖ
Master of Science degree project in Computer Science and Engineering

Chunked DASH in JavaScript

Robert Alnesjö

Principal supervisor
Torbjörn Einarsson*

Supervisor
Haibo Li

Examiner
Anders Hedman

February 22, 2018

*Edgeware AB
Abstract

Chunked DASH is getting attention for reducing otherwise high delay of live segment streaming but there are a lot of unexplored problems associated with it. This master’s thesis investigates the difficulties involved with implementing a chunked DASH player in the browser with JavaScript.

A small system containing one JavaScript client and a server which simulates live streaming by repeating VOD segments is implemented. Issues related to the downloading of chunked segments are addressed and solved such that chunked segments can be streamed within expected delay, and with accurate throughput metrics.

Sammanfattning

Chunkad DASH får uppmärksamhet för sin förmåga att minska annars hög fördröjning vid segmentbaserad direktsändning, men det finns många associerade problem som inte har utforskats i någon större utsträckning. Denna masters avhandling undersöker svårigheterna med att implementera en chunkad DASH-spelare i webbläsaren med JavaScript.

Ett litet system som innehåller en JavaScript-klient och en server som simulerar direktsändning genom att upprepa VOD-segment implementeras. Frågor relaterade till nedladdning av chunkar behandlas och löses så att innehållet kan sändas inom förväntad fördröjning och med pålitliga mätvärden av genomströmmning.
Contents

1 Introduction
   1.1 Related work ................................................. 6
   1.2 Problem statement ........................................... 7

2 Background
   2.1 Dynamic adaptive streaming over HTTP ........................ 8
   2.2 HTTP chunked transfer encoding ............................ 9
   2.3 Common Media Application Format .......................... 9
   2.4 The browser as a client environment ....................... 11

3 Live delay
   3.1 Prior to actual availability ............................... 12
   3.2 Post actual availability .................................... 13

4 Live source simulation
   4.1 Chunk packaging ............................................. 15
   4.2 Manifest adjustments ....................................... 16

5 Player
   5.1 Chunk loading ............................................... 18
   5.2 Live edge calculation ...................................... 18
   5.3 Timing and synchronization ................................ 20
   5.4 Adaption ..................................................... 21
   5.5 Measuring throughput ..................................... 21

6 Content preparation ..............................................

7 Experiments
   7.1 Live delay .................................................... 24
   7.2 Bandwidth estimation ...................................... 24
## Discussion

8.1 Duration of chunks ........................................ 29
8.2 Chunked transfer encoding and chunk parsing ................. 29
8.3 Throughput estimation ........................................ 30
8.4 HTTP version 2 ........................................... 30
8.5 Maintaining low delay during playback ......................... 30
8.6 Web caches .............................................. 31
8.7 Sustainability, ethics and societal impact ...................... 31

## Conclusion

9.1 Limitations and future work ................................. 33
1 Introduction

MPEG Dynamic Adaptive Streaming over HTTP (DASH) [IS14] is one of the most widely used systems for streaming multimedia over HTTP. A DASH server is essentially just a web server hosting a manifest and media content. This is a robust mechanism which works well with caching and content delivery network (CDN) infrastructure.

Clients intent on watching the content provided by the server will read the manifest to identify the files necessary to begin playing. The server will typically host more than one representation of the same content where quality metrics such as resolution may vary.

Small media files spanning a duration of a few seconds are typical for live streaming over HTTP, they are referred to as segments. Live streaming of segments introduces a delay equal to the duration of each segment. One way to reduce the delay is to use shorter segments, but there are benefits to keeping segments at a reasonable duration such as reducing the frequency of requests per client.

1.1 Related work

Swaminathan and Wei [SW11] demonstrated a way to deliver segments partially with HTTP chunked transfer encoding by putting some effort into making sure that each chunk can be played independently of the following chunk. Their results show that the live delay can be made independent of segment duration by introducing these shorter chunks.

Bouzakaria, Concolato, and Le Feuvre [BCL14] applied chunked transfer encoding with ISO base media file format (ISOBMFF) packaging in DASH. They introduce and make use of a new parameter in the DASH manifest to signal that segments are available early, which is an essential mechanism for chunked live streaming.

Common Media Application Format (CMAF) is a modern derivate of ISOBMFF by MPEG, and the chunks described within are designed to be used for chunked transfer of segments.

When talking about a HTTP based client it is only natural to consider developing it in JavaScript to run in the browser. There are already a few JavaScript
DASH players like the one implemented by Rainer et al. [Rai+12], dash.js\(^1\) and shaka-player\(^2\) which play segmented live content with live delay within expectations, but none that support chunked segments.

Bandwidth estimation is a common metric to control the adaption, and introducing progressive loading of segments complicates the measurement of available bandwidth as the link might not be fully utilized during the downloading process.

### 1.2 Problem statement

The purpose of this thesis is to investigate the difficulties involved with implementing a chunked live DASH player in the browser with JavaScript.

It is clear that producing, transferring and playing chunks are challenges that can be dealt with. However it is unclear how well JavaScript will deal with chunked transfer encoding through the available interfaces, and if it will be possible to accurately measure link throughput as a metric to control adaption.

To answer these concerns a chunked DASH client will be implemented in JavaScript and tested against a simulated live source. Live delay will then be measured and compared to the expected outcome, that is delay slightly above one chunk duration.

\(^1\)https://github.com/Dash-Industry-Forum/dash.js
\(^2\)https://github.com/google/shaka-player
2 Background

Before diving into the details of chunked segments we should first familiarize ourselves with DASH and briefly look at why chunked segments might be a good idea. We need to understand how the content is packaged and then transferred over HTTP, as well as how the introduction of chunks will change those steps.

2.1 Dynamic adaptive streaming over HTTP

To reduce strain on the distributing server while delivering content to a large amount of clients we place caches on the edges of the network. The Internet already distributes vast amounts of content over HTTP to an comparably vast amount of users. With the capacity of available infrastructure it is only natural that we would seek to utilize it for live content streaming as well.

One significant drawback of using HTTP for streaming video and audio is that we are forced to package the content in files, which are requested one by one. If the files are short, in terms of playback duration, then we have to request them often. On the other hand if the requests are long, since we usually have to wait for the file to be completed before we can start downloading it, the delay will increase.

A benefit to using request based loading is that the client can always choose what file to request at any time. This enables a natural way for the client to choose lower or higher bit rate codings of the same content based on its own estimation of network bandwidth or playback stability, provided that the server is hosting and advertising multiple codings of the same content.

DASH is a technique that enables streaming of media content by fetching segments over HTTP. To initialize the process a client downloads a manifest that is referred to as the media presentation description (MPD), which contains instructions on how, when and from where to download the content. It generally contains entries for multiple representations, coded with different codecs and/or quality. A set of representations encoding the same content is called an adaptation set. The client may choose one representation from the adaptation set, and switch seamlessly between them based the clients own estimation of available bandwidth. Inside the MPD it will find information necessary to assemble URLs to segments and at what time they will be available.
2.2 HTTP chunked transfer encoding

It is preferable to let segments be somewhat long to decrease load on network, web server and caches. For instance in HTTP Live Streaming (HLS), Apple suggests a segment duration of 6 seconds [TN16].

To respond to a request for some resource in HTTP we normally have to specify the size of the entire response body in the response header. The size of a segment is not known until it is fully produced, and thus segments of long duration will introduce live delay by the same duration. There is a need for a way to respond to requests without knowing the size of the final response body.

Chunked transfer encoding in HTTP/1.1 [FR14] allow omitting the content length header field and instead demand that the response body is a sequence of chunks. Each chunk begins with its own length followed by optional chunk extensions on one line, and the chunk data on the next line. The end of the chunk sequence is signaled by a chunk of length zero, which may contain some trailing header fields.

Listings 2.1 and 2.2 show how a short message could be formatted using content length and chunked transfer encoding respectively.

<table>
<thead>
<tr>
<th>Listing 2.1</th>
<th>Listing 2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP/1.1 200 OK</td>
<td></td>
</tr>
<tr>
<td>Content-Length: 7</td>
<td></td>
</tr>
<tr>
<td>message</td>
<td></td>
</tr>
<tr>
<td>HTTP/1.1 200 OK</td>
<td></td>
</tr>
<tr>
<td>Transfer-Encoding: chunked</td>
<td></td>
</tr>
<tr>
<td>4 mess</td>
<td></td>
</tr>
<tr>
<td>3 age</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

That alone will not reduce latency unless those chunks can be immediately played by the decoder when retrieved sequentially. To benefit from the chunked transfer it is necessary to package segments in a way such that each chunk can be decoded independently of subsequent chunks.

2.3 Common Media Application Format

Segments in DASH are typically structured according to the ISOBMFF [IS15]. ISOBMFF allows for very liberal structuring which is good for flexibility in writing, but adds unnecessary complexity to reading and parsing. DASH narrows down this specification and the CMAF does so even further. From here on when mentioning chunks, fragments and segments, we are referring to their CMAF counterparts.
unless otherwise stated. Chunks may still refer to HTTP chunks, but the idea is that each HTTP chunk will contain a CMAF chunk, such that the two are largely interchangeable.

What DASH refers to as a representation is roughly equivalent to a CMAF track. While representations start with initialization segments, tracks start with track headers. The header starts with a file type box (‘ftyp’), immediately followed by the movie box (‘moov’), which contains information common to all ISOBMFF fragments in this track such as the common timescale.

Similarly to initialization segments, media segments start with a segment type box (‘styp’), followed by a sequence of one or more fragments. Each fragment begins with some optional boxes followed by a sequence of one or more chunks, where the first sample within the fragment is a stream access point (SAP). A SAP is a sample within the track which can act as a starting point for the decoder. In video compression not any sample can be a SAP since forward-predicted frames (P-frames) and bidirectionally predicted frames (B-frames) rely on past and even future frames as reference for rendering.

Chunks are a form of ISOBMFF fragment starting with a movie fragment box (‘moof’) containing information on how to interpret the media data. The ‘moof’ is immediately followed by the media data container box (‘mdat’) which contains the coded sequence of media samples as a contiguous block of raw data.

![Figure 2.1: Two segments containing 24 samples. The one on top is coded as one fragment which the encoder will output all at once. The bottom one is coded with smaller chunks, each independently output by the encoder at shorter intervals.](image)

Given any subset of sequential samples from that fragment we can produce a chunk. Not all samples are SAPs, especially when it comes to compressed video where only occasional frames refresh the entire screen. Even so, since we are always requesting the entire segment it is always possible to reassemble the original coding up to the last received chunk. By having chunks last a fraction of the duration, the content of a chunk can be made available much earlier than the content of the entire fragment.
2.4 The browser as a client environment

The browser is a natural environment for implementing an application reliant on HTTP such as a DASH player. The actual playback of downloaded media files is another story and was for a long time reliant on browser plug-ins such as Adobe Flash. Introduction of the HTML video element and Media Source Extensions (MSE) enabled a standardized way for browsers to natively handle streaming and basic playback of video.

In JavaScript, retrieval of resources over HTTP is commonly done by the widely supported XMLHttpRequest (XHR). The more modern but very similar Fetch [Kes17] API lacks widespread support among older browsers but features some new functionality.

DASH Industry Forum provides the JavaScript library dash.js for live and on-demand streaming of DASH which utilizes both XHR and MSE.
3 Live delay

There are multiple aspects of the total delay that we may or may not be able to influence. Let us split the contributing factors into two groups, the delay between recording until the segment is actually available, and the delay between actual availability and playback. By actual availability we mean that the segment is completely stored as a file by the encoder/packager and all of it is ready to be downloaded by a player. This means that the DASH server is solely responsible for any contribution to the delay prior to actual availability. Actual availability may differ from the availability time as defined or calculated from the manifest, but it is restricted by DASH to be within half a segment duration from the manifest availability.

3.1 Prior to actual availability

Since the samples are stored as a block of raw data, some information on where each sample begins and ends is necessary to read them. This information is part of what is stored in the ‘moof’ by specifying, where the block of data begins, and the size of each sample in the order they are stored in. Sample sizes are typically not constant, and the size of each sample is not known until it has been produced by the encoder. Until all sample sizes are known the ‘moof’ cannot be finalized. Thus the actual availability time for a segment consisting of one single non-chunked fragment can not possibly occur before the last sample of that segment is ready.

There is bound to be some delay between a real life scene, the unprocessed digital recording and the compressed and encoded stream of samples used to package segments. In video compression there are samples that reference other samples such that the samples that are referenced must come before the referring sample in the coding order. P-frames reference past samples and thus maintain the recorded order, but B-frames reference future samples as well. Whether or not this coding delay contributes to end delay depends on the packaging as well. By selecting a number of whole sample groups it is possible to eliminate parts of that delay by letting it intersect with packaging delay.
3.2 Post actual availability

Since DASH is request based there is likely going to be a delay between availability and response start times. The manifest specifies dates at which segments will be available, and clients will prepare a request to be dispatched at that date relative to its timing source. The timing source could be the system's local time, which may not be synchronized to the server. If segments are not available at the time specified by the manifest or if synchronization offset is significant, then it becomes difficult if not impossible to completely eliminate this delay.

Once a successful response has started it will take some time until the response has been fully delivered to the requesting client, we call this the download duration. If the requested segment cannot begin playing until all of it has been downloaded, then the download duration will directly contribute to additional live delay.

To account for potential instability buffering a few segments can help reduce or remove impact such instability would have on the stream quality. Any increases to the buffer duration will increase the delay by the same amount. To achieve low delay buffer duration must be minimized, but we still want to maintain sufficient quality.
4 Live source simulation

A live source is a continuous recording from some currently ongoing event. Recorded content is coded by some encoder which will produce a continuous stream of media samples. As discussed in section 3.1, the coding process will introduce some additional latency. This is especially noticeable for video coded with B-frames, since the coder must wait for forward references.

While having a real live source based on a video camera capturing some scene would have provided a more accurate depiction of reality, there are some benefits to simulating this part instead. The idea used for this simulation is to use readily available video on demand (VOD) segments and repeat them indefinitely.

First of all the manifest must locate the segments by some means, and since the aim is for the stream to be indefinitely long, the segment template element seems like a good solution.

Segment templates are used to assemble URLs to segments based on parameters like segment number and representation ID. Listing 4.1 illustrates how the segment template element can be used in a MPD. With segments of constant duration $d_s$, the segment number of the most recently available segment at time $t$ is

$$n(t) = n_0 + \left\lfloor \frac{t - AST}{d_s} \right\rfloor - 1,$$  \hspace{1cm} (4.1)

where $n_0$ is the start number attribute of the segment template element.

Listing 4.1: Two representations using segment template to locate segments.
With a manifest specifying an indefinite amount of segments, we have to expect clients to request any one of those segments. Let the VOD media segments be the first $k$ live segments, and let the remaining live segments map to one of the VOD segments

$$n_s = f(t) = n(t) \mod k$$

such that the VOD content appears to be repeating. For this to actually work, the segments will also need to be slightly modified in their timestamps.

Actual availability was introduced in chapter 3 as the time that a segment is actually available, as opposed to manifest availability, which is the time that the segment should be available. If the indefinitely many segments for the stream is actually available as soon as the stream is, then the offset between actual availability and manifest availability will grow linearly with time. It was a requirement that this offset is not too large, and thus such a live source does neither represent correct nor desired behavior of a DASH server.

An implementation capable of a simulation like this is already provided by Torbjörn Einarsson. The rest of this chapter will be focused on introducing the changes necessary for chunked playback, and finally briefly describe how they were implemented.

### 4.1 Chunk packaging

To make it simple to experiment with chunks of various durations the chunks are packaged on demand, such that the duration of chunks can be conveniently changed by parameters to the simulator. This mechanism is based on segments prepared beforehand and stored on local disk. Provided that each such segment consists of precisely one fragment, there is at least a couple of properties that will need to be adjusted within that fragment header.

Decode time is the time expressed in track timescale at which the media fragment should begin playing. In a sense, each individual sample has its own decode time, however it is only explicitly stated once in each fragment header, thus referring to the first presented sample.

Sample entries contain meta data surrounding the underlying access unit (AU) such as decode duration and offset, size and flags. If some or all of the sample attributes are missing, then the default values must have been set and they are used instead. For instance, it is typical for all samples to have the same duration. If so then the duration can be set in either each sample entry, the fragment header for all samples within the fragment, or the track header for all fragments belonging to that track.

---

By modifying those properties it is possible to repackage a segment into fragments or chunks. Repackaging a fragment into multiple chunks demands a little bit of thought. In general we can split the fragment into any number of chunks of various duration, but to minimize delay we want to keep the duration as constant as possible.

It is generally possible to find a duration such that the fragment can be split into chunks of even duration. If the chunks split the segment then the chunk duration divides the fragment duration, and the fragment samples can be divided across the chunks. For instance the samples can be evenly split across the chunks if samples have constant duration and the sample duration divides the chunk durations.

It can be difficult to find a desired duration that divides the segment in such a way, and it might not divide both audio and video segments. Provided that a video segment has been chunked and we are in the process of chunking the corresponding audio segment, then it is not unlikely for the audio segment to not be evenly divisible by the selected chunk duration. On such an occasion we need a strategy for packaging the audio fragments in chunks with decode time as close as possible to their video counterparts. One such strategy is to package the shortest sequence of samples of combined duration greater than or equal to the desired chunk duration. Of course, the same strategy can be used for both video and audio simultaneously if desired.

Given a VOD source it is possible to generate segments for any time. Each of those segments can be split into a sequence of approximately even paired chunks. If we are able to implement such a simulator then it would be close to what could be considered an ideal chunked streaming source.

### 4.2 Manifest adjustments

Preparing the segment for chunked transfer does not mean that the client will be requesting the segment earlier. Modifying the availability start time (AST) directly would break players that are not prepared to retrieve and play segments progressively. Bouzakaria, Concolato, and Le Feuvre [BCL14] proposed the introduction of availability time offset (ATO) to notify players of early availability.
Adjusted availability start time (AASST) is specified as

\[ AASST = AST - ATO, \]  

(4.3)

where as suggested in figure 4.1, ATO is appropriately set to

\[ ATO = d_s - d_c. \]  

(4.4)

However, to separate the presence of ATO from potentially incomplete segments at AASST, an additional indicator was necessary. That indicator is a boolean property in the MPD called availability time complete (ATC) which if absent defaults to the previously normal behavior, that is true.

Furthermore a client may recognize the periodic chunk deliveries as a sign of very limited bandwidth. Some indicator is necessary within the MPD to inform clients that the segments are available earlier than normal.

Segments are not complete if the request arrives at the server between the adjusted and ordinary availability times, therefore ATC should be set to false. To minimize delay, the first playable chunk should be available for downloading as soon as it is produced.
5 Player

Since the ATO was adjusted the player has the option to request segments before they are fully available. Playback of chunked segments already work fine in dash.js, but the segment is not played until it has been fully downloaded. The player must be adjusted to push chunks directly into the decoder as they are delivered rather than when the segment downloading has finished.

5.1 Chunk loading

To do that we need to identify at what point a chunk has been fully loaded. The CMAF chunks may be transmitted as separate HTTP chunks, but that is hidden from the JavaScript APIs used for downloading. Note that whenever a whole ‘mdat’ box has been loaded, the accompanying ‘moof’ has also been loaded as long as the segment conforms to CMAF restrictions. By top level parsing of response body progress the chunks can be identified by this pattern.

In dash.js, XHR was previously used to retrieve initialization and media segments. Replacing the XHR component used for loading files with Fetch is not overly complicated, as Fetch has largely the same functionality. The streaming response body of Fetch will allow reading of new data without buffering the entire payload in memory, although it is currently lacking support in most browsers. Listing 5.1 shows an example of how Fetch can be used to progressively load chunked CMAF segments provided a top-level box parser.

5.2 Live edge calculation

Even if loaded media data that is pushed into the source buffer can be decoded and rendered immediately, a starting point must first be decided. The MPD will contain enough information for the client to decide which segment will be available at what time. Provided that the client is synchronized it has roughly two options to begin with. Either download the most recently available segment, which may be up to a segment duration old, or wait for the next to become available before downloading it.

By downloading the older segment we need to seek to the live point as to not introduce additional unnecessary delay (up to one segment duration), and then
Listing 5.1: Fetch can be used to process chunked segments progressively.

```javascript
/* Concatenate typed arrays in input order. */
function concat(...arrays) {
    /* ... */
}

/* Partition download progress into initiated chunks. */
function parse(progress) {
    /* ... */
}

/* Do something with completed chunks. */
function process(chunk) {
    /* ... */
}

fetch('http://example.com/V300/0.m4s', function ({body}) {
    return {
        reader: body.getReader()
    }
}).then(function ({reader}) {
    let remaining = new Uint8Array();
    const eat = function ({value, done}) {
        if (done) {
            return;
        }
        const progress = concat(remaining, value);
        let completed;
        [remaining, ...completed] = parse(progress).reverse();
        completed.reverse().forEach(process);
        return reader.read().then(eat);
    }
    return reader.read().then(eat);
});
```
download the next segment in the same way we would have if we were waiting for
the next segment. This approach is a bit more complicated in the sense that we
need precise seeking within the segment to a time as close to the expected live
device as possible. But if we seek too far ahead, the player will run out of buffered
media and freeze until the next chunk arrives.

5.3 Timing and synchronization

The mechanism for downloading segments relies on rather precise timing. Precise
timing of requests demands a certain degree of time synchronization towards the
segmenter. A high offset between player and segmenter may result in either

- a late request, resulting in increased delay by as much as the offset. In a
  worst case scenario, total delay is longer than the servers time shift buffer,
  such that the server will refuse the request. Or,

- an early request, offset does not need to be high at all to risk requesting
  early. The only solution here is to retry at a later time, but the client knows
  neither how early it is, nor if it is actually too late.

Provided any upper limit on synchronization and a time shift buffer of sufficient
length, the requests that are refused because of old age can be eliminated. Thus a
retry strategy can be adapted based on assumption on the upper offset limit, such
as retrying in intervals until a certain amount of time has passed.

The retry strategy helps solve the issue of early requests, but what about the
late ones? Since a request may be late by some time, delay may be increased by
just as much. We want to find out what sort of time offset to expect from an
arbitrary personal computer running the player in some browser, and if possible
minimize it. Provided that the server is accurately synchronized, then the clients
offset to a synchronized time source will be approximately the same as its offset
to server.

An initial idea is to assume that hosts almost certainly are configured with
time services that are adequately synchronized through NTP. When dealing with
browser hosts we can probably safely assume that a significant number of them
are Windows hosts. The native Windows time synchronization service w32tm\(^1\) was
made with Kerberos [KN93] five minute synchronization recommendation in mind
[KB17], which is way too imprecise for live request timing.

Unfortunately it is apparent that we can not rely on local system time to be suf-
ficiently synchronized and thus an alternative mechanism is necessary. An amend-
ment [AMD15] to the second edition of DASH introduces manifest entries for UTC

timing sources to serve this purpose. The timing source entries contain information of where and how to obtain a reference wall-clock, for instance by specifying a HTTP server and reading the date header but also various ways to interact with NTP servers.

5.4 Adaption

The essence of adaptive bit rate streaming is for the client to make a smart decision on which media representation to download and display. One way to make that decision is to estimate throughput that will be used to download the next segment. A reasonable way to approach this problem is to measure throughput of previously loaded segments and predict what the throughput will be for the next segment based on some heuristic.

If an increase or decrease in the throughput has been detected it could be time to switch to a different bit rate. Switching bit rate means taking the decision to request the next segment from another representation. It is possible to switch the currently playing representation at any time as long as the corresponding chunk is available. However that would mean requesting two representations of the same segment, thus effectively wasting bandwidth. Being a little wasteful with bandwidth may be fine when switching to higher bit rates, but definitely not when switching downwards. Therefore if the throughput can be accurately measured for each chunk we might want to switch until the next segment.

5.5 Measuring throughput

The most straightforward way to measure the throughput during a past request, which we know the size of, is to keep track of how long the download process was. By recording the request dispatch and response completion times and measuring the difference we are able to calculate the effective throughput, however that only works because the requested resource was completely available at request dispatch time. The same approach will not work for chunked payloads that are progressively output out by the server with potential idling periods between each chunk.

When fetching incomplete chunked segments, we start downloading the segment before it has been fully produced. The amount of data transferred is not substantially different, yet the total segment download time may increase tremendously if the link is not fully utilized during the full downloading period. Therefore we need to adapt some other method of throughput measurement.

From the manifest we know at what time each segment is normally available as a whole segment. Additionally the ATO specifies an adjusted availability time for
all those segments where some media data may be available, but not necessarily
the entire segment if ATC is set to false.

If some data is fully downloaded in a burst before the next chunk availability
then there will be some time during which the maximum transfer capacity is not
utilized. By some means we need to start identifying when this download idling
occurs and avoid counting that time towards the active download duration. Parsing
the payload is one way to achieve that.

A successful segment request will have received a response header and possibly
one or more chunks of the segment. If the segment consists of more chunks than
what was downloaded initially, then they will be pushed by the server when they
are ready. We make an assumption in the context of media chunks, that each
chunk will be transmitted at full speed from the server and that download idling
can only occur between chunks. Thus once a chunk has been fully downloaded we
have to expect an idling period. As soon as some progress can be observed on the
next chunk we know that the idling is over. This process can then be repeated
until the end of the payload to measure the download duration of each chunk.

As previously mentioned, loading of resources in JavaScript is done by high
level interfaces, occasionally delivering bytes to the application. This behavior
highlights a significant drawback of the method based on processing every chunk,
which is that if the entire chunk is delivered in one instance, then there is no
time difference to estimate throughput. The same can be said for when chunks
are delivered in a small number of instances. To provide sound metrics to the
estimation,

- the bytes of the first loaded instance have to be ignored, as there is really no
telling when the server actually started sending them.

- Following bytes are received at a later time and thus throughput can be
calculated,

- however the last instance may not have utilized the full link throughput
during that interval and can thus not be trusted.

Ideally throughput is estimated based on measurements on loading instances ex-
cluding the first and last, thus demanding at least three loading instances on every
chunk to provide a steady flow of throughput metrics.
6 Content preparation

A test sequence of one hour is transcoded into different constant bitrates, all using H.264 and Advanced Audio Coding (AAC) coding. For video frame rate was 30 fps, group of pictures (GOP) duration 1 s, and bitrates 300 kbps, 600 kbps, 1.2 Mbps and 2.4 Mbps. After that, GPAC MP4Box\(^1\) was used to repackage the already MP4 output into 8 s segments.

Listing 6.1

```
$ ffmpeg -i input.mp4 -c:v libx264 -r 30 -g 30 -x264-params "nal-hrd=cbr" -b:v 300k -minrate 300k -maxrate 300k -bufsize 600k output.mp4
$ MP4Box -dash 8000 -rap -url-template -segment-name '' output.mp4
```

When MP4Box is unable to fill a segment with exactly the specified duration it will round the duration down on all segments over the whole sequence. This will usually result in an undesired extra segment at the end of the sequence. To counteract that a basic repackaging tool based on the same logic used for repackaging of chunks in section 4.1 was used.

\(^{1}\)https://gpac.wp.imt.fr/mp4box/
7 Experiments

By measuring the delay of chunked playback against whole segments and short segments, we should be able to verify that the implementation behaves as expected. The simulator allows control of exactly when content should be made available to download, and by using video synchronized with local time it is possible to measure delay all the way to rendering.

Rather than specifying a time source in the manifest we make sure that both host systems are synchronized to universal time by NTP. Most of the time measurements we will be doing across both systems are in the order of seconds. Having the time offset down to a few milliseconds should make it largely negligible in comparison to other factors.

7.1 Live delay

The startup time can be measured immediately as the stream starts, while live delay is measured at the end of the session. If the expected live edge was optimistic then the player will run out of buffer before the next chunk is loaded, thus increasing delay by the time it had to wait. To measure live delay and startup time we start the stream and let it play for 10 seconds, this process is then repeated 50 times for each setup.

As can be seen in figures 7.1 and 7.2, live delay is close to one chunk duration while startup time is seemingly constant.

7.2 Bandwidth estimation

We had two approaches on throughput measurement for chunked payloads. To verify that they both work we compare both approaches on 1 s chunks. By simulating a rate limit on the source’s network interface as seen in listing 7.1, we can observe how the different setups perform in comparison to changes in the configuration.

Each run consists of the player running for one minute before manually changing the Traffic Control (tc) configuration until three minutes has passed. Figure 7.3 show the outcome of such a test performed once for each player setup, with bandwidth starting at 2 Mbps and progressing to 8 Mbps in three steps, while figure 7.4 show the same test with bandwidth decrementing.
Figure 7.1: Live delay.

Listing 7.1: Traffic control configuration for 4 Mbps rate limit.

```plaintext
# tc qdisc add dev ens3 root handle 1:0 hifsc default 1
# tc class add dev ens3 parent 1:0 classid 1:1 hifsc sc rate 4mbit ul rate 4mbit
```
Figure 7.2: Startup time.
Figure 7.3: Measured throughput while increasing simulated bandwidth.
Figure 7.4: Measured throughput while decreasing simulated bandwidth.
8 Discussion

From figure 7.1 we can see that median live delay is about $d_c + 1.3$, where the constant chunk duration $d_c$ is the theoretical minimum delay. Remaining delay is some combination of time misalignment, latency, downloading time and buffering. Latency and downloading time are minimized in the local setup, which leaves mostly misalignment and buffering.

8.1 Duration of chunks

The ultimate goal of reducing live delay during streaming is to approach zero delay. Segment duration was identified as a large contributor to the total end to end delay and chunks are meant to relieve that. To further reduce delay we can keep reducing the duration of chunks to some minimum depending on the coding.

With one second chunks a rather significant part of the delay was independent of the chunk duration. Unless that part of the delay can be reduced further, any further decreases to chunk duration will have limited effect on the total delay.

8.2 Chunked transfer encoding and chunk parsing

One of the initial concerns with JavaScript was if it could handle progressive fetching of chunked segments. Chunks could have been identified in the transfer encoding had it not been abstracted away in the Fetch API, but instead the payload was parsed based on the assumption that chunked segments are structured in compliance with CMAF. Since chunked transfer encoding was removed in HTTP/2, parsing of the response body is probably necessary anyway.

Each chunk had to be decodeable independently of future chunks and CMAF chunks was one way to accomplish that. Ultimately the parsing adds a bit of complexity to the DASH client, while limiting the server to segment structures supported by the clients chunk parser.
8.3 Throughput estimation

Both methods for measuring segment throughput performed roughly equivalent, which is the desired outcome. The content used for testing was coded with constant bit rate. If variable bit rate was used instead the results can be expected to differ, but it is not obvious which approach is more accurate as both have downsides.

The results were promising when running in a controlled environment, loading chunks in multiple instances to provide a sound basis for estimation. When experimenting with streaming over more unpredictable Internet connections, the estimation was starved for throughput metrics due to a low amount of loading instances per chunk.

The throughput measurement results are good on their own but there is more to adaption than estimating bandwidth. We ideally want chunks and short segments to be comparable in all aspects. However, since chunked segments are requested less frequently it follows that there are fewer opportunities to switch. Thus, chunked segments will be less responsive when it comes bit rate switching than short segments.

8.4 HTTP version 2

The option of chunked transfer encoding was removed in HTTP/2 [BPT15] which raises the question about the significance of this study. Chunked transfer is enabling the HTTP server to respond to a request with an initially incomplete payload, by progressively transmitting it as a sequence of arbitrarily sized chunks. Fortunately HTTP/2 response payloads are sent as a sequence of one or more data frames by default, thus providing the same functionality sought after in chunked transfer encoding.

8.5 Maintaining low delay during playback

Even if we manage to choose a good live edge with some margin for uncertainties in the communication link, if we ever run into an irregular spike, it will freeze the stream until the next chunk arrives and then continue playback even further away from the live point. MSE players allows for modifying playback rate so an interesting solution to this would be to speed up playback rate when beginning to lag behind the initial target delay, much like the fast forwarding mechanism used in [SW11].

A mechanism to help maintain a low delay on connections with occasional interference or instability is important for live streaming where delay is of concern.
One of the challenges in this would involve deciding on when to increase and possibly decrease playback speed. Increased playback rate will invalidate the reported content bit rates, but it should be fairly simple to adjust adoption based on the current rate.

### 8.6 Web caches

One of the benefits with streaming over HTTP is the ability to utilize existing infrastructure such as web-caches. Let us say there is two players maintaining live delay shorter than the ATO, and there is one web-cache between those players and the chunk packager. The first player requests the most recent segment, and the first chunk is stored in the cache as it passes through.

When the second player requests the same segment, it has either been partially stored in the cache or not stored at all. If the cache has stored the partial segment then it can respond with the partial data and then remember to send any further progress to the second player as well. Otherwise the request will reach the packager, and the cache never served any purpose.

It is possible that the cache would want to buffer a certain amount of data before distributing it to pending requests. Since an incomplete chunk is useless on its own, buffering is fine as long as chunks are not buffered past their completion time. The cache would not want to parse the payload to keep track of what a playable chunk is. For that reason it would be important that each CMAF chunk is transferred as a HTTP chunk.

Unless the cache responds in the aforementioned way, during low delay streaming over HTTP with chunked transfer, the benefits of that cache are lost. It is definitively possible to adapt the caches to behave as desired, but it is possible that such behavior is not desired in all caches. Chunked transfer of segments demand that both client and server are adapted to send and receive chunks, but any middleware such as caches could be rendered useless during live distribution.

### 8.7 Sustainability, ethics and societal impact

Due to how much space video tends to use, a very large part (measured in bytes) of today's Internet traffic consists of video. End users who are watching live HTTP streams in the browser would benefit from the lower delay of chunked DASH, however due to the added complexity to chunks the caching mechanisms may not be as effective. Increasing usage of infrastructure to stream video over the Internet could be considered unsustainable and reducing delay of live streaming does nothing but provide a better user experience, possibly attracting even more
users. Not contributing to sustainable development could certainly be considered unethical, but this work provides value in contributing to a better user experience.
9 Conclusion

We wanted to investigate the difficulties of implementing a chunked DASH player in JavaScript. For that sake an implementation was made by modifying the open source player library dash.js. We managed to significantly bring down delay for chunked segments and hit a point where other factors had larger effect on the delay. Furthermore a couple of approaches to solving throughput measurement of chunked requests was tested, and their output verified to be sound.

9.1 Limitations and future work

Live streaming over HTTP is typically done in larger scale than between one server and client, thus a very obvious limitation of the experimental setup is the small scale of it. We briefly described how a cache could be added to the chain as long as it does not buffer more than a chunk at a time.
Bibliography


