Fast DASH Bootstrap

Nassima Bouzakaria, Cyril Concolato, Jean Le Feuvre
Institut Mines-Télécom; Telecom ParisTech; CNRS LTCI
46, rue Barrault, 75013 Paris, France
{nassima.bouzakaria, cyril.concolato, jean.lefeuvre}@telecom-paristech.fr

***Abstract*—Adaptive HTTP Streaming has become a widespread technology for delivering video to the end users. However, this technology usually suffers from various latencies, requiring an initial delay before playback. This is problematic for low delay or interactive applications, such as gaming or live event webcasting. A long initial waiting time is perceived by most users as bad quality, or even non-working content. In this paper, we review the causes of initial delays in MPEG-DASH and common strategies used to reduce this latency. We propose a new method based on HTTP/1.1 and compatible with existing infrastructures for the initial setup of an MPEG-DASH session. We compare our proposal to several existing approaches based on HTTP/1.x and on the HTTP/2 server push technique. Our experimental results show that 2 RTTs can be avoided and that the total downloaded size is reduced by 25% when using HTTP/1.1 and by 20% in HTTP/2 with similar latency.**

I. INTRODUCTION

According to recent studies [1], the traffic on broadband and mobile Internet is dominated by video data. This video data is delivered over the Internet, or Over-The-Top (OTT), using the HTTP protocol, especially using HTTP adaptive streaming technology. Such technology enables reusing the Internet infrastructure (caches, proxies) and offers the ability for the client to select content with appropriate characteristics (codec, language …) and to dynamically adapt the content quality depending on the available bandwidth. The standard called Dynamic Adaptive Streaming over HTTP (DASH) [2] developed by MPEG and 3GPP is becoming the most popular technology for the delivery of audiovisual content over the internet and is the scope of this paper.

The quality of an adaptive video streaming session is influenced by many factors [3], such as the bitrate of the media content or the number and duration of rebuffering steps. The start-up delay is also perceived by users as an important factor. Some studies [4] indicate that if this delay exceeds 2 seconds, the number of people that abandon viewing dramatically increases. It is therefore important to reduce it.

The start-up delay in DASH can be divided into: the bootstrap delay, which is the time needed to download the Media Presentation Description (MPD) file and so-called initialization segments (IS), required for decoder initialization; the buffering delay required for storing the first media segments; and the initial decoding time of those segments. In this paper, we focus on the bootstrap delay.

In DASH, it is the client's responsibility to make its choices on which initialization and media segments to download. It starts by issuing an HTTP request to retrieve the MPD, informing about the different media streams (audio, video, subtitle), their formats and qualities. Once the MPD is received, the client parses it, selects a set of adaptation sets compatible with its capabilities in terms of codecs, content media types, languages. Within each adaptation set, it chooses a representation that best satisfies its needs regarding bitrate, resolution, and frame rate. It then issues additional HTTP requests to fetch the IS of each selected representation for the initial playback. This double download of MPD and IS increases the bootstrap delay. Apart from the start-up delay, DASH streaming is also impacted by additional delays due to the segmentation or buffering [5], but they are out-of-scope of this paper. This paper proposes and evaluates several methods to reduce the bootstrap delay of DASH. All methods are based on the idea that the bootstrap phase should not require multiple round-trips between the client and the server. The methods exploit the counter-intuitive fact that in some situations downloading the MPD and all IS in one download can be achieved faster than downloading the MPD and then the only needed IS. Finally, the proposed methods have been designed to have no negative impact on the existing caching and delivery infrastructure.

This paper is organized as follows. Section 2 presents related works. Section 3 reviews typical DASH client strategies for starting a streaming session. Section 4 presents our proposal to reduce the bootstrap delay. Section 5 presents the test-bed of our experiments. Section 6 details the experimentations and obtained results. Section 7 concludes this paper and proposes future work.

II. RELATED WORKS

In this section, we briefly review related works on latency reduction for web and video content.

There have been several proposals to reduce the start-up delay for short web transfers, such as those required for the delivery of DASH MPD and IS. These proposals work at the TCP transport level. One proposal [6] involves increasing the initial congestion window size (init_cwnd) to 10 TCP segments (about 15 KB) in order to minimize the web latency caused by the slow start phase of a TCP connection. The authors have shown that 90% of HTTP web responses of top sites and Google applications fit in these segments and that using the proposed init_cwnd size reduces the latency by approximately 10% with the largest benefits being demonstrated in high RTT and bandwidth delay product networks. In this paper, we will also use this init_cwnd of 10 TCP segments.

Radhakrishnan et al. [7] have identified the TCP three-way handshake as an important component of web latency imposed on new TCP connections. They proposed a new mechanism called TCP Fast Open (TFO) that enables safe data transfer during TCP's initial handshake. This means that data will be transferred within a TCP SYN and SYN ACK packets.

Through traffic analysis and network emulation, the authors have shown that TFO can improve HTTP request latency by 10% and the whole page load time from 4% to 40 %. TFO eliminates one full RTT of latency but is still limited in terms of maximum data size to be transferred during the handshake, and in which HTTP request types can be sent. This proposal requires modifications in the servers.

At the application level, Swaminathan et al. [8] proposed a low latency live video streaming approach using the HTTP/2 server push feature. The paper defined three push strategies (No-push, All-push, K-push) that determine which resources to push and when to push them. The server push was implemented in a DASH streaming session. Based on experimental results, the authors have shown that the server push approach enables low latency live streaming by simply reducing the segment duration without causing an explosion in the number of HTTP requests. However, the proposed approach only allows reducing latency during the live streaming session, and the delay required for the session start-up is not addressed. Additionally, it requires HTTP/2 which is not yet deployed in all servers nor clients.

Chérif et al. [9] proposed a DASH Fast Start system which aims at reducing the start-up delay in a DASH streaming session using the SPDY server push feature. The server pushes a set of IS that the client can accept or reject. The results show that the start-up delay was minimized up to 50%. However, the authors compared the performance of the proposed solution using SPDY to HTTP/1.1 only in the case of parallel TCP connections. In this paper, we will use a similar approach but with standard HTTP/2 and we will compare it to different TCP approaches.

The aim of our paper is to improve the start-up delay, specifically the bootstrap delay in DASH live streaming, at the application level based on HTTP/1.1, by reducing the number of round trips between the client and the server before starting the streaming session, and without modifying the existing web infrastructures or the TCP stack.

III. DASH CLIENT BOOTSTRAP STRATEGIES

For the start-up of a streaming session, during the bootstrap phase, the DASH client is required to download MPD and IS files. Note that an IS is necessary only when media segments are based on ISO/IEC 14496-12 (ISOBMFF), that we consider in this paper, as it is used by most existing DASH deployments. For that, the client can use several strategies depending on the version of the HTTP protocol and on the number of TCP connections. In this section, we survey some strategies highlighting their advantages and drawbacks. For each strategy, we indicate the number of TCP connections, the number of HTTP requests/responses and we derive the associated bootstrap delay, which are summarized in Table II.

TABLE I SYSTEM PARAMETERS

Notation	Definition
N	Number of adaptation set in an MPD
M_j	Number of representations within an adaptation set j
M	Number of representations in an MPD
X	Number of IS in an MPD
X_c	Number of IS chosen by the client
D_{ss} (A)	Download time of A in the slow start phase of a TCP connection using HTTP/1.x
D'_{ss} (A)	Download time of A in the slow start phase of a TCP connection using HTTP/2

Because MPD and IS are small (as shown in section VI), we consider that the server can send MPD and IS in the initial slow start phase of a TCP connection. For that, we assume that there is no packet loss, no delayed acknowledgement, and no congestion. Table I reports the employed parameters throughout the study.

A. Non-Persistent TCP Connection

Using HTTP/1.0, connections are non-persistent. This means that a client has to open a new TCP connection to send each HTTP request and receive the MPD file and the number (X_C) of chosen IS. Following the TCP standard, each connection begins with a three-way handshake which takes a full RTT of latency between the client and the server. Moreover, each resource suffers from a slow start phase. The bootstrap delay required to fetch all resources is T_1 as indicated in Table II.

B. Persistent TCP connection without pipelining

TCP connections can be maintained to send and receive multiple requests/responses, using the HTTP/1.1 persistent connection mechanism. The server can deliver the associated resource (MPD and X_C IS) over a single TCP connection. This feature allows avoiding connection setup for each IS and eliminates the TCP three-way handshake. The client incurs only one handshake, plus one slow start phase in the beginning. Using this approach, we can see however that the server is idle most of the time which can trigger a Slow Start Restart TCP behaviour. The bootstrap delay is T_2 as indicated in Table II.

C. Persistent TCP connection with pipelining

Pipelining is a little improvement of the persistent technique where HTTP requests and responses can be pipelined on a connection, so that the server idle time is reduced. Using this technique, the DASH client is able to make multiple requests for the MPD file and the X_C IS early without waiting for each individual response. The server still processes the HTTP requests in sequence, but can respond to a request as soon as the previous one is done. The server sends the responses in the same order as the requests were received which implies a head-of-line blocking problem [10]. In particular, this means that a DASH client that chooses to request multiple IS for a given adaptation set has to request those extra IS last, in particular if media segments are also downloaded, in order to avoid the head-of-line blocking. This latter problem is solved by multiplexing in HTTP/2, but this is not possible in HTTP/1.1. We note that in practice, not all web servers support pipelining. T_3 is the corresponding bootstrap delay reported in Table II.

TABLE II ANALYTICAL EVALUATION OF THE DIFFERENT DASH CLIENT BOOTSTRAP STRATEGIES

Request Strategies	**Number of TCP Connections**	**Number of HTTP Requests/Responses**	**Bootstrap Delay**
Non-persistent	$1 + X_C$	$1 + X_C$	$T_1 = 2 \times (1 + X_C) \times RTT + D_{ss}(MPD) + \sum_{k=1}^{X_C} D_{ss}(IS_k)$
Persistent	1	$1 + X_C$	$T_2 = (2 + X_C) \times RTT + D_{ss}(MPD) + \sum_{k=1}^{X_C} D_{ss}(IS_k)$
Pipelined	1	$1 + X_C$	$T_3 = 3 \times RTT + D_{ss}(MPD + \sum_{k=1}^{X_C} IS_k)$
Parallel	$1 + X_C$	$1 + X_C$	$T_4 = 2 \times RTT + D_{ss}(MPD) + \max_{k=1,..X_C}(2 \times RTT + D_{ss}(IS_k))$
HTTP/2	1	$1 \quad / \quad 1 + X_C$	$T_5 = 3 \times RTT + D'_{ss}(MPD + \sum_{k=1}^{X_C} IS_k)$

D. Parallel TCP connections

Another possible strategy for a DASH client consists in opening multiple parallel TCP connections. This is used for example by web browsers when downloading web page resources. The maximum number of parallel connections that recent browsers use is 6 [10]. The use of multiple connections eliminates the response queue on the server side compared to the single persistent pipelining connection, but is not always supported by servers. Each connection setup introduces an overhead due to a TCP three-way handshake and a TCP slow start state and needs to share its bandwidth with other connections. In absence of pipelining, the number of HTTP requests/responses is the same as the number of connections. The bootstrap delay is T_4 as indicated in Table II.

E. HTTP/ 2 connection

DASH client can overcome the limitations of the previous strategies based on HTTP/1.x by using HTTP/2. An HTTP/2 connection starts with a TCP three-way handshake which takes one full RTT as the TCP transport layer does not change. Most client implementations (Firefox, Chrome) support HTTP/2 only over an encrypted connection using Transport Layer Security protocol (TLS). Unfortunately, establishing a TLS secure channel between the client and the server requires a TLS handshake which takes two RTT or one RTT for abbreviated TLS handshake [10]. The server push and request-response multiplexing are the most promising features in HTTP/2. When server push is enabled, the web server pushes all IS files after receiving the MPD request instead of responding to one request for each IS file. This is achieved by the server sending PUSH_PROMISE frames to the client to signal its intention to push the X_C IS resources without requesting it. Once the client receives PUSH_PROMISE frames, it has the ability to accept or cancel the proposed IS files. However, the PUSH_PROMISE frames for IS resources must be sent by the server before the end of stream of the requested resource MPD. HTTP/2 uses true multiplexing that allows many streams (MPD and IS) to be interleaved together on a connection at the same time, so that the head-of-line blocking problem is eliminated. The bootstrap delay is T_5 as indicated in Table II.

F. Summary

MPD and IS transfers require a certain amount of round trips between client and server making the bootstrap delay a significant parameter in determining the start-up time of a streaming session. The bootstrap delay formulas in Table II are dominated by an RTT component, influenced by the number of TCP connections, the number of requests and by the slow start phase. Based on the analytical evaluation of the strategies presented in Table II, the minimum bootstrap delay using HTTP/1.x is obtained using a persistent TCP connection with pipelining. However it is not widely supported and still suffers from a big number of RTT mainly due to the number of HTTP requests/responses transfers. In the rest of this paper, for HTTP/1.x we will experiment only with the persistent TCP connection without pipelining strategy because it is the most used and supported strategy by web servers. Note however that the benefits of our approach would be the same compared to the pipelining approach.

IV. IMPROVED DASH BOOTSTRAP

In this section, we present our new approach to reduce the bootstrap delay, which consists in using a single HTTP request to fetch the necessary information to start the playback. The first HTTP request made by the DASH client to retrieve the MPD is not modified, but the response content sent by the origin server is, while remaining compatible with caches and proxies. The principle of creating this HTTP response is to rely on the MPD to carry the additional IS resources. This can be done in two ways which are detailed below. Note that although tested, multi-part messages were ruled out in this paper as they do not fit well with caches and browser-based DASH.

A. Base64 IS embedding

A simple option is to encode the IS using the Base64 encoding and to put it in the MPD file, in the initialization attribute, using the "data:" URI scheme [1]. When a client receives the MPD, it will need to decode the Base64 string to recover the original binary IS. The advantage of this naïve

[1] http://tools.ietf.org/rfc/rfc2397.txt

```
<MPD ...>
 <Period ...>
  <AdaptationSet ...>
   <Representation mimeType="video/mp4"
                   codecs="avc1.4d401f" ...>
    <ISOBMFFMoov>
      <ISOBMFTrack id="1"
                   stsd="AAAlHN0c2QA..."
                   editDelay="0.04s"/>
    </ISOBMFFMoov>
    <SegmentTemplate ... />
   </Representation>
   ...
  </AdaptationSet>
 </Period>
</MPD>
```

Fig. 1 Example of MPD embedding ISOBMFF information

method is its compatibility with the current DASH standard. The drawback is that a 33% overhead is involved when using Base64 encoding. It may therefore not be acceptable but is a good anchor point.

B. ISOBMFFMoov embedding

When looking more closely to the problem, it appears that most useful information present in the IS is also present in the MPD, such as width, height, codec profiles, sample rate, media timescale, etc. Therefore, our second option consists in adding to the MPD the missing parts required to reconstruct the IS at the client side from that MPD. For that, we analysed the MPD file and the IS of different content and we identified four potential missing pieces of information:

i. In ISOBMFF, decoder configuration is stored in the Sample Description box (stsd). For some video packaging types (identified by "avc3" and "hev1"), the configuration box is mostly empty in IS file, can be reconstructed from the MPD information, and can therefore be omitted. For other packaging types such as audio, subtitle or some video (identified by "avc1", "hvc1" or others), the box does contain information required by the client and has to be embedded in the MPD.

ii. A track may have an edit list, which shall be sent to the client to ensure proper synchronization. In most simple cases however, the edit list only consists in a single time offset.

iii. Default sample properties (size, duration, description index, flags) and sample group configurations (characteristics such as random access or pre-roll) can be configured at the file level or for each segment. In some DASH profiles such as Common File Format (CFF)[2], the properties are defined only at the media segment level. We also use this approach.

iv. Finally, to handle multiplexed representations, TrackID is required.

From this analysis, we introduced a new <ISOBMFFMoov> element in the MPD, carrying for each track its ID, the base64 stsd box and the edit list, either in base64 as shown in Fig. 1 or as a media offset to the MPD timeline. Note that the proposal can be extended to handle other file or track level boxes, such as static meta boxes. Our proposed approach is slightly similar to the IIS Microsoft Smooth Streaming existing approach[3]. However, this latter does not use at all IS files for decoder initialization. Manifests carry only the decoder configuration information. This approach is not suitable for generic ISOBMFF content.

Since this approach embeds all IS in the MPD which has a 100% cache hit ratio, it has the additional benefit of avoiding cache miss on non-popular IS.

V. Test-bed

A. Experimental Setup

Fig. 2 Experimental System

Fig. 2 depicts the architecture of the experimental system. It consists of four components: a web client, a bandwidth shaper, a network emulator, and a web server, connected via Ethernet in a local area network. The network emulator component was used to add a delay to obtain an RTT value of 50 ms using the Linux Emulator Network (Netem). Based on the bandwidth shaper component, we limited the maximum outgoing bandwidth to 2 Mbps from the server to the client using Linux Traffic Control (TC) command line tool and the Hierarchical Token Bucket (HTB). The Network emulator and the bandwidth shaper were running on the server machine.

We implemented a web server that supports HTTP/1.1 and HTTP/2 on top of NodeJS. In the case of HTTP/2, we used the server push mechanism to start pushing all the IS resources as soon as it receives the client request for the MPD. If the client does not want a pushed IS files, it can reject it. We used the Chrome Canary browser and the Dash-JS[4] video player which is based on XMLHttpRequest (XHR).

Because init_cwnd is a critical parameter in determining how quickly the DASH streaming session can start, our experiments were evaluated under two different values of init_cwnd. We set at first the init_cwnd value to 3 TCP segments and then to 10 TCP segments. The init_cwnd was configured on the server side, running Ubuntu, with the default congestion control algorithm "TCP Cubic", using the "initcwnd" option of the ip route command. The Maximum Transmission Unit (MTU) allowed by Ethernet is 1500 bytes. When we exclude IP and TCP headers from the MTU, it remains a Maximum Segment Size (MSS) with 1460 bytes.

B. Dataset

We used ISOBMFF live profile DASH content from the DASHIF[5]. We have selected 33 sequences (MPD files and associated IS), for which multiple qualities, bitrates, codecs, languages are available. Common Encryption test cases were not selected due to some authoring issues in the source IS. Only the first period IS were considered, as we want to

[2] http://uvvuwiki.com/images/c/cb/CFFMediaFormat-1.1r1.pdf

[3] http://www.iis.net/downloads/microsoft/smooth-streaming

[4] http://dashif.org/reference/players/javascript/1.3.0/index.html

[5] http://dashif.org/testvectors/

evaluate bootstrap time not seek time. N equals to 2 (video and audio) in each MPD. M is 7 where M_{video} is bounded between 1 and 6 representations, in various bitrates or resolutions. Videos identified by "avc1" and "avc3" are present. In addition, M_{audio} is only 1 representation, using the "aac" codec. None of these sequences uses shared IS among representations (bitstream switching). Therefore, X varies between 2 and 7 in this content.

VI. Experiments and Results

In order to validate our approach, we have conducted two types of experiments: experiments to measure the total download size of the MPD and IS files, including the HTTP response headers for our two methods and for all strategies reported in Table II, and experiments to measure and compare the bootstrap delay between the persistent TCP connection without pipelining strategy, our proposal using HTTP/1.1 and HTTP/2, and the HTTP/2 push-based approach. All results are available publicly[6].

A. Total download size

We first measured the IS size for all representations in our test sequences, and noted that this size is within the range 600 bytes to 1KB.

Then for all strategies reported in Table II, we measured the size of the responses i.e. the MPD, the X_C files, and the headers when:

- X_C is maximal (i.e. equal to M) as implemented by GPAC player[7] .
- X_C is minimal (i.e. equal to N) as implemented by Dash-JS[4] player where the average IS size of each adaptation set is used.

Table III reports the average, min and max sizes for the 33 DASHIF sequences. We can see first that our ISOBMFFMoov-based embedding method reduces the downloaded data size by 25% compared to the approach that download the minimal amount of IS and the MPD separately (as implemented in Dash-JS). As we can also see, the strategy used by GPAC, which downloads all IS to prepare for future switches, always leads to more bytes downloaded than Dash-JS. Interestingly also, we can see that downloading all IS in one single HTTP response using Base64 encoding may lead to a smaller amount of data being downloaded. This is due to the size of the HTTP response headers. We measured that those headers are around 257 bytes per response, mainly due to long strings used for cache information (Etag, modified dates). Note that those measures already exclude the large (~230bytes) Cross-OriginResourceSharing headers (CORS).

TABLE III. TOTAL DOWNLOAD SIZE (MPD, IS, HEADERS) FOR EACH STRATEGY (BYTES)

	X_C=M (GPAC)	**X_C=N** (Dash-JS)	**MPD** Base64 IS	**MPD** ISOBMFFMoov
Average	7524	5221	7627	4075
Min	3972	3641	3829	2451
Max	10364	8168	11211	6844

[6] http://download.tsi.telecom-paristech.fr/gpac/dash-bootstrap
[7] http://gpac.io

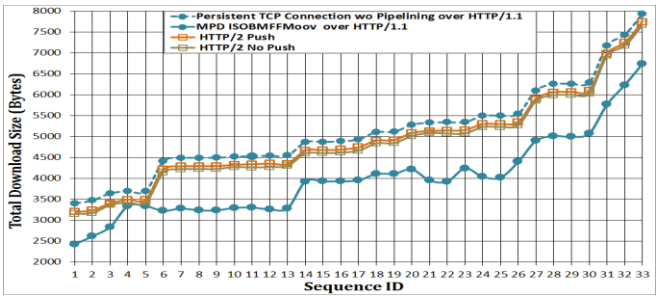

Fig. 3 Total download size for each strategy

We then compared with HTTP/2 with and without push. The total HTTP/2 download size is obtained from the transfer size field in the Network Panel of Google Chrome. As we can see in Fig. 3, using our method, over HTTP/2 with header compression, the total download size is reduced by approximately 25% compared to the amount of data being downloaded for the persistent TCP connection approach, and 20% compared to the HTTP/2 push method.

B. Bootstrap delay

We now compare, in terms of bootstrap delay, our ISOBMFFMoov-based approach first to the persistent TCP connection without pipelining based on HTTP/1.1 and then to the HTTP/2 server push approach.

For the persistent connection without pipelining strategy, we measured using Google Chrome Network Panel the elapsed time between when the Dash-JS player establishes a TCP connection to request the MPD from the web server and when it receives the last byte of the last IS. The MPD processing time by Dash-JS as reported by Chrome is not included in this measurement. Additionally, we also measured the time to download the ISOBMFFMoov-based MPD.

Fig. 4 shows these measurements when the downloads were made over an Ethernet network, using HTTP/1.1, with varying TCP's init_cwnd (3 and 10 TCP segments). The DASHIF sequences are sorted according to the total download size measured in the previous experiment when the IS and MPD are delivered separately over a single persistent TCP connection.

These results show first that the bootstrap delay using our approach is decreased by around 2 RTT (100 ms) compared to the persistent approach used by Dash-JS player. These 2 RTT are due to the two request-response cycles that the Dash-JS player needs to retrieve the video and audio IS to start the initial playback.

We notice also that the bootstrap delay when init_cwnd is set to 3 TCP segments seems stable for almost all sequences using our approach. From the 26th to the 33th sequence, the delay is increased by 1 RTT (50 ms). This is explained by the fact that the number of TCP segments allowed in the init_cwnd (3 TCP segments about 4380 bytes) is not sufficient to fit the entire MPD with embedded IS information. In this case, the TCP slow start algorithm requires waiting for acknowledgements to arrive before new data is sent which induces an additional RTT. The size of these eight sequences

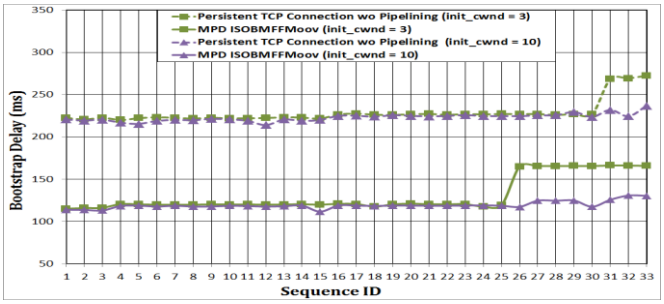

Fig. 4 Bootstrap delay measured for the ISOBMFFMoov-based approach and persistent TCP connection without pipelining approach over HTTP/1.1

is increased because they are packaged using the "avc1" mode and therefore, the base64 encoded "stsd" box is larger. We observe the same behaviour for the persistent approach used by Dash-JS except that the bootstrap delay is increased by 1 RTT (50 ms) from the 30th to the 33th sequence. This is due to the MPD size that exceeds the init_cwnd size for these sequences.

When increasing the init_cwnd to 10 TCP segments (approximately 14 600 bytes), the bootstrap time is stable for all sequences for both approaches. This is because the size of the downloaded resources is less than the init_cwnd size.

Finally, it should be noted that our approach is more efficient when the number of IS chosen by the client (X_C) is closer to X, i.e. when the MPD contains several adaptation sets with different content types. We presented here a worst case, with only 2 adaptation sets.

Beside the experiment measurements, we also computed the theoretical download time for those resources for both approaches according to the formulas shown in Table II. The download time of a resource in the slow start phase without losses is given below [11]:

$$D_{ss}(S) = \left\lceil \log_{\gamma}\left(\frac{S(\gamma-1)}{init\_cwnd}+1\right) \right\rceil * RTT + \frac{S}{C} \quad (1)$$

where S is the download resource size, γ is set to 2 because we assumed no delayed acknowledgments and C is the network capacity. D_{SS} is composed of the number of RTT required to transfer data in the slow start plus the transmission delay. We compared the theoretical bootstrap delay with the real one for both approaches and we confirmed that they are approximately identical which proved the reliability of our experiments.

We finally measured the bootstrap delay using our ISOBMFFMoov-based approach and the server push method over HTTP/2. The server push is enabled on the web server and on the client, and the server is aware of the all IS to push for a given MPD.

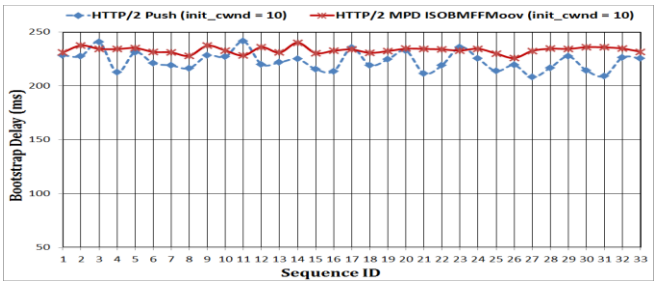

Fig. 5 Bootstrap Delay on HTTP/2

The results in Fig. 5 show that both methods take 3 RTT: one RTT for the TCP handshake, one RTT for the TLS handshake, and one RTT for the only MPD request. Both methods provide similar results with a slight advantage for the HTTP/2 push approach despite the fact that the download size of our approach is smaller. After deep inspection with Wireshark, we suspect a problem with the NodeJS server that we will investigate in the future.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have reviewed the possible cause of latency in the bootstrap phase of a DASH session, analysed the different strategies currently used by existing implementations to limit this delay, using the different options offered by the HTTP and TCP layers, and provided an analytical evaluation of their bootstrap delay. We then have proposed a more compact method for transmitting the MPD and initialization data to the client, and compared the different methods, showing that the total download size is reduced by 25% compared to the amount of data being downloaded for the persistent TCP connection without pipelining strategy and 20% versus HTTP/2 push method. Furthermore, we show a gain of 2 RTTs in HTTP/1.x and no penalty when using HTTP/2. This suggests that our approach can be valuable even during the transition phase to HTTP/2. Future works include evaluating the impact on our proposal in player implementation and on session switching delays.

REFERENCES

[1] Sandvine, "Global Internet phenomena report 1H 2014", Sandvine Intelligent Broadband Networks, 2014.

[2] I. Sodagar, "The MPEG-DASH standard for multimedia streaming over the Internet," Transactions on MultiMedia, IEEE, vol. 18, no. 4, pp. 62–67, April 2011, doi: 10.1109/MMUL.2011.71.

[3] S. Egger, T. Hossfeld, R. Schatz and M. Fiedler, "Waiting times in quality of experience for web based services," QoMEX 2012, Yarra Valley, Australia, July 2012.

[4] Conviva, "Viewer experience report," February 2013, http://www.conviva.com/reports/Viewer_Experience_Report.pdf

[5] N.Bouzakaria, C. Concolato and J. Le Feuvre, "Overhead and performance of low latency live streaming using DASH," In Proc. Of IEEE IISA 2014 Greece. pp. 92-97.

[6] N. Dukkipati, et al. "An argument for increasing TCP's initial congestion window," SIGCOMM Computer Communication Rev., ACM, 2010, 40, 26-33.

[7] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain and B. Raghavan, "TCP fast open," Proceedings of the Seventh Conference on Emerging Networking EXperiments and Technologies, ACM, 2011, 21:1-21:12.

[8] S. Wei, and V. Swaminathan, "Low latency live video streaming over HTTP 2.0," Proc of Network and Operating System Support on Digital Audio and Video Workshop, ACM, 2014, 37:37-37:42.

[9] W. Chérif, Y. Fablet, E. Nassor, J. Taquet, Y. Fujimori, "DASH fast start using HTTP/2", NOSSDAV, 2015, 25-30

[10] I. Grigorik, "High performance browser networking," O'Reilly Media, May 2013.

[11] N. Cardwell, S. Savage and T. Anderson. "Modeling TCP latency," in IEEE INFOCOM, 2000, 1724-1751