HTTP/2-Based Methods to Improve the Live Experience of Adaptive Streaming

Rafael Huysegems, Tom Bostoen, Patrice Rondao Alfface
Bell Labs, Alcatel-Lucent
Copernicuslaan 50
B-2018 Antwerp, Belgium
rafael.huysegems@alcatel-lucent.com

Jeroen van der Hooft, Stefano Petrangeli, Tim Wauters, Filip De Turck
Ghent University - iMinds
Gaston Crommenlaan 8/201
B-9050 Ghent, Belgium
jeroen.vanderhooft@intec.ugent.be

ABSTRACT

HTTP Adaptive Streaming (HAS) is today the number one video technology for over-the-top video distribution. In HAS, video content is temporally divided into multiple segments and encoded at different quality levels. A client selects and retrieves per segment the most suited quality version to create a seamless playout. Despite the ability of HAS to deal with changing network conditions, HAS-based live streaming often suffers from freezes in the playout due to buffer under-run, low average quality, large camera-to-display delay, and large initial/channel-change delay. Recently, IETF has standardized HTTP/2, a new version of the HTTP protocol that provides new features for reducing the page load time in Web browsing. In this paper, we present ten novel HTTP/2-based methods to improve the quality of experience of HAS. Our main contribution is the design and evaluation of a push-based approach for live streaming in which *super-short* segments are pushed from server to client as soon as they become available. We show that with an RTT of 300 ms, this approach can reduce the average server-to-display delay by 90.1 % and the average start-up delay by 40.1 %.

Categories and Subject Descriptors

H.m [**Information Systems**]: Miscellaneous

Keywords

HTTP adaptive streaming, DASH, HTTP/2, SPDY, video streaming, live streaming, latency, server push

1. INTRODUCTION

Over the last years, the delivery of multimedia content became more prominent than ever. Particularly, video streaming applications are now responsible for more than half of the Internet traffic [30]. To enable video streaming over the best-effort Internet, the concept of HTTP Adaptive Streaming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MM '15, October 26-30, 2015, Brisbane, Australia
© 2015 ACM. ISBN 978-1-4503-3459-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2733373.2806264.

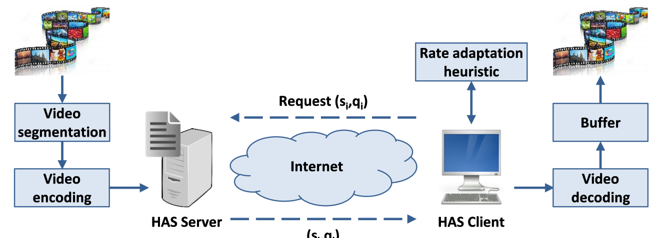


Figure 1: The concept of HTTP Adaptive Streaming.

(HAS) was introduced. As shown in Figure 1, video content is encoded at different quality levels. Each quality level is determined by its corresponding average bitrate. In addition, the content is divided in segments that have a typical duration of one to ten seconds. Each segment can be decoded independently of other segments. A HAS client initiates a new session by downloading a manifest file. This manifest file provides a description of the different available quality levels and segments. Based on the network conditions and the current buffer-filling level, the Rate Determination Algorithm (RDA) in the HAS client determines the quality for the next segment download. The objective of the RDA is to optimize the global Quality of Experience (QoE) determined by the occurrence of video freezes, the average quality level, and the frequency of quality changes. The main advantage of HAS over progressive download and traditional real-time streaming is its ability to adapt the video quality to the available bandwidth in order to avoid video freezes. As a consequence, HAS facilitates video streaming over a best-effort network. In addition, HTTP-based video streams can easily traverse firewalls and reuse the already deployed HTTP infrastructure such as HTTP servers, HTTP proxies, and Content Delivery Network (CDN) nodes. Because of these advantages, major players such as Microsoft, Apple, Adobe and Netflix massively adopted the adaptive streaming paradigm. As most HAS solutions use the same architecture, the Motion Picture Expert Group (MPEG) proposed a standard called Dynamic Adaptive Streaming over HTTP (DASH) [31].

Despite the many HAS advantages, several inefficiencies still have to be solved to improve the user's QoE, especially in live video streaming.

- Video freezes caused by rebuffering: Dobrian et al. [11] show that rebuffering has the largest impact on user engagement. Conviva [10] estimates the number of HAS sessions that suffer from video freezes at 27

percent. Especially in environments with rapid bandwidth changes such as mobile networks, the client may have insufficient time to adapt and may need to rebuffer [24]. Rebuffering could be avoided by using larger client buffers but unfortunately this increases also the end-to-end delay for live streaming.

- Low average video quality: In [10], the number of sessions impacted by low resolution is estimated at 43 percent.

- Frequent video-quality changes: Subjective tests described in [22] and [32] show that frequent and abrupt fluctuations in the selected video-quality levels as well as video-quality oscillations have a negative impact on the QoE.

- Large camera-to-display delay: Lohmar et al. [21] show that the total delay from camera to display in live streaming should be kept as small as possible. The viewing experience of a live soccer game can be spoiled by social media or neighbors cheering for a goal when this goal was not yet shown on the screen. In current HAS deployments, this camera-to-display delay is in the order of tens of seconds.

- Large interactivity delay: Such delay is defined as the waiting time for the viewer when interacting with the HAS client. We distinguish start-up delay, seeking delay, and channel change delay.

Our goal is to reduce the number of video freezes, increase the average video quality, reduce the number of quality-level changes, reduce the latency for live streaming, and reduce the interactivity delay by using new HTTP/2 features. Early 2012, the IETF httpbis working group [18] started the standardization of HTTP/2, to address a number of deficiencies in HTTP/1.1. In February 2015, the new HTTP/2 standard was published as an IETF RFC [2] and is now supported by major browsers such as Chrome, Firefox and Internet Explorer. The main focus of HTTP/2 is to reduce the latency in Web delivery, using four new features that provide possibilities to terminate the transmission of certain content, multiplex several requests, prioritize more important content, and push content from server to client.

The main contributions of this paper are threefold. As a first contribution, we propose ten novel methods to improve the QoE of HAS, based on HTTP/2 features. Two methods are based on request/response multiplexing and prioritization. These HTTP/2 features enable simultaneous requests for subsequent segments or segment layers in the case of Scalable Video Coding (SVC) to avoid the per-segment/per-layer round-trip time (RTT). Another two methods are based on the cabability of HTTP/2 to abort a stream (and the corresponding segment retrieval). This capability is used when a channel change is detected or when the bandwidth suddenly drops while downloading a too high quality segment. Six more methods use HTTP/2's server push. To avoid RTT cycles, an HTTP/2-based server can push manifests, segments, or segment layers to the client when the server knows the client will request these in the near future. The server can also push segments or segment layers additional to or as an alternative for segments or segment layers requested by the client as network conditions require. In the *full push* method, the server continuously pushes new segments to the client when they become available. The client sends HTTP messages to the server to start, stop, pause, or resume the stream or to request a quality-level change. By eliminating wait cycles such as the RTT between subsequent segment retrievals and the client polling time, this method improves the average selected video quality. As a second contribution, we show that HTTP/2 server-push can also be used to improve the QoE for live streaming by alleviating the disadvantages linked to the use of super-short segments. This allows us to bring the segment duration to a sub-second level. In addition, we propose a method to solve the encoding overhead induced by super-short segments. This combination results in a significantly reduced start-up and end-to-end delay for live video. As a third contribution, we present detailed experimental results based on an emulation-based setup with full push and super-short segments to characterize the potential of the presented solution compared to traditional HAS clients.

The remainder of this paper is organized as follows. Section 2 gives an overview of the related work on HAS and HTTP/2. In Section 3, we provide a root-cause analysis of the HAS QoE problems. Section 4 introduces ten novel methods to improve the QoE of HAS and describes how a combination of the full-push method and super-short-segments results in a superior QoE for live HAS streaming. An evaluation of the results is presented in Section 5. Next, section 6 defines future work, before coming to final conclusions in Section 7.

2. RELATED WORK

2.1 HTTP Adaptive streaming

Techniques to improve the QoE for HAS can be divided in three main classes.

The first class focuses on the client's RDA. E.g., Benno et al. [3] propose a more robust RDA for wireless live streaming. Claeys et al. [9] propose an RDA that dynamically learns the optimal behavior for the corresponding network environment.

The second class uses SVC to encode the segments. Famaey et al. [14] and Sanchez et al. [29] compare SVC-based HAS to AVC-based HAS. Because an SVC client has an increased number of decision points, such a client copes better with a highly variable bandwidth as in mobile scenarios. Although SVC reduces the footprint for storage, caching, and transport compared to a complete simulcast AVC system, the SVC coding adds an overhead of 12 percent for a single medium-quality stream and 26 percent when delivering a single high-quality stream. For VoD, Sánchez et al. [28] show that when both the cache feeder link and the access links are congested, SVC-based HAS leads to an improved QoE. Muller et al. [25] compare AVC-based DASH with SVC-based DASH in a constrained environment such as a mobile network. They conclude that the flexibility of SVC's layered coding structure allows for a more aggressive buffer model. Bouten et al. [6] use Differentiated Services (DiffServ) in the IP network to give priority to the base-layer (BL) segments. Because the SVC-based client is more robust to video freezes, it is possible to reduce the client buffer from 6 to 30 seconds in AVC-based HAS to 2 seconds. However, the practical application of SVC in live streaming is still under question because SVC introduces a significant

encoding overhead and increases the complexity of the decoder.

The third class explores how the QoE can be improved by making the network and server HAS aware. In current HAS deployments, the greedy QoE optimization prevents clients to reach a globally optimal distribution of resources and QoE. Bouten et al. [7] and Petrangeli et al. [26] avoid such suboptimal distribution by introducing intelligence in the service-provider network that can help the client's local RDA decisions. Akhshabi et al. [1] propose to use server-based traffic shaping to reduce video quality oscillations at the client.

2.2 HTTP/2

In this paper, we target an improved QoE of HAS by using HTTP/2. This new version of HTTP is based on SPDY. The main objective of SPDY is to speed up Web browsing. According to Google [16], a reduction in page load time of up to 64 percent can be achieved. Other studies show that the mere replacement of HTTP by SPDY helps only marginally. Cardaci et al. [8] evaluate SPDY over high latency satellite channels. On average, SPDY slightly outperforms HTTP. Erman et al. [13] provide a detailed measurement study to understand the benefits of using SPDY over cellular networks. They report that SPDY does not clearly outperform HTTP due to cross-layer dependencies between TCP and the cellular network technology. Elkhatib et al. [12] conclude that SPDY may both decrease as well as increase page load times. SPDY's multiplexed connections last much longer than HTTP's. Such long-lasting connections make SPDY more susceptible to packet loss, which gives rise to problems with TCP backoff.

2.3 HAS over HTTP/2

Mueller et al. are the first to evaluate the performance of adaptive streaming over SPDY, more specifically for the DASH standard [23]. The existing HTTP/1.0 or HTTP/1.1 layer is replaced by SPDY, without any further modifications to the HAS client or server. The authors show that if SPDY is used over SSL, the gains obtained by using header compression, a persistent connection, and pipelining are almost completely cancelled out by the losses due to the SSL and framing overhead.

Wei et al. are the first to explore how new HTTP/2 features can be used to improve HAS [37]. By reducing the segment duration from five seconds to one second, they manage to reduce the camera-to-display delay by about ten seconds. They avoid an increased number of GET requests by pushing k segments after each request, using HTTP/2 server push. This approach has the disadvantage that when a client switches to another quality level, the pushed stream using the old quality level is in competition with the segments downloaded at the new quality level. This increases the switching delay for the client and the bandwidth overhead in the network. In later work, the authors show that the induced switching delay is about two segment durations and independent of the value of k, while the introduced bandwidth overhead heavily depends on this value [36]. Moreover, HTTP/2 functionalities are used to push audio segments upon receiving a request for the associated video segments.

3. ROOT CAUSES OF HAS QOE PROBLEMS

To better understand possible HAS improvements, we first identify four root causes of the HAS QoE problems that are presented in Section 1.

A first deficiency of HAS is its susceptibility to large RTTs. Because segments are fetched sequentially, an RTT is lost between subsequent transfers. This has following effects: (1) At start-up, the HAS client has to retrieve several objects (manifest, segment, ...) before the playout can be started. When the RTT is large, the lost RTT cycles significantly increase the initial playout delay. (2) One RTT is lost between subsequent segment retrievals, possibly leading to a poor link utilization and average quality level. Both effects are stronger for short segments and large RTTs. In 2013, [15] measured an average RTT during peak periods of 18 ms for fiber-to-the home, 26 ms for cable, and 44 ms for DSL connections. When using a Content Delivery Network (CDN), [19] estimates the RTT as 51 ms for fiber-to-the home, 67 ms for cable, 238 ms for DSL, 250 ms for 4G, and 550 ms for 3G. If the requested object is not found in the CDN node, an additional 135 ms must be added, e.g., when the CDN node is located on the US West Coast and the origin server on the US East Coast. For Wi-Fi, 6 ms to 100 ms must be added. These figures indicate that the RTT can have a significant impact on the QoE for HAS, especially for wireless connections.

A second deficiency is the pull-based nature of HAS for live streams. At the client, the polling for new segments introduces an additional wait time during which the available bandwidth remains unused. This idle time reduces the link utilization and increases the live delay.

A third HAS deficiency is caused by the client buffer. For live streaming, this buffer is typically 3 to 5 times the segment duration and accounts for a significant part of the end-to-end delay.

A fourth HAS deficiency is the inability of HAS (and HTTP/1.1 in general) to terminate an ongoing segment transfer. This affects the QoE in two situations: (1) If a client just started a segment download when the user requests a channel change, the client must first complete the entire download of the obsolete segment causing significant channel change delay. (2) If a client started the download of a high-quality segment when the available bandwidth suddenly decreases, the client must first complete the entire download. This condition often leads to video freezes.

4. HTTP/2-BASED QOE-IMPROVEMENT METHODS

In this section, we propose ten novel or improved HTTP/2 based methods that enhance the QoE of HAS as presented in Table 1. To implement these methods, it may be required to modify the server, the client, or both. In this paper, the term *server* may refer to an origin server, an HTTP proxy, or a CDN node.

4.1 Stream Termination

HTTP/2 enables the use of multiple streams within one TCP connection. This feature makes it possible to use a separate stream per segment download. At any moment, the client can terminate an active stream by sending an RST_STREAM frame to indicate that the stream is no longer needed [2]. Upon reception of such a frame, the server stops

Method	HTTP/2 features used	QoE impact	Relation with other methods	Required updates
High-quality segment termination	stream termination	freeze	none	client
Obsolete segment termination	stream termination	channel change delay	none	client
Pull many	request/response multiplex, stream prioritization	average quality, freeze	none	client
Pull many with termination	request/response multiplex, stream prioritization, stream termination	average quality, freeze	on top of pull-many method	client
Safety push	push, stream prioritization	freeze	none	server, (client)
Initial push	push	initial delay, channel change delay	none	server, (client)
Layered push	push, stream prioritization	average quality	none	server
Manifest push	push	average quality	none	server
Overruling push	push	freeze	none	server, client
Full push	push	average quality, initial delay, channel change delay, live delay	includes methods: initial push, manifest push	server, (client)

Table 1: Overview of the different methods to improve the QoE for live HAS streams.

the ongoing transmission and closes the stream. This feature is actively exploited in the methods detailed in the following sections. Note that the advantages of these methods increase when the RTT is small, because the terminate message is conveyed faster from the client to the server. As a result, the occupied resources are released earlier and can be reused for the transport of a new segment.

4.1.1 High-Quality-Segment Termination

When the network conditions deteriorate severely, the continuation of a high-quality segment download takes a long time. Such a slow download could lead to a video freeze. By using HTTP/2 stream termination, the client can actively terminate the ongoing high-quality segment transfer. This way, network resources are released sooner and the client can start the download of a segment encoded at a lower quality level faster.

4.1.2 Obsolete-Segment Termination

When a user switches between video channels, the segment that is currently under transfer can no longer be used in the video playout. By terminating the transfer of the obsolete segment, the client can immediately request the first segment(s) of the new channel. Because network resources are released sooner, the channel change delay can be significantly reduced.

4.2 Request/Response Multiplexing and Stream Prioritization

In HTTP/2, multiple streams can be used to download segments in parallel over one TCP connection. To orchestrate the segment delivery, the client can specify that a new stream depends on a previous one. In this way, the client expresses a preference to allocate server processing and bandwidth resources to the parent stream rather than to the dependent stream. In the methods described below, the HAS client requests multiple segments at once to avoid lost RTT cycles between subsequent retrievals. Via HTTP/2 dependencies, the client ensures that segments that are required first for playout will be received first. Because lost RTT cycles are avoided, such methods increase the link utilization and the average video quality.

4.2.1 Pull Many

In this method, the client sends simultaneously requests for n subsequent segment requests. The delivery order is imposed by the client using HTTP/2 stream dependencies.

When SVC encoding is applied, the client downloads different quality layers to reconstruct one playable segment. As a result, multiple RTTs are lost per playable segment, leading to a lower link utilization and a lower average quality. In the *pull many* method, the different layers composing a segment are requested all at once. Using stream dependencies, the client makes sure that the base layer and the different enhancement layers are received in the correct order. Just as for standard SVC, the main advantage of this approach is that when a segment is urgently required for playout, and a higher layer is still under transfer, the base layer is already available. This way, playout freezes can be avoided.

4.2.2 Pull Many with Termination

This method builds on the pull-many method. Traditional RDAs predict the most appropriate quality level for the next segment download. In our approach, the client requests both the base layer and all enhancement layers for each segment. This triggers the server to send all the quality layers in a back-to-back fashion. The advantage is that at any moment during the transfer of the segments, the client can cancel the download of additional enhancement layers via stream termination. This happens if the bandwidth is not sufficient to transport all layers in time for playout. With this method, instead of predicting the optimal quality one segment in advance, the client simply adapts to the current bandwidth conditions with a delay of one RTT required to terminate the stream.

4.3 Server Push

HTTP/2 enables a server to preemptively send (push) responses to a client based on a previous request from the client. Especially for live streaming, using a push-based approach has multiple advantages. First, because subsequent segments can be pushed back-to-back, lost RTT cycles between such segments are avoided, increasing the average link utilization and video quality. Second, a reduction of the live

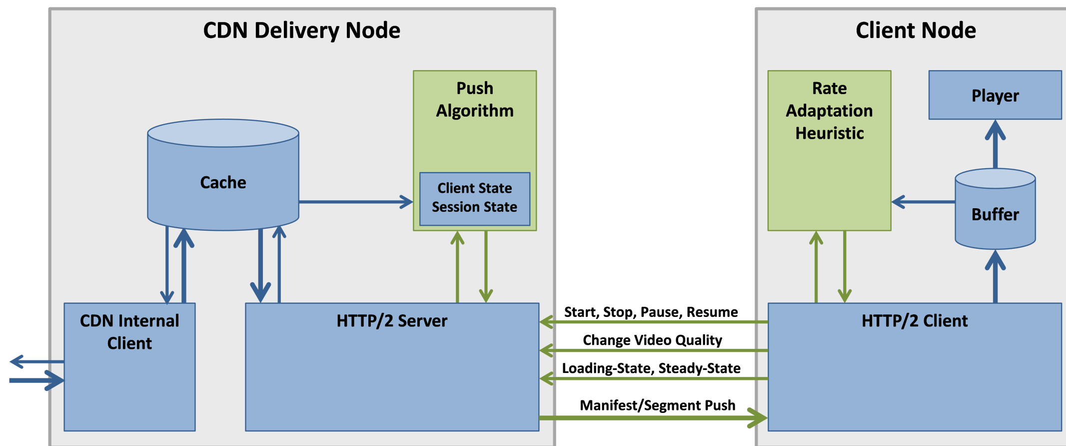


Figure 2: Implementation of the full push method in a CDN environment with the modified RDA at the client and the added push algorithm at the CDN delivery node.

delay can be obtained, because video data is pushed as soon as it becomes available at the server. This is in contrast with traditional pull-based HAS approaches, where client-based polling is required to retrieve new segments. However, a legacy browser architecture using client plug-ins can pose restrictions on the application of push methods for HAS because the browser's API does not offer push support. When a video segment or manifest is received by the browser via push, the client plug-in is not informed. This has two possible effects. First, if the client's RDA selects another quality level for the same segment, the already pushed segment can't be used. Second, transfers are still performed based on client polling but with the advantage that already pushed objects are served immediately from the local cache in the browser as soon as the client issues a new HTTP GET request.

4.3.1 Safety Push

When the client buffer drops below a certain threshold, the client typically enters the panic mode, in which the lowest quality level is retrieved. Using the *safety push* method, the server pushes the lowest quality level of a requested segment. This push is performed before the transfer of the segment at the requested quality level. Consequently, if the client enters the panic mode, the lowest quality is already available, avoiding buffer starvation and the related video freeze. The overhead introduced by this method is only acceptable if the bit rate of the highest quality is several orders of magnitude higher than the bit rate of the lowest quality. The *safety push* may be applied (1) continuously, (2) only when the client recently reduced the requested quality level, or (3) only when the client's buffer filling drops below a certain threshold. The buffer filling can be conveyed from the client to the server in the HTTP request or deduced by the server via session reconstruction [17].

4.3.2 Initial Push

In this method, all objects required to start the playout of the content such as (an) additional manifest file(s), DRM objects and the first segment(s) are pushed when the client requests a manifest from the server. For this purpose, the server is aware of the relationship between the different HAS objects. Instead of requesting objects sequentially and losing one RTT per object, the client receives all the required data together and starts the playout immediately, resulting in a significant reduction of the start-up time. For VoD streaming, the server pushes the first segment. For live streaming, the server pushes a segment that was made available d seconds ago such that the client can build a buffer that is maximum d seconds long. The value of d used by the server can be fixed for a particular type of client or specified by the client in the manifest request.

4.3.3 Layered Push

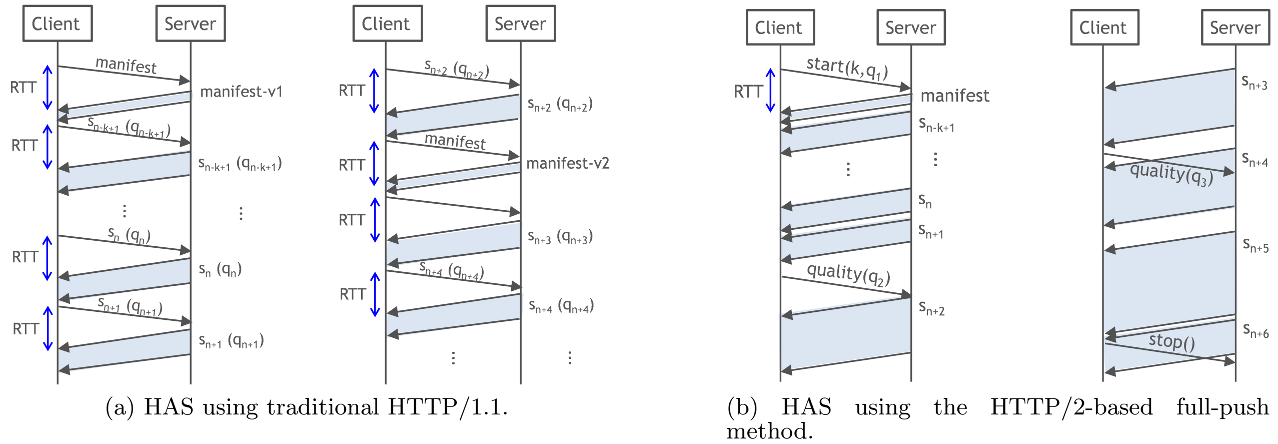
In the *layered push* method, an SVC-based client requests only the highest enhancement layer required for the desired quality level. By knowing the dependencies between the different layers, the server pushes the required lower layers autonomously. As a result, all layers are delivered using one RTT cycle, increasing the link utilization and the average video quality.

4.3.4 Manifest Push

This method is applicable to live clients that require a regular update of the manifest file such as Apple HLS clients. These clients rely on polling to periodically fetch a new manifest file. The client parses the manifest file and checks if a new segment was added. If so, the client downloads the new segment. Every segment cycle, the line remains idle for two RTT cycles (manifest and segment retrieval) and the polling time for the manifest (time between the availability of the new manifest at the server and the retrieval of the manifest by the client). Using the *manifest push* method, the server pushes the manifest as soon as a new version becomes available at the server. As a result, the idle time per cycle is reduced to one RTT to retrieve the segment. Based on previous segment retrievals, the server identifies clients that benefit from this method. For this reason, the server has to keep state about the ongoing sessions.

4.3.5 Overruling Push

In HAS, the decision on which quality level to download is made by the client's RDA. The RDA takes this decision based on the current buffer filling and a prediction of the available bandwidth for the next download. This prediction is based on the perceived bandwidth from server to client during the download of previous segments. However, there



(a) HAS using traditional HTTP/1.1.

(b) HAS using the HTTP/2-based full-push method.

Figure 3: Sequence diagram for HAS live streaming with and without the HTTP/2-based full-push method.

are situations where the previously perceived bandwidth is a poor estimation of the future bandwidth. It is also possible that the server (e.g., a CDN delivery node) has better information on the available bandwidth. The goal of the *overruling push* method is to provide a possibility for the server to influence the quality selection process of the client. This can be done by pushing a segment to the client at a (typically lower) quality level. In this case, the client adds the segment to the playout buffer and refrains from requesting the same segment in another (higher) quality level.

4.3.6 Full Push

Figure 2 shows an application of the *full-push* method whereby segments (and manifests) are continuously pushed from the server (CDN delivery node) to the client. The client controls the session by sending start, stop, pause, or resume messages under the form of an HTTP GET or HTTP POST to the server. Each time a segment is completely received by the client, the RDA determines the perceived bandwidth and the actual buffer filling level. Based on these parameters, it can send a request to change the applied quality level for the pushed segments. As soon as the server receives such a request, it will change the quality starting with the next segment to be pushed. In case of VoD, the client also informs the server about the state of its buffer by sending loading-state or steady-state messages, indicating respectively that segments must be sent back-to-back or that segments must be sent at the playout rate to avoid buffer overflow. In case of live streaming as illustrated in Figure 3, the client can specify in the start-message the initial quality level and the start segment in terms of a number of segments k before the last available segment. The server will then push back-to-back all segments from the start segment s_{n-k+1} up to the last available segment s_n of the live stream. From that moment, the server will send a new segment each time when it becomes available. To do this, the server keeps track of clients and sessions.

4.4 Application in Live Streaming

Compared to traditional UDP-based video streaming such as RTP, HAS exhibits advantages such as easy firewall traversal, reuse of existing HTTP infrastructure, and adaptability. However, real-time UDP-based streaming still exhibits some advantages over HAS: a higher bandwidth utilization, the capability to use multicast to reduce the network load, a lower start and channel-change delay, and a lower end-to-end delay for live streaming.

To improve the QoE for HAS live streaming, a shorter segment duration could be used. This has following advantages: (1) Short segments are sooner available at the client because less time is required for encoding at the head-end and for transport. This results in a reduced live delay. (2) When the client starts a session, the first segment needs less time for transport, resulting in a reduced initial delay and a faster response to channel change. (3) At a channel change, there is less obsolete video data in the pipeline from server to client that still needs to be received by the client. As a result, the channel-change delay is further reduced. (4) Short segments increase the number of measuring and decision points for the RDA. In case of deteriorating network conditions, the RDA is able to react faster, possibly preventing a freeze.

Unfortunately, it is generally infeasible to use super-short segments in current HAS systems because of the following reasons: (1) More HTTP-GET messages are required to retrieve the segments, resulting in a larger overhead for the server and network. (2) Because more segments must be retrieved, more RTT cycles are lost between subsequent requests. This reduces the link utilization and the average quality. (3) In HAS, every segment starts with an IDR frame in order to be decoded independently of other segments. For video coding standards such as H.264/AVC and High Efficiency Video Coding (HEVC), the encoding efficiency of an IDR frame is significantly lower than the efficiency of frames composed of P or B slices. As a result, a higher bitrate is required to reach the same quality as for longer segments. A standard way to estimate this bitrate overhead required to achieve an equivalent average PSNR score over a set of segments is to measure the Bjontegaard Delta rate [5] for several test sequences. The estimated overheads for various segment lengths are reported in Section 5.

However, the drawbacks of short segments can be mitigated by applying additional methods. (1) The increased overhead of HTTP-GET messages can be avoided by using the *full-push* method. (2) Similarly, the problem caused by the additional lost RTT cycles between subsequent segment retrievals can be avoided by using full push since segments are pushed back-to-back. (3) The problem of the lost encoding efficiency could be mitigated by using *quality-level dependent encoding*. Using this type of encoding, the high quality levels contain less IDR frames than the lower qual-

ity levels. Since all segments have the same duration in every level, not every segment in the highest level starts with an IDR frame. Consequently, upwards quality switching requires a high quality segment that starts with an IDR frame. Downward quality switching can be done at every segment since every low-quality segment starts with an IDR frame. This type of encoding matches very well with the HAS characteristics since a HAS client must be able to switch down quickly to avoid a freeze. However, clients act very conservatively when increasing the quality. Typically, a client waits several seconds, evaluating the available bandwidth, before it decides to increase the quality. For now, quality-level dependent encoding was left as future work. Instead, we based our evaluation on the typical HAS encoding scheme where every segment starts with an IDR frame.

5. EVALUATION

To illustrate the possible gains of an HTTP/2-based push approach with super-short segments, we implemented the full-push scheme proposed in Section 4 and evaluated results for different network conditions and segment durations. The experimental setup is discussed below, followed by a detailed overview of the obtained results.

5.1 Experimental Setup

The applied video sequence is the *Big Buck Bunny* video, which has a total length of 596.4 seconds. This video is segmented using five segment durations: 133, 266, 500, 1000, and 2000 ms. Furthermore, the video is encoded in H.264 using seven quality levels: 1, 1.7, 3, 5, 8, 12.5, and 20 Mb/s. Because shorter segments are encoded less efficiently compared to the 2 second segments, we modified these nominal bit rates to reflect the overhead of the additional IDR encoded frames. Considering segments of 2 seconds as the reference, the estimated average bit rate was increased with 0.5% for 1 second segments, 2.16% for 500 ms segments, 4.89% for 266 ms segments and 9.84% for 133 ms segments, respectively. While these overheads enable to encode the segments at the same average PSNR level, the perceived quality impact of experiencing a higher frequency of IDR frames can be investigated in future work. For a fair comparison of the obtained average quality levels in experiments that use a different segment duration, we refer in our results to the normalized bit rate, since this value represents the quality level as perceived by the user.

To evaluate the proposed approach, the network topology in Figure 4 is emulated using the MiniNet framework on the Virtual Wall, a testbed containing 300 physical servers [33, 20]. The topology consists of a single HAS client, streaming video from a dedicated HAS server. The bandwidth between client and server is limited to 25 Mb/s on link 2, which is sufficient to provide the content at the highest quality level. To use HTTP/2's server push, modifications to the server and client are required, which are summarized below.

5.1.1 Server-Side Implementation

The HAS server implementation is based on the Jetty web server, which was extended to provide support for HTTP/2 [35]. Jetty's HTTP/2 component allows to setup a push-based strategy, which defines all resources that need to be pushed along with the requested resource. Such a strategy is ideal for web-based content, where the required JavaScript and CSS files, images, and other content can immediately be

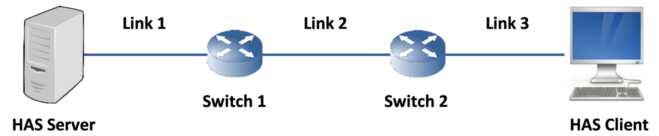


Figure 4: Emulated network topology.

pushed. However, since we target a live-stream scenario, not all segments are available when a request is issued. Therefore, we defined a new request handler that processes GET requests issued by the client. This handler allows a client to issue a live-stream request, passing along parameters such as the preferred buffer size and quality level. When this request corresponds to a new session, the server starts a push thread that pushes instantly the five last released video segments at the lowest quality. This way, the client can ramp up its buffer quickly, while not overloading the network. In order to simulate a live stream scenario, a release thread makes new segments available every segment duration. As soon as new segments are available, the push thread is notified and a segment is pushed to the corresponding client. When the client wants to change the quality level at which the segments are pushed, a new GET request is issued and the quality level is updated at server side accordingly.

5.1.2 Client-Side Implementation

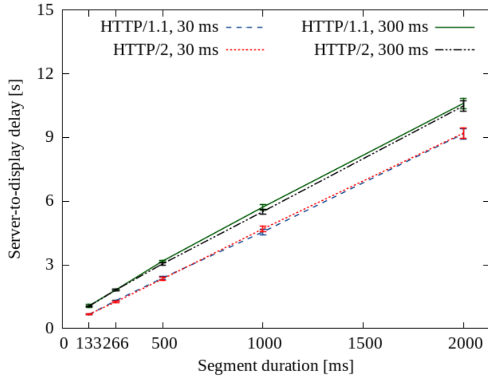
The HAS client is implemented on top of the libdash library, the official reference software of the ISO/IEC MPEG-DASH standard [4]. To make use of HTTP/2's server pushing feature, a number of changes is made. First, an HTTP/2-based connection is added to enable the reception of pushed segments. The nghttp2 library is used to set up an HTTP/2 connection over SSL [34]. Note that the reference software uses curl to issue all GET requests, yet this library does not yet support HTTP/2 push. Second, the RDA is modified to recalculate the quality level every time a segment is received and the corresponding push stream is closed. While in HTTP/1.1 a GET request is required for every segment, no request is sent in the HTTP/2-based scheme if no quality change is required. Third, the perceived bandwidth is estimated based on the elapsed time between the reception of the push promise and the time the segment is available. In HTTP/1.1, this estimation is based on the total download time, which includes the time to send the GET request.

The used RDA is the FINEAS RDA[1], developed by Petrangeli et al. [27]. It can deal with low buffer sizes and has shown to outperform well-known RDA's such as MSS. The used RDA parameters are a quality window of 70 seconds, a panic threshold of two segments and a buffer target of 80%. These values were selected after tweaking the RDA for a buffer size of five segments. Note that our main goal is to show the possible gain obtained by using HTTP/2 with super-short segments instead of HTTP/1.1; a comparison of the performance of different RDA's using HTTP/2 will be left as future work.

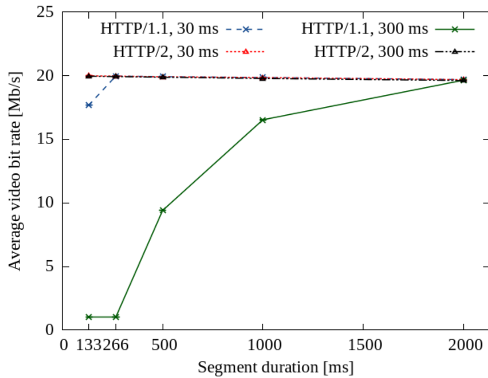
5.2 Evaluation Metrics

The following evaluation metrics are considered: (i) the average video quality, expressed as the average normalized bit rate of all video segments; (ii) the server-to-display delay, defined as the time between the release of a segment and its

[1]The in-network computation proposed by the authors has not been implemented in this work.



(a) Average server-to-display delay



(b) Average video bit rate

Figure 5: Impact of the segment duration on the average normalized bit rate and the average server-to-display delay, both for an RTT of 30 ms and 300 ms.

playout at client-side. Note that this is not the same as the camera-to-display delay, as no real live event is captured in our experiments; (iii) the initial start-up delay, which is defined as the time between requesting the live video at client-side and the playout of the first video segment.

To properly evaluate the impact of the segment duration and the RTT on these metrics, the *Big Buck Bunny* video was streamed multiple times. In our experiments, all five segment durations were tested using HTTP/1.1 and HTTP/2 over SSL, with realistic RTTs in the range of 30 ms to 300 ms. To account for possible outliers, experiments were repeated multiple times for every configuration. Results are therefore shown using the observed averages and the appropriate 95% confidence intervals.

5.3 Detailed Results

5.3.1 Segment Duration and Round-Trip Time

Figure 5 shows the impact of an increasing segment duration for an RTT of 30 ms and 300 ms. Figure 5a shows the average server-to-display delay, defined as the time between the release of a segment and its playout at client-side. Both for an RTT of 30 ms and 300 ms, a clear increase is observed for higher segment durations. This is because the selected buffers are designed to hold five video segments, and the playout delay is directly proportional to the buffer size. While this increase is observed both for HTTP/1.1 and HTTP/2, results for the latter are slightly better. This is because, using a push-based approach, an average decrease of half an RTT cycle is achieved. Using a segment duration of 133 ms, the initial server-to-display delay is reduced to 0.87 s and 0.77 s for HTTP/1.1 and HTTP/2 respectively. A naive and straightforward way to lower the playout delay is thus to simply use super-short segments. However, as shown by Figure 5b, the average video bit rate is significantly lower when short segments of 133 ms and 266 ms are used. This is especially true for an RTT of 300 ms, where the lowest video bit rate is always selected. This is because it is impossible to get a segment in time, as its duration is even lower than the RTT. For every requested segment, a playout freeze is thus observed. For larger segment durations, the average bit rate increases because of a higher bandwidth utilization. Using HTTP/2's server push however, the average video bit rate is always around 20 Mb/s. This indicates that the proposed approach results in a higher bandwidth utilization and selected quality level, which is attributed to the gain of an RTT cycle for every segment request.

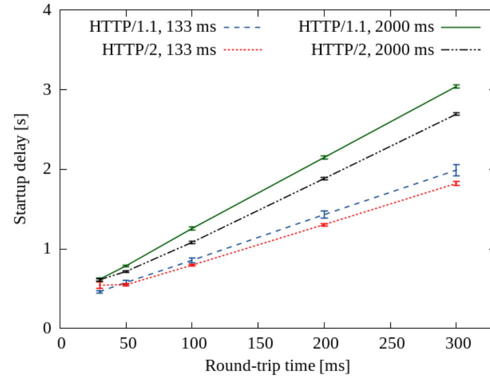


Figure 6: Impact of the RTT on the average startup delay, both for a segment duration of 133 ms and 2 s.

Figure 6 shows the impact of an increasing RTT on the client's startup delay, for a segment duration of 133 ms and 2 s. While differences are small for an RTT of 30 ms, a clear increase is shown for higher RTTs. This is the consequence of the TCP slow-start phase, which requires multiple RTT cycles to send the manifest and the first segment from server to client. For both segment durations, results for HTTP/1.1 are clearly inferior to those for HTTP/2. The average gain is one RTT cycle, attributed to the fact that no additional requests are required to get the first video segments. From these results, we can conclude that using HTTP/2's server push is indeed beneficial when the manifest and video segments are stored on the same server. Using HTTP/2 with a segment duration of 133 ms, instead of HTTP/1.1 with a duration of 2 s, the average startup delay for an RTT of 300 ms can actively be reduced from 3.04 s to 1.82 s. It is worth noting that, if the user would switch from one videostream to another, the channel change delay would be reduced by exactly one RTT cycle if the existing TCP connection is reused.

A summary of results for HTTP/1.1 and HTTP/2 with an RTT of 300 ms is presented in Table 2, for a segment duration of 133 ms and 2 s. These results indicate that using super-short segments to limit the server-to-display delay, is not feasible when HTTP/1.1 is used in a scenario with high RTTs: when the RTT exceeds the segment duration, the lowest bit rate is always selected. Using the push-based ap-

Configuration	Segment duration [ms]	Bit rate [Mb/s]	Server-to-display delay [s]	Startup delay [s]
HTTP/1.1	133	1.00 ± 0.00	1.07 ± 0.07	1.99 ± 0.07
	2000	19.62 ± 0.00	10.59 ± 0.25	3.04 ± 0.02
HTTP/2	133	19.94 ± 0.00	1.05 ± 0.02	1.82 ± 0.03
	2000	19.61 ± 0.00	10.48 ± 0.25	2.69 ± 0.01

Table 2: Performance summary for HTTP/1.1 and HTTP/2, for an RTT of 300 ms. The average values are reported, together with the 95% confidence intervals.

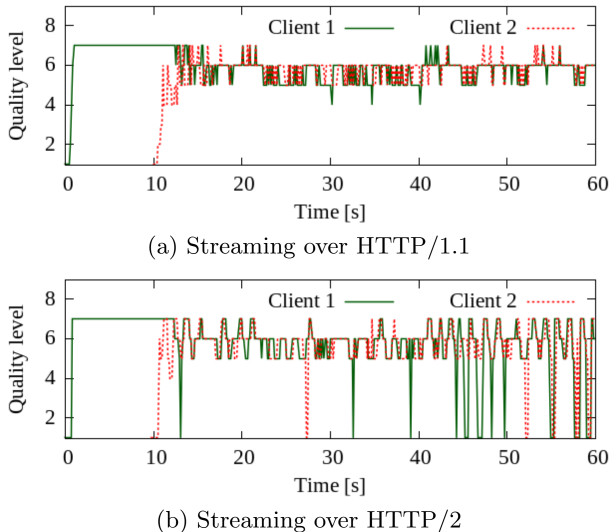


Figure 7: Played quality level for two clients streaming video with a segment duration of 133 ms and an RTT of 30 ms, both for HTTP/1.1 and HTTP/2.

proach however, segment durations as low as 133 ms can be used with an RTT of up to 300 ms. In contrast to streaming over HTTP/1.1, the video content is delivered at the highest quality when the perceived bandwidth is sufficiently high. Comparing the server-to-display delay for HTTP/1.1 with a segment duration of 2 s and HTTP/2 with a segment duration of 133 ms, a reduction of 90.1% is observed. This means that the client is capable of following the live signal more closely, resulting in a better user experience. Furthermore, a reduction of 40.1% is achieved for the start-up delay, indicating that the client can start the playout of the video significantly faster. Overall, results for the traditional HTTP/1.1 approach are inferior to those for HTTP/2, showing the potential of the proposed push-based approach.

6. FUTURE WORK

Future work will focus on (1) a more elaborate evaluation of full push and super-short segments using time-varying bandwidth bottlenecks, competing clients, and the combination of HAS and non-HAS traffic, (2) the application of the quality-level dependent encoding technique, and (3) an evaluation of the other proposed methods.

To demonstrate possible challenges with competing clients, we performed a simple experiment where we expanded the topology in Figure 4 with one additional client. In Figure 7a, the two competing clients use traditional HTTP/1.1. When the first client starts, the RDA almost immediately increases the selected quality to 20 Mb/s. Once the second client starts, the first client lowers its video quality and the clients further compete for the available bandwidth (25 Mb/s). In Figure 7b, both clients use the proposed full-push approach over HTTP/2. Again, the RDA immediately increases the selected quality to 20 Mb/s. However, once the second client starts streaming, upward peaks are wider and short peaks to the lowest quality level are observed. This is explained as follows. While a segment n is being received by the client, new segments $n+1, ..., n+k$ are already pushed to the client at the same video quality as segment n. Because under the influence of the competing clients, the network is not always capable to provide the necessary bandwidth, the download of the segments might take longer. As a result, the RDA's panic threshold for the buffer filling is hit more frequently, leading to the selection of the lowest video quality. More work is required to evaluate possible solutions such as the use of a more frequent update signal from client to server or a limitation of the number of segments in flight between server and client.

7. CONCLUSIONS

In this paper, we proposed ten HTTP/2 based methods to improve the QoE for HAS streaming. Furthermore, we presented a new streaming concept for live streaming that combines the advantages of push-based real-time streaming and the advantages of HTTP adaptive streaming. This concept is based on the combination of super-short segments and our HTTP/2-based full-push method. To quantify the advantages, we implemented a working prototype based on the MiniNet framework. We used a modified Jetty server that is HTTP/2-push-enabled and a modified libdash client. To perform an initial evaluation of both methods, we used a constant network bandwidth sufficient to support the highest quality levels in the HAS setup.

First, we showed that high RTT values significantly reduce the average quality of the HAS sessions, especially when segment durations are short. Next, we showed that with an RTT of 300 ms, super-short segments can reduce the server-to-display delay for live streaming with 89.9 %. However, it was also demonstrated that the use of super-short segments causes an unacceptable degradation of the average HAS quality. Finally, we evaluated the combination of super-short segments with the full-push method. We demonstrated that HTTP/2-based full push is able to eliminate the disadvantages inherent to the use of super-short segment durations such as the HTTP message overhead and the reduced average quality. Moreover, the combination of the two methods was able to reduce the server-to-display delay for live streams by 90.1% and the start-up delay by 40.1%.

8. ACKNOWLEDGMENTS

This research was performed partially within the iMinds V-FORCE (Video: 4K Composition and Efficient stream-

ing) project under an Agency for Innovation by Science and Technology in Flanders (IWT) grant agreement no. 130655 and by FLAMINGO, a Network of Excellence project (318488) supported by the European Commission under its Seventh Framework Programme. Jeroen van der Hooft is funded by grant of the IWT."

9. REFERENCES

[1] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen. Server-Based Traffic Shaping for Stabilizing Oscillating Adaptive Streaming Players. In *NOSSDAV*, 2013.

[2] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2. Technical Report Internet-Draft, 2015.

[3] S. Benno, A. Beck, J. Esteban, L. Wu, and R. Miller. WiLo: A Rate Determination Algorithm for HAS Video in Wireless Networks and Low-Delay Applications. In *Globecom Workshops*, 2013.

[4] bitmovin. libdash. https://github.com/bitmovin/libdash/.

[5] G. Bjontegaard. Calculation of Average PSNR Differences between RD-Curves. In *ITU-T SG16/Q.6, Doc. VCEG-M033*, 2001.

[6] N. Bouten, M. Claeys, S. Latré, J. Famaey, W. Van Leekwijck, and F. De Turck. Deadline-Based Approach for Improving Delivery of SVC-Based HTTP Adaptive Streaming Content. In *NOMS*, 2014.

[7] N. Bouten, J. Famaey, S. Latré, R. Huysegems, B. Vleeschauwer, W. Leekwijck, and F. De Turck. QoE Optimization Through In-Network Quality Adaptation for HTTP Adaptive Streaming. In *CNSM*, 2012.

[8] A. Cardaci, L. Caviglione, A. Gotta, and N. Tonellotto. Performance Evaluation of SPDY over High Latency Satellite Channels. Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. 2013.

[9] M. Claeys, S. Latré, J. Famaey, T. Wu, W. Van Leekwijck, and F. De Turck. Design of a Q-Learning-Based Client Quality Selection Algorithm for HTTP Adaptive Video Streaming. In *AAMAS*, 2013.

[10] Conviva. Viewer Experience Report 2014. 2014.

[11] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang. Understanding the Impact of Video Quality on User Engagement. *SIGCOMM*, 2011.

[12] Y. Elkhatib, G. Tyson, and M. Welzl. Can SPDY Really Make the Web Faster? In *IFIP*, 2014.

[13] J. Erman, V. Gopalakrishnan, R. Jana, and K. K. Ramakrishnan. Towards a SPDY'Ier Mobile Web? In *CoNEXT*, 2013.

[14] J. Famaey, S. Latré, N. Bouten, W. Van de Meerssche, B. De Vleeschauwer, W. Van Leekwijck, and F. De Turck. On the Merits of SVC-based HTTP Adaptive Streaming. In *IFIP*, 2013.

[15] FCC. Measuring Broadband America. http://transition.fcc.gov/cgb/measuringbroadbandreport/2013/Measuring-Broadband-America-feb-2013.pdf, 2013.

[16] Google. SPDY: An Experimental Protocol for a Faster Web. Technical report, 2009.

[17] R. Huysegems, B. De Vleeschauwer, K. De Schepper, C. Hawinkel, T. Wu, K. Laevens, and W. Van Leekwijck. Session Reconstruction for HTTP Adaptive Streaming: Laying the Foundation for Network-Based QoE Monitoring. In *IWQoS*, 2012.

[18] IETF. Hypertext Transfer Protocol (httpbis). 2012.

[19] Igvita. *High Performance Browser Networking*. O'Reilly, 2014.

[20] iMinds. Virtual Wall. http://ilabt.iminds.be/iminds-virtualwall-overview/.

[21] T. Lohmar, T. Einarsson, P. Frojdh, F. Gabin, and M. Kampmann. Dynamic Adaptive HTTP Streaming of Live Content. In *WoWMoM*, 2011.

[22] R. K. P. Mok, X. Luo, E. W. W. Chan, and R. K. C. Chang. QDASH: A QoE-aware DASH System. In *MMsys*, 2012.

[23] C. Mueller, S. Lederer, C. Timmerer, and H. Hellwagner. Dynamic Adaptive Streaming over HTTP/2.0. In *ICME*, 2013.

[24] C. Müller, S. Lederer, and C. Timmerer. An Evaluation of Dynamic Adaptive Streaming over HTTP in Vehicular Environments. In *MoVid*, 2012.

[25] C. Müller, D. Renzi, S. Lederer, S. Battista, and C. Timmerer. Using Scalable Video Coding for Dynamic Adaptive Streaming over HTTP in Mobile Environments. In *EUSIPCO*, 2012.

[26] S. Petrangeli, M. Claeys, S. Latré, J. Famaey, and F. De Turck. A Multi-Agent Q-Learning-Based Framework for Achieving Fairness in HTTP Adaptive Streaming. In *NOMS*, 2014.

[27] S. Petrangeli, J. Famaey, M. Claeys, S. Latré, and F. De Turck. QoE-driven Rate Adaptation Heuristic for Fair Adaptive Video Streaming. *ACM TOMM*, 2015.

[28] Y. Sánchez de la Fuente, T. Schierl, C. Hellge, T. Wiegand, D. Hong, D. De Vleeschauwer, W. Van Leekwijck, and Y. Le Louédec. iDASH: Improved Dynamic Adaptive Streaming over HTTP Using Scalable Video Coding. In *MMSys*, 2011.

[29] Y. Sánchez de la Fuente, T. Schierl, C. Hellge, T. Wiegand, D. Hong, D. De Vleeschauwer, W. Van Leekwijck, and Y. Le Louédec. Efficient HTTP-Based Streaming Using Scalable Video Coding. In *MMSys*, 2012.

[30] Sandvine. Global Internet Phenomena Report, 1H 2014. 2014.

[31] T. Stockhammer. Dynamic Adaptive Streaming over HTTP: Standards and Design Principles. In *ACM MM*, 2011.

[32] S. Tavakoli, J. Gutierrez, and N. Garcia. Subjective Quality Study of Adaptive Streaming of Monoscopic and Stereoscopic Video. *Journal of Selected Areas in Communications*, 2014.

[33] M. Team. MiniNet. http://mininet.org/.

[34] T. Tsujikawa. nghttp2. https://nghttp2.org/.

[35] Webtide. Jetty. https://webtide.com/.

[36] S. Wei and V. Swaminathan. Cost Effective Video Streaming Using Server Push over HTTP 2.0. In *MMSP*, 2014.

[37] S. Wei and V. Swaminathan. Low Latency Live Video Streaming over HTTP 2.0. In *NOSSDAV*, 2014.