Video Streaming over HTTP/2: Design and Evaluation of Adaptive Server-Paced Push

Huei-Wen Ferng, Shan-Hsiang Shen, and Chih-Wei Lai

Abstract: **By using the server push of the hypertext transfer protocol (HTTP) version 2.0, i.e., HTTP/2, along with the technique of server pacing, a novel scheme is proposed in this paper to deliver video segments. Furthermore, the load of bitrate adaptation is shifted to the server to avoid bandwidth competition and wastage caused by bitrate switching. It can be explicitly shown that no significant overhead is brought by our proposed scheme via complexity analysis. Moreover, our proposed scheme generates one HTTP request only. With extensive simulations, we successfully demonstrate that it is superior over the closely related schemes in terms of the average achievable bitrate, the number of buffer stalls, the ratio of unclaimed pushes, etc., in particular, when a harsh network condition arises.**

Index Terms: **Adaptive, Bitrate adaptation, HTTP/2, server pacing, server push, video streaming.**

I. INTRODUCTION

THE HTTP adaptive streaming (HAS) is on top of the HTTP protocol and has been widely deployed. Because the HTTP traffic can easily penetrate the network address translation (NAT) and firewall, HAS has become one of the most widely used video streaming technologies [1]. The well-known instances of HAS include the HTTP live streaming provided by Apple, the HTTP dynamic streaming provided by Adobe, and the smooth streaming provided by Microsoft. Although these technologies are similar, they are not compatible. Up to now, the dynamic adaptive streaming over HTTP (DASH) [2] proposed by the motion picture expert group (MPEG) is the only international standard for HAS.

HAS pre-encodes a video clip into multiple versions with different levels of quality and cuts the video into segments. A client selects and downloads a suitable version for each video segment based on the network condition such as available bandwidth to maximize the video quality. Because HAS is a pull-based scheme [3], a client needs to send an HTTP request for each video segment. It is crystal clear that longer segment periods reduce the total number of video segments to be downloaded, resulting in fewer HTTP requests accordingly. However, longer segment periods cause unstable video buffering and

H.-W. Ferng, S.-H. Shen, and C.-W. Lai are with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, email: {hwferng, sshen}@csie.ntust.edu.tw, rusty0831@hotmail.com.

H.-W. Ferng is the corresponding author.

higher latency inevitably [4], [5] because of slow reaction to the changes of the network condition. Therefore, most of the Internet video streaming schemes prefer a shorter video segment period. On the contrary, frequent HTTP requests bring higher power consumption to mobile devices [6], [7], lower link utilization, higher round-trip time (RTT), and more requests to be processed by network nodes [8], [9]. To address these issues, the international organization for standardization (ISO) and the international electrotechnical commission (IEC) are working on the extension of DASH. The number of video segments to be downloaded will be explicitly specified in an HTTP request to reduce the total number of HTTP requests [10].

Moreover, HTTP/2 [11] standardized in 2015 has been the most important update for the communication between servers and browsers since the time instant when HTTP/1.1 was released in 1999. HTTP/2 supports a push scheme at the server-side. Based on this scheme, the server estimates the content that the client may be interested in when it receives a request from the client and pre-sends the content to the client to reduce the download latency even though more network bandwidth may be occupied. However, the server push scheme in the current version of HTTP/2 only defines the format of transmitted content and control messages between the communication peers. Therefore, no further specifications are posed regarding the way to deploy servers and clients. That is the reason why many previously proposed schemes apply the server push scheme to HAS for reducing the HTTP request overhead so that power consumption of mobile devices can be lowered [12] and the playback latency of the live video streaming can be shortened [5].

In the literature, the closely related schemes were proposed in [8], [9]. These schemes tried to reduce the HTTP request overhead by using the server push scheme but they still encountered some problems. In particular, the traffic with the new data rate competes for the network bandwidth with the traffic with the old data rate when an adaptive video stream at a client-side switches the video data rate in a harsh environment. Such competition causes data rate switching delay and wastage of network bandwidth, resulting in lower video quality. To properly address the aforementioned issues, we shall propose a novel push scheme in which the video adaptation scheme is moved to the video server in this paper. For estimating the state and the video buffer level of a client, some corresponding algorithms are designed accordingly so that a suitable video version can be selected for each video segment. With negligible extra overheads, the video server can handle the streaming process. Therefore, our proposed scheme can successfully cut the number of HTTP requests to one and resolve the bandwidth competition issue. Compared to the closely related server push schemes under extensive simulations, our proposed scheme can effectively keep the number of

HTTP requests at 1, achieve a higher average video bitrate, and bring no unclaimed pushes in a high RTT network environment with mobile hosts as compared to the closely related schemes. Undoubtedly, the quality of experience (QoE) for video streaming can be greatly enhanced by our proposed scheme.

The rest of this paper is organized as follows. The literature review on the related work is given in Section II. As for our proposed scheme, it is depicted in Section III. Section IV offers the time and space complexity of our proposed scheme with comparison to the closely related schemes. To show the effectiveness of our proposed scheme, performance evaluation is provided in Section V with comparison to the closely related schemes. Finally, Section VI concludes this paper.

II. RELATED WORK

Generally speaking, video streaming can be categorized into on-demand streaming and live streaming [2]. The content of on-demand streaming, e.g., movies, is pre-recorded so that all the video segments are available before streaming. On the other hand, the content of live streaming, e.g., a live soccer game, is made dynamically and may rely on HAS naturally. To address the corresponding issues of video streaming, in particular, HAS, two push schemes were proposed in [5], i.e., Live All-Push and K-Push [8]. In [13], Samain *et al.* gave a systematic comparison between the information-centric networking (ICN) and TCP/IP regarding the dynamic adaptive video streaming.

In the literature, [14] is one of the pioneering work applying HTTP/2 to DASH. In [12], Wei *et al.* applied the server push scheme to deliver multiple video segments to change the HTTP request/response period for matching the period of the radio resource control (RRC). Their experiments revealed that 17.9% of reduction in power consumption for mobile devices can be reached when delivering 30 video segments during each push. Huysegems *et al.* [15] provided several ideas to increase the QoE for HAS by utilizing some HTTP/2 features. Their main contribution lies in the overhead mitigation of HTTP requests as well as the reduction in the service-to-display delay and the start-up latency. With the aid of a forgetting factor, Aguayo *et al.* [16] designed a DASH adaption algorithm named adaptive forgetting factor (AFF) which has better behavior in video stalling and the number of bitrate switches.

To reduce the content redundancy and display latency of HAS, scalable video coding was leveraged by [17]. However, several HTTP requests are needed for each video segment. The aggregated RTT then causes high latency obviously, degrading the quality of service (QoS). To address this issue, Hooft *et al.* [18] proposed a server push scheme. When a client requests an enhancement layer, the corresponding base layer is pushed to the client as well to shorten the latency and to improve the bandwidth utilization.

Note that players at clients stay in the mode of buffering initially and start to play the video after the content at the buffer has exceeded a preset threshold. Obviously, a high aggregated RTT to download multiple video segments causes a critical start-up latency when the RTT gets larger. To address this issue, some papers in the literature proposed the corresponding push schemes. In [19], the server starts to deliver video segments to a client when the media presentation description (MPD) file has been downloaded by the client. Cherif *et al.* [19] showed that their scheme can reduce 50% or so of the start-up latency. Bouzakaria *et al.* [20] proposed a push scheme so that the video server pushes all initialization segments to a client when the MPD file has been requested. Such a scheme has a shorter start-up latency than the referenced scheme in [20].

For the remaining part of this section, some closely related schemes applying server push to reduce the HTTP request overhead are further examined, including Live All-Push, K-Push, and Dynamic K-Push [9]. For Live All-Push, the video server keeps pushing the video segments, if any, until the end of streaming. In [5], two problems of Live All-Push were reported explicitly. The parameter K of K-Push [8] represents the number of video segments to be pushed for each request by the video server at the same data rate. Although K-Push can reduce the load of HTTP requests as reported in [5], it inevitably brings bandwidth wastage and high switching latency caused by bandwidth competition between the data stream at an old data rate and that at a new data rate when switching the video data rate. Nguyen *et al.* [9] proposed Dynamic K-Push to switch K based on the network condition and the buffer level at the client-side. To reach such a goal, a cost function was first defined in [9] in terms of the number of pushed video segments per time, the duration of a video segment, the buffer level in seconds at the client-side, and the threshold of the lowest buffer level. Based on the cost function, Dynamic K-Push then finds the K with the minimum cost before pushing video segments. According to the simulation results given in [9], Nguyen *et al.* claimed that Dynamic K-Push can reduce the number of HTTP requests, maintain the stable buffer level, and increase the average video data rate. Nevertheless, the push scheme may still suffer from the competition for network bandwidth between different data flows when changing the video data rate.

For our proposed scheme, the bitrate adaptation method is migrated from a client-side to the server-side[1] and a server-paced push scheme determines the number of video segments to be pushed and when to push them to a client based on the state of the client. Unlike K-Push [8] and Dynamic K-Push [9], which are the two schemes in the literature selected for comparison with our proposed scheme later, there is no competition for network bandwidth between data flows caused by video data rate switching. After pushing a video segment, the bitrate adaptation method calculates the throughput based on the data size, etc. to determine the video data rate for the next video segment. Explicitly, our design changes the pull-based nature of HAS to the push-based nature by employing the HTTP/2 server push scheme but switches video data rates according to the network condition as the traditional HAS. Further compared to the pull-based HAS, the push-based HAS employed by our proposed scheme avoids the RTT accumulation problem [15], [18], [21], [22], then shortening the latency efficiently. Even if the bitrate adaptation method is moved to the server, no much overhead is brought to the server and this can be manifested by the time complexity analysis to be discussed later. Besides, the stateful nature of HTTP/2 [23] will not be changed to accommodate our

---

[1]The issue of server errors or switching will not be covered by this paper and deserves part of the future work.

proposed scheme. These show that more advantages but fewer disadvantages are brought by our proposed scheme.

III. PROPOSED ADAPTIVE SERVER-PACED PUSH

A. Video Buffering

Measuring in time for the buffered video is a more direct way than measuring in packets to reflect the buffering condition as employed by [24], [25]. Therefore, the playback time in seconds was employed by [9], [24], [26]. Using such a way, the output rate is always 1-second playback time per second no matter what the video data rate is. Assume that $T(i)$ represents the average throughput when transmitting the ith video segment, $br(i)$ stands for the video data rate of the ith video segment, and t is the transmission time of the ith video segment. When the ith video segment enters the buffer, its video buffer level increases by $T(i) \times t/br(i)$. The average input rate to the video buffer is then $(T(i) \times t)/(br(i) \times t) = T(i)/br(i)$. Therefore, the real-time virtual client buffer proposed by this paper adopts the aforementioned feature to estimate/mimic the output rate during streaming without caring for the current video data rate. As for the details of the real-time virtual client buffer, it is depicted in the next paragraph.

B. Real-Time Virtual Client Buffer

First of all, the general behavior of a video player is given as follows [27]: At the beginning, the video playback stays in the state of buffering until the video buffer level surpasses a predefined threshold ($buf_{\min}$) before video playback. During the video playback, the video player keeps the video buffer level closer to a target video buffer level denoted by buf. Except for these two thresholds used for the virtual client buffer, $buf_{\max}$ in the maximal video playback time can be further defined to denote the buffer capacity.

In our architecture employed, the video server cannot directly have the state and the buffer level at a client-side. Instead, the video server mimics the behavior of the video player to trace its state and buffer level. Although [26]–[28] touched the buffer level estimation, the virtual client buffer was not addressed. Moreover, our proposed push scheme will determine how to push video segments to a client based on the state of the client. For these reasons, a new virtual buffering scheme called the real-time virtual client buffer (RVCB) is proposed in this paper. For each new connection, the video server enables an RVCB for that client by running an independent thread governed by a finite state machine (FSM) based on the information of the buffer level and current state. Of course, the video server calculates and updates the video buffer level based on the size of the video segment after a client receives a video segment. This allows the video server to determine its video push policy based on the buffer level, etc. Note that two states are associated with the output rate for our proposed virtual client buffer: a buffering state with the output rate of 0 and a playing state with the output rate of 1. With the help of the two states, an FSM can be designed to mimic the state of the video player at a client-side. The *Buffering* state then serves as the initial state for each connection. The FSM will be switched to this state if the buffer is out of content. When staying in this state, the output

rate of 0 is associated and the video server refills the buffer until the buffer level reaches the threshold of $buf_{\min}$. Once $buf_{\min}$ is exceeded, the FSM is then switched to the *Playing* state at the output rate of 1 until the end of playback. Of course, the FSM will be switched to the *Buffering* state from the *Playing* state if the buffer is out of content.

C. Server-Paced Push Policy

For our proposed push policy, it allows the video streaming server to determine the way to push videos based on the state of a client. When a client requests an MPD file, the video streaming server then allocates an RVCB for the client and sets the state of the FSM to the *Buffering* state. Then, the video streaming server keeps sending video segments back to back until the buffer level reaches $buf_{\min}$. After that, the FSM is switched to the *Playing* state. Instead of pushing video segments back to back, the video streaming server determines the volume of video segments to be pushed according to the gap between the current buffer level at the client-side and the target buffer level (buf). Besides, a bitrate adaptation method takes the turn after finishing pushing video segments to decide a suitable bitrate for the next video segment based on the currently available throughput derived from the volume of pushed data and the corresponding time taken.

D. Challenges and Solution to Push Videos over HTTP/2

There exist some challenges to applying the server push in HTTP/2 directly to HAS. When receiving a request by the server and having video segments to push, the video server will issue a PUSH_PROMISE frame for each data to be pushed to let the client know the PUSH information, including the file name, the promised stream ID, etc. Via the promised stream ID, the client can know which stream will carry the data. Once the PUSH_PROMISE frame is sent, the server can respond and send data through the promised stream. There are some specifications and limitations for the server push over HTTP/2. First, the PUSH_PROMISE frame should be pushed via the client-initiated data stream, implying that the video server can only push data to a client via responding to an HTTP request. Second, the PUSH_PROMISE frame must be sent before any frames for pushing data to avoid a race condition.

Although Live All-Push [5] issues one HTTP request by the client and pushes all video segments to the client by the video server, the server cannot send the PUSH_PROMISE frame to push video segments in Live All-Push because the only data stream initialized by the client for the MPD file is closed right after the MPD file is received. Live All-Push addressed this issue by sending push tags to trigger the push process at the server-side. Actually, these push tags are similar to the HTTP GET requests, revealing that the number of requests cannot be lowered at all.

To achieve the original goal of the server push in HTTP/2, i.e., reducing web page loading time, it can be done by parsing the hyperlinks to know all the associated objects first. Then, encapsulate these objects into DATA frames and send these frames. For HAS, the aforementioned way should be properly taken care of because the video versions are selected dynamically and adaptively during playback. Obviously, the video server can-
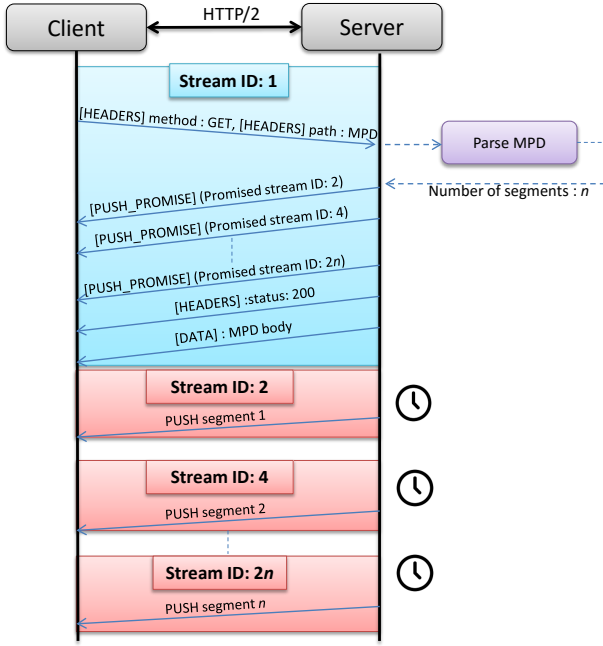
Fig. 1. Detailed signal flows of the proposed server push over HTTP/2.

not determine which video segments should be pushed before responding to the MPD file. Towards this goal, a solution is proposed in the following and illustrated in Fig. 1. Although the server cannot know all the video segments to be pushed before responding to the MPD file, it can reserve data streams for video segments by pre-sending multiple PUSH_PROMISE frames to the client. The detailed procedure is depicted as follows. The video server parses the MPD file when receiving a request for the MPD file and counts the number of video segments to be pushed. Then, the video server sends all the PUSH_PROMISE frames via the only data stream (i.e., Stream ID of 1 shown in Fig. 1) opened by a client for downloading the MPD file before the closure of that data stream. At this moment, the file names corresponding to the video segments to be pushed cannot be made sure. Therefore, the path (i.e., **:path**) in the pseudo-header of the PUSH_PROMISE frames is virtually set to **Dummy**. Such a setting will not cause any problem because PUSH_PROMISE frames are not in charge of transmitting data to be pushed. This should be enforced that the protocol stack of HTTP/2 will not try to encapsulate the *virtual* **Dummy** file, which is found from **:path** of the pseudo-header, into DATA frames and send these frames before our server-paced push scheme takes over the video streaming process. After all the PUSH_PROMISE frames (for example, n frames shown in Fig. 1) are sent, the status code (200) in the header and the MPD file (MPD body) via DATA frames follow. Of course, extra data steams with promised stream IDs of $2, 4, \cdots, 2n$ in the state of *reserved (local)* will be created and reserved to push the subsequent video segments to be pushed later. Note that the flag of *End Stream* will be set for the last DATA frame to close the data stream. Of course, the data stream with promised stream ID 1 enters the *closed* state and releases resources if the video server has received a frame with the flag of *End Stream* set by the client already. At this moment, the only data stream initialized by the client is closed

and our bitrate adaptation method then takes over to adapt video bitrates accordingly as shown in Fig. 1 via the data streams previously reserved by the PUSH_PROMISE frames. The real file names, sizes, etc. are reflected by the newly added fields of *url* and *file_size* in the headers to let the client know the corresponding information.

E. Design of Bitrate Adaptation

As illustrated in [4], [27], a bitrate adaptation method usually includes two modules: throughput estimation and bitrate selection. In the following paragraphs, let us detail our bitrate adaptation method.

E.1 Proposed Bitrate Adaptation Method

Our proposed bitrate adaptation method can be shown via the compound of Algorithms 1–4. In the following, let us discuss these algorithms in detail.

Algorithm 1 Per-session initialization

1:　$idx = 1, br = NULL$ //Global Variable Initialization
2:　$b = 0, s = BUFFERING$ //Global Variable Initialization
3:　$T_s = NULL$ //Global Variable Initialization
4:　Parse the requested MPD file to get the total number of segments n, representations within this adaptation set $R[]$, number of representations m, and segment duration τ.
5:　**for** $i = 1$ to n **do**
6:　　　Send a PUSH_PROMISE frame with promised Stream ID $2i$.
7:　**end for**
8:　Respond to the HTTP GET request with the MPD payload and HTTP status code (200).
9:　Call Algorithm 2 for initial buffering.

Connection initialization: As shown in Algorithm 1, an initialization procedure should be performed by the video server upon receiving the request for an MPD file by a client. Firstly, five global variables are initialized. Here idx means the index of the next video segment to be pushed, br is the bitrate of the next video segment to be pushed, b represents the level of the virtual client buffer, s denotes the state of the virtual client buffer, and T_s stands for the smoothed throughput measured according to the pushed video segments. These global variables store the states of playback, which will be referred by the corresponding algorithms. Secondly, the algorithm retrieves the number of video segments n, the adaptation set of bitrates for all supported video versions $R[]$, the number of representations m, and the segment duration τ by parsing the requested MPD file. Note that m is the number of elements, i.e., representations, in the set of $R[]$ and $R[0] < R[1] < \cdots < R[m-1]$. For the reason explained previously, the video server needs to send a PUSH_PROMISE frame for each video segment. Thirdly, the video server calls Algorithm 2 for the initialization of buffering after responding to the HTTP GET request with the MPD payload and status code.

When performing initialization of buffering, Algorithm 2 calculates the number of video segments n_{buf} to be pushed via the number of segments to fill up the client buffer with the mini-

Algorithm 2 Initialization of buffering

1: $n_{buf} \leftarrow \lceil buf_{\min}/\tau \rceil$
2: $br \leftarrow R[0]$ //Start from the lowest bitrate.
3: **for** $i = 1$ to n_{buf} **do**
4:     Push segment i of bitrate br and measure the throughput (T).
5:     Call Algorithm 6 to select the bitrate of the next segment. //Algorithm 6 will update the bitrate of the next selected segment to br.
6:     $b \leftarrow b + \tau$ //Update the virtual buffer level.
7: **end for**
8: $idx \leftarrow idx + n_{buf}$ //Update the segment index.
9: $s \leftarrow PLAYING$ //Switch to the $PLAYING$ state.

---

mum threshold, i.e., $\lceil buf_{\min}/\tau \rceil$, and starts from the lowest bitrate. When finishing pushing a segment, the algorithm selects a bitrate for the next video segment according to the estimated throughput by calling Algorithm 6 which will be discussed later. Then, the virtual buffer level is updated. Such a procedure is repeated for n_{buf} times. Once the video buffer at the client-side reaches $buf_{\min}$, the segment index is updated and the FSM is then switched to the *Playing* state. When staying in the *Playing* state, Algorithm 7 imitates the output rate of the video buffer level at the client-side officially with a period of c in seconds. If the video buffer is exhausted, then Algorithm 2 stops and the FSM is switched to the *Buffering* state until the video buffer level reaches $buf_{\min}$ again.

Pushing video segments and estimation of the video buffer level at the client-side: In Algorithm 1, the total number of segments to be pushed, i.e., n, has been acquired through parsing the MPD file. Therefore, the remaining goal is pushing these segments sequentially and properly. Once all the segments have been pushed, the entire session of video streaming should be terminated as shown in lines 1–3 of Algorithm 3. When staying in the *Playing* state, the video server calculates the number of video segments to be pushed, i.e., n_{buf}, based on the difference between the current video buffer level, i.e., b, and the target video buffer level, i.e., buf, to let the video buffer level be close to buf after pushing as follows:

$$n_{buf} = \begin{cases} \lceil \frac{buf-b}{\tau} \rceil, & \text{if } 0 < b < buf, \\ 0, & \text{if } b \geq buf. \end{cases}$$

Note that no more segments should be pushed if $b \geq buf$. The aforementioned procedure is shown in Algorithm 3. If $n_{buf} > 0$, Algorithm 3 will call Algorithm 4 to push video segments. Whenever a video segment is pushed, the virtual client buffer level will be updated according to Algorithm 4 as well. Given that the ith video segment with size τ in seconds and bitrate $br(i)$ is received by the client, the change of the virtual client buffer level is estimated as follows:

$$\Delta b = \tau - \frac{br(i) \times \tau}{T(i)}.$$

Such a change is explained in the following. τ cannot be added to the virtual client buffer level directly because a time period is required to let that segment reach the client after being pushed

Algorithm 3 Calculating the number of segments to be pushed

1: **if** $idx > n$ **then**
2:     Terminate the entire session of video streaming because there is no more segments to be pushed.
3: **end if**
4: **if** $b \geq buf$ **then**
5:     Quit the algorithm. //Quit the algorithm because current buffer level has reached the target buffer level already.
6: **end if**
    $n_{buf} \leftarrow \lceil \frac{buf-b}{\tau} \rceil$ //Calculate the number of segments to be pushed.
7: **if** $idx + n_{buf} - 1 > n$ **then**
8:     $n_{buf} \leftarrow n - idx + 1$ //Ensure that non-existing segments will not be pushed.
9: **end if**
10: **for** $i = 1$ to n_{buf} **do**
11:     Call Algorithm 4 to push segments.
12: **end for**

---

by the server. This time period should be reflected by the buffer level at the client-side for sure because the buffer level at the client-side drops as time goes. Here, the transmission time, i.e., $br(i) \times \tau/T(i)$ in seconds, is employed. Therefore, the change of the virtual client buffer level should be $\tau - br(i) \times \tau/T(i)$. If the bitrate of the ith video segment ($br(i)$) is higher than $T(i)$, it results in $\Delta b < 0$, implying that the exhaustion of the video buffer level is faster than refilling. In Algorithm 4, Algorithm 5 will be called to calculate the new smoothed throughput and Algorithm 6 will be called to select the bitrate of the next segment. These will be detailed in the following two paragraphs.

Algorithm 4 Update of the virtual buffer level and segment push at the $PLAYING$ state

1: Push segment idx of bitrate $br(idx)$, save segment size in bytes to $size(idx)$ and elapsed time in seconds to t.
2: $T(idx) \leftarrow \frac{size(idx) \times 8}{t}$ //Measure throughput ($T(idx)$).
3: $b \leftarrow b + \tau - \frac{br(idx) \times \tau}{T(idx)}$ //Update the virtual buffer level.
4: Call Algorithm 5 to calculate the new smoothed throughput.
5: Call Algorithm 6 to select the bitrate of the next segment.
6: $idx \leftarrow idx + 1$ //Update the segment index.

Throughput estimation: Due to the fact that the adaptation method is moved to the video server in our architecture, the overhead of our method is critical for scalability. To this end, the throughput-based method is adopted in our design for lowering time and space complexity. As far as the throughput-based method is concerned, it can be further divided into the following categories: instant throughput based (ITB), smoothed throughput based (STB), and conservative throughput based (CTB) methods [27]. To maximize the average bitrate for our method, STB is leveraged. In our paper, the smoothed throughput is calculated via the following equation:

$$T_s(i) = \begin{cases} (1 - \rho) \times T_s(i-1) + \rho \times T(i), & \text{if } i > 1, \\ T(i), & \text{if } i = 1, \end{cases}$$

where $T_s(i)$ means the smoothed throughput after the ith video

segment is transmitted, $(1 - \rho)$ and ρ are the weighting factors, and $T(i)$ is the throughput when the ith video segment is transmitted. Our bitrate adaptation method then selects a bitrate for the $(i + 1)$th video segment according to $T_s(i)$ with the algorithm shown in Algorithm 5.

Algorithm 5 Calculation of smoothed throughput

Require: T : The latest throughput measured for pushing a segment
1: **if** $T_s = 0$ **then**
2:      $T_s \leftarrow T$ //The initial situation
3: **else**
4:      $T_s \leftarrow (1 - \rho) \times T_s + \rho \times T$
5: **end if**

Video bitrate selection: Note that the bitrate should be close to the estimated throughput ideally. However, it is hard to estimate future throughput exactly. For this reason, we leverage a more conservative way to estimate the throughput. First of all, we calculate T_{safe} by multiplying $T_s(i)$ by a safety margin denoted by α ($0 < \alpha < 1$) as follows:

$$T_{safe} = (1 - \alpha) \times T_s(i).$$

The bitrate with the highest quality but lower than T_{safe} is selected from $R[]$. If all candidate bitrates are higher than T_{safe}, the bitrate with the lowest quality is then set. Such a procedure is shown in Algorithm 6.

Algorithm 6 Bitrate selection

Require: α : Safety margin
1: $T_{safe} \leftarrow (1 - \alpha) \times T_s$
2: **for** $i = m - 1$ to 0 **do**
3:      **if** $R[i] < T_{safe}$ **then**
4:          $br \leftarrow R[i]$ //Set the selected bitrate to br.
5:          Quit the algorithm.
6:      **end if**
7: **end for**
8: $br \leftarrow R[0]$ //Choose the lowest bitrate, i.e., $R[0]$, if no match is found.

Algorithm 7 Real-time virtual client buffer: Emulation of the client buffer level reduction

Require: c : Cycle duration to call this algorithm.
1: **if** $b \leq 0$ **then**
2:      $s \leftarrow BUFFERING$ //Switch to the $BUFFERING$ state.
3:      Quit the algorithm.
4: **else**
5:      $b \leftarrow b - c$ //Reduce the virtual buffer level.
6: **end if**

To show the relationship among Algorithms 1–7, the following remark is given.

Remark 1: Note that Algorithms 1–3 will be performed sequentially for a session of video streaming in our proposed scheme. About Algorithm 4, which calls Algorithms 5 and 6,

Table 1. Symbols associated with complexity.

Symbol	Description
m	number of representations (bitrate versions) within the adaptation set
T_{add}	execution time of addition
T_{sub}	execution time of subtraction
T_{mul}	execution time of multiplication
T_{div}	execution time of division
T_{assign}	execution time of variable assignment
$T_{compare}$	execution time of comparing two variables
I_{loop}	loop iteration overhead
$T_{subscript}$	execution time of array subscripting
C_{call}	function call overhead
C	one or more basic addressable units of memory (byte) to store an integer
C_{http2}	HTTP/2 per-connection memory overhead to keep states
C_{mem}	memory footprint of the proposed adaptation logic at the server

it is called by Algorithm 3. Actually, Algorithm 6 is called by Algorithm 2 as well. As for Algorithm 7, it is controlled by another thread to emulate the reduction of the virtual client buffer level.

IV. COMPLEXITY OF THE BITRATE ADAPTATION METHOD

Our bitrate adaptation method is involved with Algorithms 1–7, where Algorithm 4 activated by Algorithm 3 and the algorithms called by it, i.e., Algorithms 5 and 6, keep running during video streaming, while Algorithms 1 and 2 are called for initialization once during the whole video streaming. As for Algorithm 7, it simply handles the the emulation of the client buffer level reduction. Therefore, we shall focus on the time and space complexities of the algorithms running repeatedly during video streaming.

A. The Execution Time

A.1 The Execution Time of Algorithm 5

During playback, only line 1 and line 4 in Algorithm 5 will be executed repeatedly since T_s has been set already. The associated total execution time is the $T_{compare} + T_{sub} + 2T_{mul} + T_{add} + T_{assign}$, where the corresponding symbols can be referred to Table 1.

A.2 The Worst-Case Execution Time of Algorithm 6

Through proper derivation, the total execution time of Algorithm 6 is then $(m + 1) \times T_{subscript} + m \times (I_{loop} + T_{compare}) + 2T_{assign} + T_{sub} + T_{mul}$.

A.3 The Execution Time of Algorithm 4

Through proper derivation, the total execution time of Algorithm 4 is $8T_{assign} + 5T_{mul} + 2T_{div} + 3T_{add} + 3T_{sub} + (m + 1)T_{compare} + 2C_{call} + (m+1)T_{subscript} + mI_{loop}$ for the worst case.

B. Time Complexity

When the video server handles a connection, the bitrate adaptation method (Algorithm 4) takes the worst-case execution time as discussed previously. It is explicitly that each time-related symbol in Table 1 has its upper bound. Letting C_{max} be the

maximum value of these upper bounds and $f(n)$ denote the execution time when the server handles n concurrent connections, we then have

$$f(n) \leq n(3m + 25)C_{\max}. \tag{1}$$

Note that $C_{\max}$ is quite small and m is usually small, implying that the extra overhead caused by moving bitrate adaptation to the video server in our architecture is negligible. Compared to our architecture, no extra overhead in time complexity is brought by the traditional HAS, K-Push, and Dynamic K-Push. Of course, (1) says that $f(n) \in O(n)$ which is acceptable in scalability.

C. Space Complexity

Let us check the space complexities for the related schemes in the following paragraphs.

C.1 Traditional HAS

In the traditional HAS, no extra space is required because HTTP/1.1 is a stateless protocol.

C.2 K-Push and Dynamic K-Push

Due to the fact that HTTP/2, which keeps some states because the HPACK header compression scheme employed by HTTP/2 is stateful, is employed by K-Push and Dynamic K-Push, the video server allocates extra memory space denoted by C_{http2} for each connection. The space complexity associated with the two schemes denoted by $g(n)$ when n concurrent connections exist is derived as follows:

$$g(n) = nCC_{\text{http2}}, \tag{2}$$

which says that $g(n) \in O(n)$.

C.3 Our Proposed Push Scheme

For our bitrate adaptation method, the video server keeps the following states for each connection: 1) idx, 2) br, 3) T_s, 4) b, and 5) s. Therefore, the extra memory space for each connection denoted by C_{mem} is $C_{\text{http2}} + 5$. The space complexity associated with our proposed scheme denoted by $h(n)$ when n concurrent connections exist is derived as follows:

$$h(n) = nCC_{\text{mem}} = nC(C_{\text{http2}} + 5), \tag{3}$$

which says that $h(n) \in O(n)$.

C.4 Comparison among the Proposed Scheme, K-Push, Dynamic K-Push, and the Traditional HAS

Considering n concurrent connections, the extra memory space required for our proposed scheme is $5nC$ as compared to K-Push, Dynamic K-Push. Likewise, the extra memory space required for our proposed scheme is $nC(C_{\text{http2}} + 5)$ as compared to the traditional HAS. Actually, either $5nC$ or $nC(C_{\text{http2}} + 5)$ is negligible, implying that the complexity involved with our proposed scheme is still acceptable.

V. NUMERICAL RESULTS AND DISCUSSIONS

In this section, the performance comparison between our proposed scheme and the closely related schemes in the literature will be shown by simulation. Such a comparison will be illustrated via the following performance metrics: 1) average (achievable) bitrate, 2) the number of buffer stalls, i.e., buffer emptiness/exhaustion, 3) the number of HTTP requests, 4) the volume of unclaimed pushes, 5) the ratio of unclaimed pushes, which is the ratio of the volume of unclaimed pushes and the volume of total pushes.

A. System Architecture and Simulation Environment

Our simulation system will be developed by Node.js. In addition, the node-http2 [29] HTTP/2 protocol stack developed by Google is leveraged for the system of our simulations. Furthermore, K-Push [8] and Dynamic K-Push [9] are implemented via Node.js as well for the purpose of comparison. As for the details of the system architecture and simulation environment, these are depicted in the following paragraphs.

A.1 Traffic Shaping at the Server

For part of our simulations, the bandwidth change and the RTT variation for mobile devices under a real scenario will be emulated to acquire the performance of the push schemes considered under such a scenario. This is done by a traffic shaping mechanism at the server with the CentOS. By analyzing the bandwidth trace log [30] previously employed by Dynamic K-Push [9] using traffic shaping and by applying the traffic control provided by the Linux kernel of the CentOS at the server, the bandwidth change and the RTT variation are emulated accordingly.

Considering the framework of K-Push and Dynamic K-Push at a client-side, the simulator at a client-side for our proposed scheme can be built with suitable modification, i.e., the bitrate adaptation logic is moved to the server. Therefore, the client receives video segments passively from the video server and buffers/plays a video segment according to the current state. Moreover, a performance-profiler is implemented in our simulator to measure performance metrics.

A.2 Simulation Environment

About the version of Node.js employed, it is 4.4.5. For the client, it is built on the Intel Core i7-2600 CPU with the operation system of Microsoft Windows 7.0 SP1 x86_64 and 16GB DDR3-1066 memory. As for the server, it is built on the Intel Core i7-2600 CPU with the operation system of CentOS 6.2 x86_64, the OS kernel of version 2.6.32-220.el6.x86_64, and 16GB DDR3-1066 memory. As for the parameters of the simulation environment, they are listed in Table 2. Note that two different scenarios are considered explicitly in our following simulations: a Gigabit Ethernet and the Internet with Mobile Clients (under different RTTs) to consider the dynamic changes in the network, e.g., congestion. In the Gigabit Ethernet, the abundant bandwidth is assumed. Therefore, no congestion is incurred in such a scenario. However, the dynamic changes, e.g., congestion, are considered for the Internet with Mobile Clients (under

Table 2. Parameter setting for the simulation.

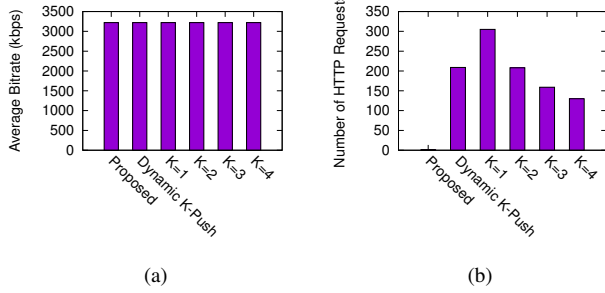| Parameter	Setting/Description
buf_{min}, buf	12 s, 16 s
c, ρ, α	1 s, 0.35, 0.3
DASH dataset	Well-known "Big Buck Bunny" DASH Dataset [31], Video size: 9 m 55 s, Segment duration : 1 s, Number of segments : 596, Representations (kbps) : 220.81, 414.57, 606.16, 789.12, 1046.42, 1282.02, 1623.84, 2181.78, 2555.94, 3227.65
Real world bandwidth trace	HSDPA-bandwidth logs for the mobile HTTP streaming scenario [30]

Fig. 2. Performance of different push schemes in a Gigabit Ethernet: (a) Average bitrate and (b) number of HTTP requests.
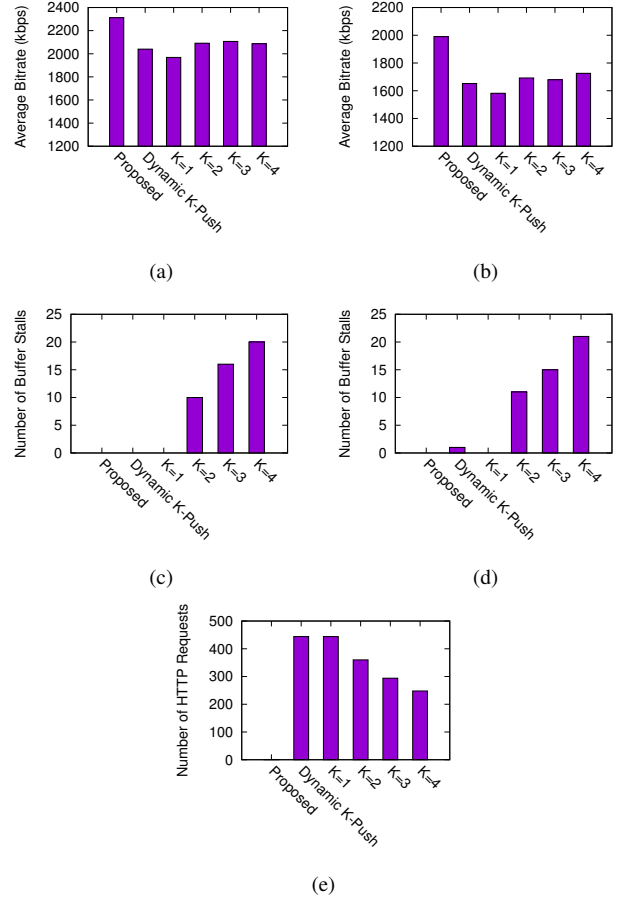

Fig. 3. Average bitrates, numbers of buffer stalls, and numbers of HTTP requests under different RTTs: (a) RTT = 50 ms, (b) RTT = 100 ms, (c) RTT = 50 ms, (d) RTT = 100 ms, and (e) RTT = 100 ms.

different RTTs). Different values of RTTs are used to reflect the changes in the network for sure.

B. Simulation Results and Discussions

B.1 Simulation in a Gigabit Ethernet

Shown in Fig. 2 are the simulation results in a Gigabit Ethernet. Due to the abundant bandwidth of the Gigabit Ethernet, no unclaimed pushes exist and the average video bitrates achieved by different push schemes are almost the same and reach 3220 kbps as shown in Fig. 2(a) because the highest video quality at 3227.65 kbps is almost affordable during the whole playback. Note that the video stream starts from the lowest video quality at 220.81 kbps initially and then switches to the highest bitrate later. This explains why the average bitrates are slightly lower than the maximally affordable bitrate. Because the total number of video segments is 596, the traditional HAS requires 597 HTTP requests as shown in Fig. 2(b) for the whole video, including the one for the MPD file. As for K-Push and Dynamic K-Push, the number of HTTP requests can be lowered. The higher K is, the fewer HTTP requests are sent. Last but not least, our proposed scheme only requires one HTTP request.

B.2 Simulation for the Internet with Mobile Clients under Different RTTs

The bitrates under such a scenario are shown in Figs. 3(a) and 3(b) with RTTs of 50 ms and 100 ms, respectively. Explicitly, our proposed scheme is affected the most by RTT which can stand for the congestion level. When the more congested network condition, i.e., the condition with RTT of 100 ms, is posed, the bitrate drops significantly. However, our proposed scheme achieves the highest average bitrate among the related schemes. As for the average bitrates of K-Push with $K = 2, 3, 4$, they are a bit higher than those of Dynamic K-Push and K-Push with

$K = 1$. When the value of RTT is getting larger (from 50 ms to 100 ms), the improvement regarding the average achievable bitrate gained by our proposed scheme becomes more apparent as compared to the other schemes as shown in Figs. 3(a) and 3(b). Further examining the number of buffer stalls as shown in Figs. 3(c) and 3(d), K-Push with $K = 2, 3, 4$ incur more buffer stalls (and playback stalls accordingly) than our proposed scheme, K-Push with $K = 1$, and Dynamic K-Push because more segments cause severe network bandwidth competition. Actually, there are no buffer stalls for our proposed scheme and K-Push with $K = 1$ under both RTT values. As for Dynamic K-Push, it is found once regarding the buffer stall when the value of RTT is 100 ms. Shown in Fig. 3(e) are the numbers of HTTP requests for all schemes when the value of RTT is 100 ms. From this figure, one can see that our proposed scheme only requires one single HTTP request no matter what the video size is. On the contrary, the numbers of HTTP requests are still quite large (ranging from 248 to 444) for the other schemes as compared to our proposed scheme even if K-Push with a large K can lower the number of HTTP requests greatly. To check the video buffer levels for different schemes during the whole video playback, one can refer to Fig. 4. Unlike K-Push with $K = 2, 3, 4$, almost no occurrence of buffer emptiness/exhaustion is found for K-Push with $K = 1$, Dynamic K-Push, and our proposed scheme. According to Fig. 3(b), our proposed scheme provides
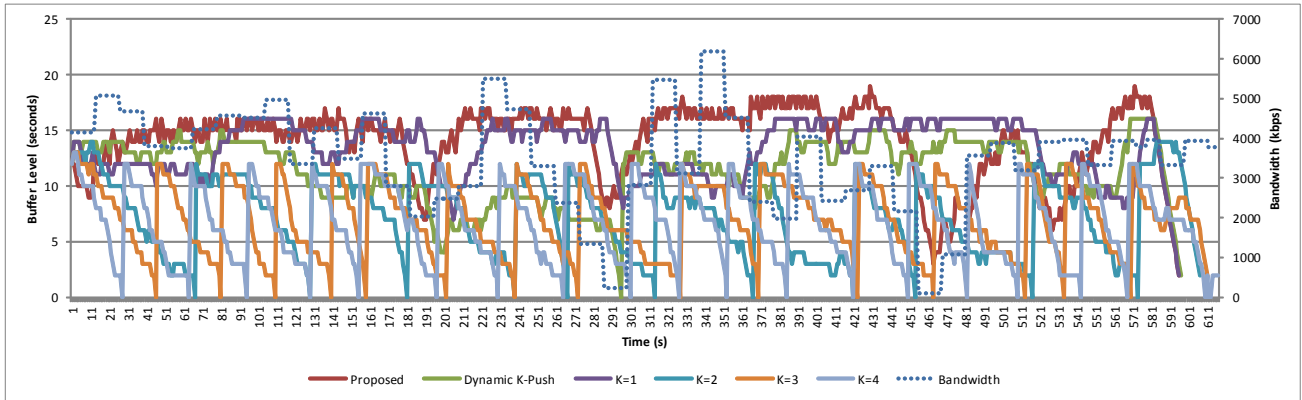
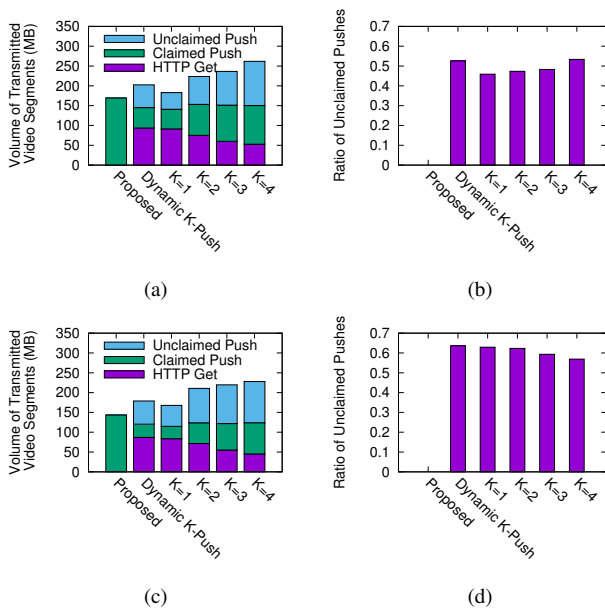Fig. 4. The buffer levels during the whole video playback when RTT = 100 ms.

Fig. 5. Volumes of different transmitted video segments and ratios of unclaimed pushes in the Internet under different RTTs: (a) RTT = 50 ms, (b) RTT = 50 ms, (c) RTT = 100 ms, and (d) RTT = 100 ms.

20% and 26% higher in the average bitrate than Dynamic K-Push and the K-Push with $K = 1$, respectively. As for K-Push with $K = 2, 3, 4$, frequent occurrences of buffer emptiness/exhaustion caused by insufficient bandwidth to carry video segments are observed, i.e., 10, 16, and 20 times. Compared to K-Push with $K = 4$, our proposed scheme has 15% higher in the average bitrate. The aforementioned observations explicitly indicate that our proposed scheme achieves better video quality than the other related schemes.

B.3 Further Investigation in the Internet with Mobile Clients

With worse network conditions, unstable network bandwidth is inevitable. To adapt network bandwidth changes, more frequent video bitrate changes are observed, causing unclaimed pushes easily for K-Push and Dynamic K-Push. This is investigated in the following. When the value of RTT is 50 ms,

Dynamic K-Push and K-Push switch bitrates frequently, causing a lot of unclaimed pushes as shown in Fig. 5(a). Dynamic K-Push wastes 57.27 MB and K-Push with $K = 4$ wastes 111.57 MB during the whole video playback. In terms of the ratio of unclaimed pushes as shown in Fig. 5(b), the percentages of wastage revealed by Dynamic K-Push and K-Push with $K = 4$ are 52.59% and 53.35%, respectively. For our proposed scheme, such a problem can be removed and avoided. Therefore, no unclaimed pushes are observed at all for our proposed scheme. When the RTT gets larger, i.e., 100 ms, the average achievable bitrate drops and the volume of unclaimed pushes grows for both K-Push and Dynamic K-Push. For this case, Dynamic K-push wastes 58.59 MB and K-Push with $K = 4$ wastes 103.95 MB in pushing unclaimed segments as shown in Fig. 5(c) with 63.62% and 56.90%, respectively, of wastage as shown in Fig. 5(d). It explicitly says that a worse network condition makes even worse performance for both Dynamic K-Push and K-Push. Unlike these two schemes, Our proposed scheme does not suffer from any unclaimed pushes at all, revealing that it can work much better even in a harsh network condition. In Table 3, the simulation results when RTT = 100 ms are summarized. It clearly shows that our proposed scheme outperforms the other schemes in terms of the average achievable video bitrate, the number of buffer stalls, the number of HTTP requests, and the ratio of unclaimed pushes.

VI. CONCLUSIONS

To solve the drawbacks incurred by HTTP/2, K-Push, and Dynamic K-Push, an adaptive server-paced push scheme is proposed in this paper. Our proposed scheme moves bitrate adaptation to the server-side and integrates server-paced push and bitrate adaptation properly. It not only successfully lowers the number of HTTP requests to one but also improves the QoS of video streaming greatly in terms of the achievable bitrate, the number of buffer stalls, and the ratio of unclaimed pushes with acceptable extra overheads to the sever checked by complexity analysis. Checking via simulations, in particular, in a harsh network environment with a long RTT, i.e., 100 ms, our proposed scheme achieves the highest video bitrate among the closely re-

Table 3. Summary of simulation results.

	Proposed	Dynamic K-Push	$K = 1$	$K = 2$	$K = 3$	$K = 4$
Average Bitrate (kbps)	1990.13	1652.09	1581.43	1692.16	1679.14	1725.69
Number of Buffer Stalls	0	1	0	11	15	21
Number of HTTP Requests	1	444	444	360	294	248
Ratio of Unclaimed Pushes	0%	63.62%	62.90%	62.33%	59.35%	56.90%
Total Pushed Data (MB)	143.85	92.10	84	139.08	164.40	182.68
Unclaimed Pushes (MB)	0	58.59	52.83	86.68	97.57	103.95

lated schemes. The observed percentages of improvement in the achievable video bitrate are at least 15%. Unlike Dynamic K-Push and K-Push, in particular, K-Push, our proposed scheme avoids playback stalls caused by emptiness/exhaustion of the video buffer. As for the ratio of unclaimed pushes, no unclaimed pushes are brought by our proposed scheme, while Dynamic K-Push and K-Push incur at least 56.9% of unclaimed pushes. These observations firmly support our proposed adaptive server-paced push scheme and highly recommend it for adoption by video streaming over HTTP/2 for sure.

REFERENCES

[1] A. Begen, T. Akgul, and M. Baugher, "Watching video over the web: Part 2: Applications, standardization, and open issues," *IEEE Internet Comput.*, vol. 15, no. 3, pp. 59–63, Dec. 2011.

[2] ISO/IEC 23009-1, "Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats," International Organization for Standardization, Mar. 2012.

[3] A. Begen, T. Akgul, and M. Baugher, "Watching video over the web: Part 1: Streaming protocols," *IEEE Internet Comput.*, vol. 15, no. 2, pp. 54–63, Dec. 2011.

[4] T. C. Thang, H. T. Le, H. X. Nguyen, A. T. Pham, J. W. Kang, and Y. M. Ro, "Adaptive video streaming over HTTP with dynamic resource estimation," *J. Commun. Netw.*, vol. 15, no. 6, pp. 635–644, Jan. 2013.

[5] S. Wei and V. Swaminathan, "Low latency live video streaming over HTTP 2.0," in *Proc. ACM NOSSDAV*, 2014, p. 37.

[6] M. A. Hoque, M. Siekkinen, and J. K. Nurminen, "Energy efficient multimedia streaming to mobile devices – A survey," *IEEE Commun. Surveys Tutorials*, vol. 16, no. 1, pp. 579–597, Nov. 2014.

[7] G. Tian and Y. Liu, "On adaptive HTTP streaming to mobile devices," in *Proc. IEEE PV*, Dec. 2013, pp. 1–8.

[8] S. Wei and V. Swaminathan, "Cost effective video streaming using server push over HTTP 2.0," in *Proc. IEEE MMSP*, Sept. 2014, pp. 1–5.

[9] D. V. Nguyen, H. T. Le, P. N. Nam, A. T. Pham, and T. C. Thang, "Adaptation method for video streaming over HTTP/2," *IEICE Commun. Express*, vol. 5, no. 3, pp. 69–73, Jan. 2016.

[10] ISO/IEC 23009-6, "Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 6: DASH with server push and WebSockets," International Organization for Standardization, Feb. 2016.

[11] M. Belshe, R. Peon, and M. Thomson, "RFC 7540: Hypertext transfer protocol version 2 (HTTP/2)," *Internet Engineering Task Force*, May 2015.

[12] S. Wei, V. Swaminathan, and M. Xiao, "Power efficient mobile video streaming using HTTP/2 server push," in *Proc. IEEE MMSP*, Oct. 2015, pp. 1–6.

[13] J. Samain, G. Carofiglio, L. Muscariello, M. Papalini, M. Sardara, M. Tortelli, and D. Rossi, "Dynamic adaptive video streaming: Towards a systematic comparison of ICN and TCP/IP," *IEEE Trans. Multimedia*, vol. 19, no. 10, pp. 2166–2181, 2017.

[14] C. Mueller, S. Lederer, C. Timmerer, and H. Hellwagner, "Dynamic adaptive streaming over HTTP/2.0," in *Proc. IEEE ICME*, July 2013, pp. 1–6.

[15] R. Huysegems, J. van der Hooft, T. Bostoen, P. Rondao Alface, S. Petrangeli, T. Wauters, and F. De Turck, "HTTP/2-based methods to improve the live experience of adaptive streaming," in *Proc. ACM Multimedia*, Oct. 2015, pp. 541–550.

[16] M. Aguayo, L. Bellido, C. M. Lentisco, and E. Pastor, "Dash adaptation algorithm based on adaptive forgetting factor estimation," *IEEE Trans. Multimedia*, vol. 20, no. 5, pp. 1224–1232, May 2018.

[17] 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 *Proc. ACM MMSys*, Feb. 2011, pp. 257–264.

[18] J. van der Hooft *et al.*, "An HTTP/2 push-based approach for SVC adaptive streaming," in *Proc. IEEE/IFIP NOMS*, April 2016, pp. 104–111.

[19] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori, "DASH fast start using HTTP/2," in *Proc. ACM NOSSDAV*, March 2015, pp. 25–30.

[20] N. Bouzakaria, C. Concolato, and J. Le Feuvre, "Fast DASH bootstrap," in *Proc. IEEE MMSP*, Oct. 2015, pp. 1–6.

[21] J. van der Hooft *et al.*,, "HTTP/2-based adaptive streaming of HEVC video over 4G/LTE networks," *IEEE Commun. Lett.*, vol. 20, no. 11, pp. 2177–2180, Aug. 2016.

[22] N. Bouten, S. Latré, J. Famaey, F. De Turck, and W. Van Leekwijck, "Minimizing the impact of delay on live SVC-based HTTP adaptive streaming services," in *Proc. IFIP/IEEE IM*, May 2013, pp. 1399–1404.

[23] R. Peon and H. Ruellan, "HPACK: Header compression for HTTP/2," Tech. Rep., May 2015.

[24] G. Tian and Y. Liu, "Towards agile and smooth video adaptation in dynamic HTTP streaming," in *Proc. ACM CoNEXT*, Dec. 2012, pp. 109–120.

[25] T.-Y. Huang, R. Johari, and N. McKeown, "Downton abbey without the hiccups: Buffer-based rate adaptation for HTTP video streaming," in *Proc. ACM SIGCOMM workshop*, Aug. 2013, pp. 9–14.

[26] L. De Cicco, S. Mascolo, and V. Palmisano, "Feedback control for adaptive live video streaming," in *Proc. ACM MMSys*, Feb. 2011, pp. 145–156.

[27] T. C. Thang, H. T. Le, A. T. Pham, and Y. M. Ro, "An evaluation of bitrate adaptation methods for HTTP live streaming," *IEEE J. Sel. Areas Commun.*, vol. 32, no. 4, pp. 693–705, Mar. 2014.

[28] Y. Shuai and T. Herfet, "Improving user experience in low-latency adaptive streaming by stabilizing buffer dynamics," in *Proc. IEEE CCNC*, Jan. 2016, pp. 375–380.

[29] "node-http/2," [Online]. Available: https://github.com/molnarg/node-http2

[30] "Dataset: HSDPA-bandwidth logs for mobile HTTP streaming scenarios," [Online]. Available: http://skuld.cs.umass.edu/traces/mmsys/2013/pathbandwidth/

[31] "Big Buck Bunny Movie," [Online]. Available: http://www.bigbuckbunny.org
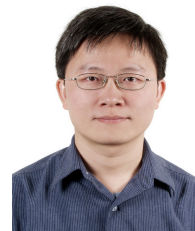
Huei-Wen Ferng received the B.S. degree in Electrical Engineering from the National Tsing Hua University, Hsinchu, Taiwan, in 1993 and the Ph.D. degree in Electrical Engineering from the National Taiwan University, Taipei, Taiwan, in 2000. He joined the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, as an Assistant Professor in August 2001. From February 2005 to January 2011, he was an Associate Professor. Since February 2011 and June 2012, he has been a Professor and a Distinguished Professor, respectively. From August 2016 to July 2019, he was the department head. Funded by the Pan Wen-Yuan Foundation, Taiwan, he spent the summer of 2003 visiting the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, U.S.A. His research interests include Internet protocols, video streaming, wireless networks, mobile computing, high-speed networks, protocol design, teletraffic modeling, queuing theory, security, and performance analysis. He was a recipient of the research award for young researchers from the Pan Wen-Yuan Foundation, Taiwan, in 2003 and was a recipient of the Outstanding Young Electrical Engineer Award from the Chinese Institute of Electrical Engineering (CIEE), Taiwan, in 2008. He is a senior member of the IEEE.

Shan-Hsiang Shen received the M.S. degree from National Chiao Tung University, R.O.C., in 2004, and the Ph.D. degree from University of Wisconsin, U.S.A., in 2014. He is currently an Associate Professor with the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, R.O.C. His main research interests include software-defined networking, network function virtualization, network security, and cloud computing.

Chih-Wei Lai received the B.S. degree in Information Management from the National Central University, Taoyuan, Taiwan, in 1998, and the M.S. degree in Computer Science and Information Engineering from the National Taiwan University of Science and Technology, Taipei, Taiwan, in 2017. He is currently a Senior Staff Software Engineer and Researcher of the Trend Micro Inc., a cyber security software company in Taiwan. His research interests include Internet protocols, video streaming, and anti-malware technologies, e.g., sandboxing, deep packet inspection, etc.