Evaluating and Improving Push based Video Streaming with HTTP/2

Mengbai Xiao[1] Viswanathan Swaminathan[2] Sheng Wei[2,3] Songqing Chen[1]

[1]Dept. of CS
George Mason University
{mxiao3, sqchen}@gmu.edu

[2]Adobe Research
Adobe Systems Inc.
vishy@adobe.com

[3]Dept. of CSE
University of Nebraska-Lincoln
shengwei@unl.edu

ABSTRACT

The sever-initiated push mechanism is one of the most prominent features in the next generation HTTP/2 protocol, having shown its capability on saving network traffic and improving the web page retrieval latency. Our prior work has investigated the server push-based mechanism for HTTP video streaming and proposed a *k-push* scheme, where the server pushes *k* video segments following the response to a request. In this study, we further conduct an analysis and evaluation of the *k-push* scheme in HTTP streaming. Our results uncover that the push mechanism can efficiently increase the network utilization (under certain conditions) compared to regular HTTP streaming. However the results also show that the *k-push* scheme deteriorates network adaptability and leads to the "over-push" problem, in which the pushed video content waste network resources due to user abandonment behaviors. To overcome these limitations, we propose a new " *adaptive-push*" scheme, which dynamically adjusts the parameter *k* to adapt to the runtime environment. To evaluate the performance of *adaptive-push*, we implemented a prototype system. The experimental results show that compared to *k-push*, *adaptive-push* can improve the network adaptability. Furthermore, our real-world trace based simulation results show that *adaptive-push* can effectively alleviate the over-push problem.

CCS Concepts

•**Networks** → *Application layer protocols;*

Keywords

HTTP streaming; Video streaming

1. INTRODUCTION

Internet video streaming has gained a huge amount of popularity in the recent years. Cisco predicts that the Internet video traffic will be 80 percent of all consumer Internet traffic in 2019, up from 64 percent in 2014 [3]. Much of today's Internet video traffic is delivered via HTTP streaming, which is widely deployed by the content providers, such as YouTube [8] and Netflix [14]. In HTTP streaming, the video is usually chunked into segments with fixed duration and further encoded into multiple bit rate levels. These segments are served to clients as HTTP responses where the desired bit rates are designated in the corresponding requests. The popularity of HTTP streaming results from not only its combination to the multiple bit rates encoding, which yields the ability of client reacting to the bandwidth variation, but also the ease of deployment-it takes advantage of the existing HTTP based web content delivery architecture, such as the content delivery networks (CDN), without additional infrastructure. The major implementations of HTTP streaming include Adobe HDS [13], Apple HLS [12], and Dynamic Adaptive Streaming over HTTP (DASH) [18].

However, the HTTP streaming implementations can barely exploit all the available bandwidth of the underlying links due to not only the TCP repetitive sawtooth pattern traffic but also the overhead of the HTTP requests. To maximize the utilization of available bandwidth, the segment duration should be as long as possible. But this can lead to unnecessary network traffic if the video session is abandoned in the early stages. The problem becomes more prominent in live streaming since the live latency is determined by the segment duration [22, 19]. Although a shorter segment duration benefits the live latency, it essentially lowers the network utilization while cramming in more requests, which is also known as *request explosion* [22].

HTTP/2 helps address some of these problems. HTTP/2 features the server-initiated push mechanism, which allows the HTTP server to push back additional responses in advance without receiving the corresponding requests. This characteristic provides an opportunity for eliminating the request overhead with short segment duration. Our prior study [22] investigated the potential of HTTP/2 for HTTP streaming and showed that the server-initiated push mechanism is helpful in improving live latency with a *k-push* scheme, in which addition *k* segments are pushed by the server in response to one request.

In this study, we for the first time conduct an analysis and evaluation of the *k-push* scheme. Our results show that while the *k-push* scheme can improve the underlying network bandwidth utilization for HTTP streaming, it also leads to several limitations: 1) there is diminishing marginal returns with the increasing number of pushed segments, i.e., the parameter *k*. The gain of streaming throughput is not linearly related to the increment of *k*; 2) *k-push* leads to a degraded adaptability; and 3) *k-push* also causes the over-push problem.

To address these limitations, in this study, we propose *adaptive-push*, which dynamically scales the number of segments pushed, *k*, during the video playback at runtime. In a HTTP streaming session, the number of segments pushed is scaled up for higher network utilization, while it is also constrained if the network adaptability is impacted. Furthermore, since in practice a majority of video sessions are abandoned in the first few seconds, *adaptive-push* also suggests a small initial *k*, which can increase as the user's session progresses. To evaluate the performance of our *adaptive-push*

NOSSDAV'16, May 13 2016, Klagenfurt, Austria

DOI: http://dx.doi.org/10.1145/2910642.2910652

scheme, we have implemented a prototype. The experimental results show that *adaptive-push* is able to greatly improve the network utilization and adaptability. Furthermore, our simulations based on a large-scale real-world trace show the ability of *adaptive-push* on alleviating the over-push problem.

2. RELATED WORK

In this section, we briefly introduce HTTP streaming and HTTP/2 with relevant background information.

2.1 HTTP Streaming

In HTTP streaming, the video quality can be properly selected based on varying bandwidth by the client (i.e., video player). Many prior studies focused on the quality selection algorithms. QDASH [16] integrates a proxy-like bandwidth measurement component to accurately and promptly draw the network bandwidth. The authors also suggest a gradual quality switch algorithm to improve the subjective user-perceived quality. FESTIVE [15] identifies three prominent metrics, namely efficiency, fairness, and stability, in the HTTP streaming systems. A suite of techniques are developed to help make beneficial trade-offs among the metrics. Huang et al. [9] suggested that it is difficult to accurately estimate the underlying bandwidth above the HTTP layer according to the surveys conducted on popular video streaming services. Almost all the rate determination algorithms are implemented on the client side, but this ability will be constrained by the push mechanism. Given that the server is not involved in the quality determination makes the makes the integration of the push mechanism into HTTP streaming more challenging.

2.2 HTTP/2

HTTP/2 [11] originates from SPDY [6], which was developed by Google. Many prior efforts have focused on the performance of these two protocols. Cardaci et al. [1] showed that the SPDY protocol slightly outperformed HTTP/1.1 over the high latency satellite links. Furthermore, according to a detailed measurement over four months [5], SPDY does not clearly outperform the HTTP/1.1 over cellular networks, which lacks of harmonious interaction with TCP. Since the HTTP video streaming services are built on top of the HTTP protocol, the advancement from HTTP 1.1 to HTTP/2, as well as the study in HTTP/2, benefits the performance and user experiences in video streaming as well. In particular, we focus on the study of potential benefits provided by new features in HTTP/2, such as server push as discussed in the next subsection.

2.3 Server Push-based Streaming

The server-initiated push in HTTP/2 is a mechanism designed to help reduce web page load latency. To push a HTTP response, the server initiates a special frame named *push promise*. When receiving the push promise frame, the HTTP client will not send out the corresponding request until the response is pushed to the client completely. After that, the client directly retrieves the response from the browser cache. HTTP/2 only designates a mechanism of how push works, and the concrete implementation is left to the application.

With the push mechanism proposed in HTTP/2, there have been studies in the literature to improve and evaluate HTTP/2 for video streaming. For example, Mueller et al. [17] explored the performance of DASH-compliant streaming over HTTP/2. Sheng et al. studied the potential of the push mechanism in reducing the live latency [22], eliminating unnecessary requests [21] and saving power in 4G networks [23]. In [10], the authors focused on improving the live experience of HTTP streaming by exploiting the HTTP/2 features. One major strategy in this work is to adopt *full-push* while additional messages are used to control the video quality adaptation. Wael Cherif et al. [2] also investigated the push mechanism for fast start in the DASH-compliant video streaming.

Table 1: Experimental Parameters

Video Length (s)	120
Video Quality (kbps)	{49.2, 217.2, 504, 752}
Segment Duration (s)	{1, 2}
Push Number k	{0, 1, 4, 9}
Bandwidth (kbps)	{200, 560, 880}
RTT (ms)	{20, 300, 500}

3. K-PUSH ANALYSIS

3.1 K-Push and Rate Adaption Difficulty

To adopt the push mechanism for video streaming using HTTP, We have proposed the *k-push* scheme in our prior work [22]. If a multimedia stream applies the *k-push* scheme, its HTTP sessions are then composed of continuous *push cycles*. In each push cycle, there are $k+1$ segments involved, where $k \geq 0$. The first segment acquired in a regular HTTP session is called *lead segment*. The request of the lead segment designates the number of segments pushed k and implies the bit rate level in this push cycle. Figure 1 shows a comparison between regular HTTP streaming and *k-push* based HTTP streaming. Figure 1 (a) shows a regular HTTP streaming session without push, which is referred as a *no-push* scheme. Figure 1 (b) shows a typical HTTP streaming session that applies the *k-push* scheme. The *k-push* scheme performs like the *no-push* scheme whenever $k = 0$.

It is worth noting that there is one key assumption in the proposed *k-push* scheme. To preserve the stateless nature of the HTTP server, it is essential that the server is not responsible for video quality adaptation. From the client side, the adaptation within one push cycle is also difficult to accomplish. This is because most clients are browser clients, where there are few interfaces exposed to the application level to cancel previous HTTP sessions. Hence the server pushes the following k segments in the same quality level as the lead segment.

3.2 Playback Bandwidth

The multimedia streams benefit from the *k-push* scheme in several aspects, especially the live latency [22]. The previous work identifies that the live streaming can take advantage of the push mechanism since the primary obstacle for improving live latency, i.e. reducing the segment duration, is the *request explosion* problem [22]. The HTTP request rate can be drawn from $\frac{1}{(k+1)D}$, where D is the segment duration. The *k-push* scheme can effectively alleviate this problem by eliminating $\frac{k}{k+1}$ requests.

To empirically investigate why the streaming sessions can benefit from the push mechanism and its potential problem, we conduct an experiment playing the same VOD session under different network conditions and various k's. The video is encoded in 4 bit rate levels. Table 1 shows the bit rate levels and other experimental parameters. By carefully capping the network bandwidth at different values above the video bit rate levels, the network bandwidth utilization can be analyzed by directly observing the average video quality. Figures 2 and 3 show the results when bandwidth is set as 880 kbps. In these figures, the number of segments pushed (i.e., the parameter k) is shown in the x-axis while the y-axis is the average bit rate in kbps. Note that $k = 0$ represents the *no-push* scheme. The Individual histogram in a cluster shows the result of different RTT.

From the Figure 2 and 3, we can observe that a shorter segment duration leads to a lower video quality in the *no-push* case, and this

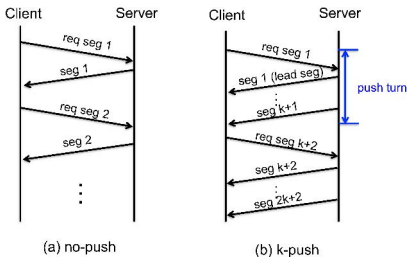

Figure 1: *no-push* and *k-push*

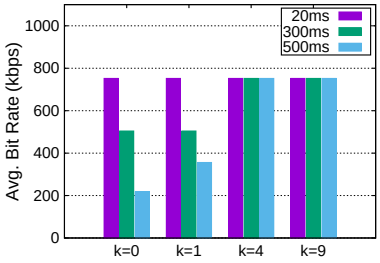

Figure 2: segment (2s) quality in different network contexts when varying push number

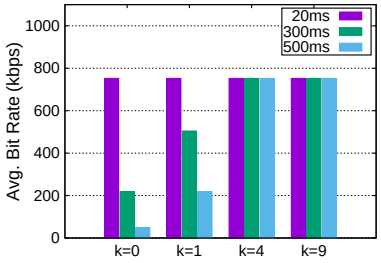

Figure 3: segment (1s) quality in different network contexts when varying push number

can be fixed by increasing k. This confirms the *request explosion* problem when reducing the segment duration. More interestingly, the different RTTs lead to the same situation, which degrades the segment quality in the *no-push* but gets fixed with increased k. By observing this, we note that not all available network bandwidth is used to transfer the payload of video/audio data in HTTP streaming. HTTP requests play a part in shrinking the actual bandwidth to the *playback bandwidth*, which is denoted as the effective bandwidth used to deliver the video payload. Increasing either the single request time (higher RTT) or the request rate (shorter segment duration) leads to more time consumed by HTTP requests. In the case where there is not enough playback bandwidth, lower quality levels are selected even though the actual bandwidth (880 kbps) is substantially higher than that for quality requirement (752 kbps). By gradually eliminating this part of overhead by designating a higher push number, we find that the playback bandwidth approaches the actual bandwidth.

3.3 Beyond Playback Bandwidth

From the previous experiments, we also find that it is not worth continuing to increase the number of segments pushed for higher playback bandwidth. Firstly, the marginal return of increasing k is diminishing. And secondly, the adaptability is sacrificed because the rate adaptation is not allowed in one push cycle (Section 3.1). Furthermore, the bandwidth is wasted by the pushed content if the user drops off the video session without finishing the playback of all the pushed content. We discuss these problems next in detail.

3.3.1 Playback Bandwidth Variation

From the experiments in the last section, we observe that the gains of *k-push* diminishes with higher k. Figures 2 and 3 represent significant video quality improvement between the cases of $k = 0$ and $k = 1$. However, there is no perceptible improvement when k is increased from 4 to 9. The gains of *k-push* come from the reduction of HTTP requests, and these benefits approach a fixed proportion when k is relatively large. The playback bandwidth is also pushed up to the actual bandwidth.

3.3.2 Network Adaptability

For one segment, theoretically, the time consumed to transfer it is expected to be less than the segment duration. Otherwise, the video playback will stall. However, any temporary additional (superfluous) time can be absorbed by the playback buffer in practice. When the segment transmission experiences inadequate bandwidth, the playback would be continuous as long as the buffer length (calculated in time) is longer than the superfluous time.

It is obvious that the minimum required buffer is larger when more segments are involved in one push cycle. Since video quality adaptation cannot occur in the middle of one push cycle, more time is needed to finish a push cycle with more video segments whenever the actual bandwidth drops. As a result, the *k-push* scheme undermines the network adaptability.

3.3.3 Over-push Problem

Another limitation of *k-push* is the over-push problem. The user may decide not to continue watching a video after checking the first few seconds of the video. In this case, a large push number k would risk downloading more video segments than required. Therefore k should be determined more wisely to avoid or minimize such waste.

4. DESIGN AND IMPLEMENTATION OF ADAPTIVE PUSH

The last section showed that the *k-push* scheme has a few disadvantages while it does have some advantages in HTTP streaming. In this section, motivated by the results from the previous section, we propose a new push scheme, called *adaptive-push*, to take advantage of the push mechanism in HTTP/2 while minimizing its negative impact. Both the *k-push* and the *adaptive-push* are orthogonal to the quality selection algorithms, irrespective of whether these algorithms exploit the playback bandwidth or not.

4.1 Algorithm Description

The core function of *adaptive-push* is named *next-k* and is executed between push cycles. It dynamically scales k for the next push cycle while leveraging the factors of playback bandwidth variation, network adaptability and over-push problem. *Adaptive-push* iteratively invokes the *next-k* function until the end of the video session. Note that this scheme is only required on the client side, and no effort is required on the server side.

At a high level, the *next-k* works as follows: 1) For the very first cycle, an initial k is used; 2) k is increased according to the old value from the last push cycle; and 3) the k value is also capped by both leveraging the current buffer level and predicting the future bandwidth variation.

How to initialize k: To minimize or even eliminate the over-push problem, a small value, such as 0, is used for the first push cycle.

How to increase k: Because of the diminishing marginal return on playback bandwidth and the risk of draining the playback buffer, *adaptive-push* increases k with two different rates. When k is small, it is increased at a fast rate to effectively approach the actual bandwidth. Otherwise, a slower rate is more appropriate when the playback bandwidth is high enough. T_1 and T_2 are used to represent two thresholds that slow or stop the increment of k.

How to cap k: It is straightforward that a valid k needs to meet the constraint that the future bandwidth can support segment transmission in current quality without consuming up the playback buffer

$$(k+1)(\frac{bD}{B} - D) < L,$$

where b is the current bitrate, D is the segment duration, B is the predicted bandwidth, and L is the buffer length. As a result, k is decreased to an appropriate value without degrading the network adaptability.

4.2 Push Implementation

To evaluate the performance of *adaptive-push*, we have implemented a prototype. We present the implementation details next.

For the ease of portability, as well as the compatibility to HTTP, *adaptive-push* uses HTTP header extensions for signaling k. An additional HTTP header field, named *PushDirective*, is embedded in the lead segments. The value of this field exhibits as the number of segments pushed k. After receiving a request with *PushDirective*, an HTTP server is aware of the desire of the client. The server can either agree with the request, launching a push cycle of $k+1$ segments, or it can scale down k according to some other considerations, e.g., the current workload. Server uses a header extension field called *PushAck* in the response to signal k, the number of segments it is going to push. The client is therefore capable of accommodating the agreed number and adjusting its behaviors, such as adapting the following k requests to the same bit rate level as the lead segment and deciding when to initiate the next push cycle.

4.3 Server and Client Implementation

In *adaptive-push*, both the HTTP server and the client should understand the push protocol. Therefore our prototype (and thus the experimental platform) consists of three components: a **HTTP/2 Server**, a **Video Player** and a **Network Shaper**.

HTTP/2 Server: We select the Jetty [4], a Java-based HTTP server, as the server-side implementation because of its HTTP/2 support. The push module is implemented as a *Filter* class. Once a passing request with *PushDirective* is identified as one to a DASH segment, the server launches a push cycle, where the corresponding segment and its following k segments are pushed back sequentially.

Video Player: we implement the video player based on the open source project *dash.js* [7], which is a DASH-compliant video player in JavaScript. The video player is packaged as a web application and deployed on the Jetty server. Whenever the player is about to start a push cycle, it will invoke the *next-k* function to determine k. This k is then sent as the *PushDirective*.

Network Shaper: in our experiments, we use network shaper to change network conditions and then observe the performance of the *adaptive-push* scheme. The network shaper throttles the network bandwidth and introduces a planned network delay. We adopt the Linux command line tool *tc* to manipulate these two parameters. The class *htb* is used for for bandwidth throttling and the queue discipline *netem* is used for changing the network delay.

5. PERFORMANCE EVALUATION

Based on the implemented prototype, we conduct experiments in order to quantify the performance of the *adaptive-push* scheme. All of our experiments are conducted on a Linux machine with a 64-bit Intel Pentium CPU 2.8 GHz dual core, 6 GB memory, 2×32 KB L1 caches, 2×256 KB L2 caches and shared 3 MB L3 cache. The installed operating system is Ubuntu 12.04 with Linux kernel 3.13.0-66-generic.

5.1 Playback Bandwidth

We extend the experiments in section 3.2 to evaluate how much playback bandwidth can be achieved in the *adaptive-push*. All experimental parameters are the same as those shown in Table 1. The buffer size is set as infinity and the video player downloads the video segments in a progressive manner. We directly measure the playback bandwidth instead of the average quality. The playback bandwidth can be derived by dividing the size of all downloaded segments by the time consumed. Due to page limit, we only report the experimental results with 1-second segment duration.

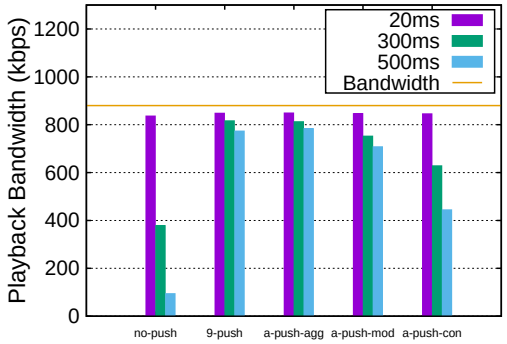

Figure 4: Playback bandwidth when actual bandwidth is 880 kbps and segment duration is 1 second

Figure 4 shows the results of experiments conducted with the network bandwidth of 880 kbps and the segment duration of 1 second. The x-axis represents different push schemes. *no-push* and 9-*push* are the original *k-push* scheme when k is set to 0 and 9, respectively. The remaining three push schemes are the *adaptive-push* with various configurations. The differences among them are how the future bandwidth is estimated. The *a-push-agg* aggressively estimates the future bandwidth as 880 kbps. The *a-push-mod* and the *a-push-con* predict the future bandwidth moderately and conservatively, as 415 kbps and 49.2 kbps, respectively. The y-axis is the measured playback bandwidth in kbps, and the histograms in a cluster represent various RTTs. Note that the actual bandwidth is marked in the figure with a horizontal line. As expected, all schemes have similar playback bandwidth when the network delay is low (20 ms), which means trivial request overhead. When the network delay is increased, the playback bandwidth diminishes quickly in *no-push*, just as in the previous experiments measuring the average video quality. Similar to what we observed before, the 9-push scheme fixes this problem. The performance of *a-push-agg* is almost as advantageous as the 9-*push* because k is not limited by the buffer length. The maximum k gets lower when the estimation of bandwidth is more conservative, leading to the lower playback bandwidth in both *a-push-mod* and *a-push-con*. However, the playback bandwidth observed in them are still better than the *no-push* scheme.

Figure 5 shows the variations of k in these experiments. The x-axis represents the video playback progress in seconds and the y-axis represents the corresponding number of segments pushed k. The lines in different colors represent the different RTTs. When the RTT is high (300 ms or 500 ms), we can see that the aggressive strategy (lines with cross) and the moderate strategy (solid lines) increase k very fast in the early stages. The curves clearly show the two rates of increasing k. If the conservative strategy is used, the buffer length becomes a constraint, and in this case, the k is at most 3.

The same experiments are also conducted while we cap the actual bandwidth at 560 kbps. The results are shown in Figure 6. The *a-push-med* and *a-push-con* perform better than in the high bandwidth cases. This is reasonable since the maximum k is also affected by the selected video quality in addition to the estimated bandwidth. In all cases the *adaptive-push* schemes achieve higher playback bandwidth because of the elimination of almost half of the requests.

Similarly, Figure 7 depicts the changes of the corresponding k in the experiments. As shown in the figures, the trends of variations

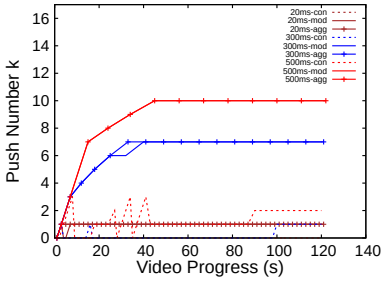

Figure 5: The variation of k during the playback when the actual bandwidth is 880 kbps and the segment duration is 1 second

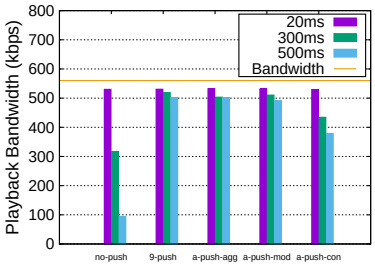

Figure 6: Playback bandwidth when actual bandwidth is 560 kbps and segment duration is 1 second

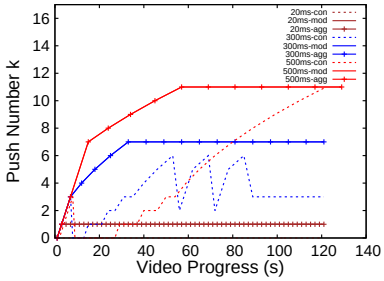

Figure 7: The variation of k during the playback when the actual bandwidth is 560 kbps and the segment duration is 1 second

Table 2: Buffer Length Statistics

	Buffer Length (s)	
	E	δ
no-push	29.08	1.69
9-push	16.46	10.34
a-push-con	23.55	7.90

are almost the same as those shown in Figure 5. In addition, when the bandwidth is 560 kbps, we can observe oscillations in the value of k when applying the conservative strategy. By further analyzing the dumped traces, we figure out the reason is the quality switch. In this case, the increasing k leads to enough playback bandwidth being sustainable towards the higher quality level. However, higher video quality imposes a more strict constraint on k while the buffer length is the same. Eventually the k and playback bandwidth drop again, restarting another loop.

5.2 Network Adaptability

To evaluate the network adaptability of various push schemes, we conduct the experiments under dynamic network conditions. The video played for this evaluation is 5 minute long. We conduct the experiment as follows. We do not throttle the network in the first 30 seconds for accumulating the playback buffer, and then change the network characteristics every 30 seconds. The bandwidth changes every 90 seconds in the sequence of 480 kbps, 640 kbps and 800 kbps. For each bandwidth, the RTT changes per 30 seconds as 20 ms, 300 ms and 500 ms. The buffer management strategy applied here is straightforward: 1) the client stops downloading the segments if the buffer length is greater than 30 seconds; 2) it resumes segment retrieval as long as the buffer drops below 30 seconds; and 3) additionally, the client does not stop acquiring in the middle of a push cycle. The experiments are conducted with three schemes: *no-push*, *9-push*, and *a-push-con*. Each experiment is repeated 5 times. Intuitively we expect that a scheme with better network adaptability will have a high average buffer level and a small standard deviation. The result is reported in the table 2. E denotes the expected value of buffer length and δ stands for the corresponding standard deviation. We find that the *no-push* scheme has the best network adaptability while it has the highest average buffer level and the lowest standard deviation. In contrast, the *9-push* scheme cannot respond to the network variation very well and it needs more buffer to absorb the jitter. The *a-push-con* makes a good tradeoff between these two schemes.

5.3 Over-pushed Video Content

In the most ideal case that every request is issued when watching last segment, the original HTTP streaming over-pushes at most one segment. However, this unnecessary network traffic may increase while the number of segments pushed grows in the *k-push* scheme. The *adaptive-push* is also capable of alleviating the over-push problem. To verify this, we conduct a few simulations based on a large scale of Apple HLS trace. This trace, which is measured on the mobile devices, ranges from 07/15/2015 to 08/31/2015, and there are $\sim$ 12 million records from the second largest mobile streaming service provider Vuclip [20]. In this trace, the logged information includes the time when users stop watching their videos. To study the over-push result, we implement a simulator in perl and configure the parameters similar to the previous experiments. In the simulator, we always select the highest video quality lower than the playback bandwidth of last push cycle, and the lowest quality is selected for the first push cycle.

Figure 8 shows the result of the simulation under the bandwidth of 200 kbps when the segment duration is 2 seconds. The y-axis represents the cumulative distribution function and the x-axis is the over-pushed video length in seconds. The dashed lines are the statistical results of *k-push* and *no-push*, which are considered as the upper bound and lower bound to the *adaptive-push*. The solid lines are used to plot the result of *adaptive-push* under different RTTs combined with the number of segments pushed capping policies. The *no-push* case uncovers that $\sim$ 45% videos over-push 1 second video to the client. The k-push case shows that 50% videos will over-push at least 12 second video content. In this figure, thinner lines are the results when RTT is low. In these cases, the server over-pushes at most 5 seconds video content since a smaller k is required to approach the real bandwidth in a low RTT environment. The thick solid lines plot the results when the RTT is as high as 500 ms. Even though the push number will finally reach the same value 13 as the *k-push* case, we can find that much less video content is over-pushed, which benefits from the small initial push number. There are only $\sim$ 14% video sessions that will over-push more than 12 second video content.

We vary the network bandwidth and the segment duration in the remaining simulations. The results are shown in Figures 9 and 10. From these figures, we observe that a shorter segment duration leads to less over-pushed content due to the finer chunking granularity. Different polices capping the push number also result in different over-pushed video length. It is reasonable that a more conservative decision leads to less over-pushed data. Such a trend is reflected in all the results.

6. CONCLUSION

HTTP streaming is widely used for delivering Internet video content today. Various mechanisms have been adopted in HTTP Streaming to improve the client experience. In this paper, we have evaluated the potential of the push mechanism introduced in HTTP/2

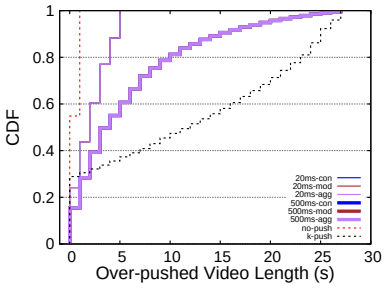

Figure 8: Cumulative Distribution Function (CDF) of over-pushed video length where the bandwidth is 200 kbps and the segment duration is 2 seconds

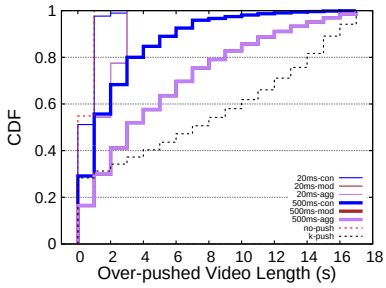

Figure 9: Cumulative Distribution Function (CDF) of over-pushed video length where the bandwidth is 540 kbps and the segment duration is 2 seconds

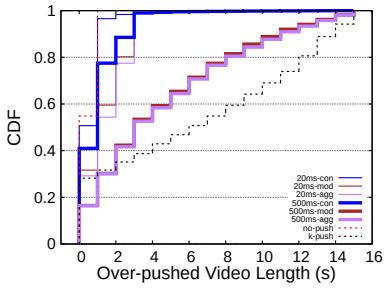

Figure 10: Cumulative Distribution Function (CDF) of over-pushed video length where the bandwidth is 880 kbps and the segment duration is 2 seconds

for video streaming. Motivated by our evaluation results, we have designed an *adaptive-push* scheme to improve the performance of HTTP Streaming. *Adaptive-push* is designed to exploit the maximum playback bandwidth by considering the playback bandwidth variation, the network adaptability, and the over-push problem. It makes a desirable trade-off between the *no-push* scheme and the intuitive *k-push* scheme, allowing a dynamic push technique to be integrated into current streaming delivery systems. We have evaluated the effectiveness of our *adaptive-push* scheme based on prototype implementation and simulations. The results confirm that the *adaptive-push* scheme can enhance the playback bandwidth while effectively alleviating the over-push problem.

7. ACKNOWLEDGMENT

We appreciate constructive comments from anonymous referees. The work is partially supported by NSF under grants CNS-1117300 and CNS-1524462.

8. REFERENCES

[1] A. Cardaci, L. Caviglione, A. Gotta, and N. Tonellotto. Performance evaluation of spdy over high latency satellite channel. In *Personal Satellite Services*, pages 123–134. Springer, 2013.

[2] W. Cherif, Y. Fablet, E. Nassor, J. Taquet, and Y. Fujimori. Dash fast start using http/2. In *Proceedings of the 25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 25–30. ACM, 2015.

[3] Cisco. Consumer Internet Traffic Report. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html.

[4] F. ECLIPSE. Jetty. http://www.eclipse.org/jetty/.

[5] J. Erman, V. Gopalakrishnan, R. Jana, and K. Ramakrishnan. Towards a spdy'ier mobile web? In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 303–314. ACM, 2013.

[6] M. B. et al. SPDY Protcol. https://tools.ietf.org/html/daft-ietf-httpbis-http2-00.

[7] D. I. FORUM. dash.js. https://github.com/Dash-Industry-Forum/dash.js/wiki.

[8] I. GOOGLE. YouTube. https://www.youtube.com/.

[9] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari. Confused, timid, and unstable: picking a video streaming rate is hard. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 225–238. ACM, 2012.

[10] R. Huysegems, T. Bostoen, P. Rondao Alface, J. van der Hooft, S. Petrangeli, T. Wauters, and F. De Turck. Http/2-based methods to improve the live experience of adaptive streaming. In *Proceedings of the 23rd Annual ACM Conference on Multimedia Conference*, pages 541–550. ACM, 2015.

[11] IETF. Hypertext Transfer Protocol Version 2 (HTTP/2). https://tools.ietf.org/html/rfc7540.

[12] A. Inc. HTTP Live streaming. http://developer.apple.com/resources/http-streaming.

[13] A. S. Inc. HTTP Dynamic Streaming on the Adobe Flash Platform. http://www.adobe.com/products/httpdynamicstreaming.

[14] N. Inc. Netflix. https://www.netflix.com/.

[15] J. Jiang, V. Sekar, and H. Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2012.

[16] R. K. Mok, X. Luo, E. W. Chan, and R. K. Chang. Qdash: a qoe-aware dash system. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 11–22. ACM, 2012.

[17] C. Mueller, S. Lederer, C. Timmerer, and H. Hellwagner. Dynamic adaptive streaming over http/2.0. In *Multimedia and Expo (ICME), 2013 IEEE International Conference on*, pages 1–6. IEEE, 2013.

[18] T. Stockhammer. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133–144. ACM, 2011.

[19] V. Swaminathan and S. Wei. Low latency live video streaming using http chunked encoding. In *Multimedia Signal Processing (MMSP), 2011 IEEE 13th International Workshop on*, pages 1–6. IEEE, 2011.

[20] Vuclip. Vuclip. http://www.vuclip.com/index.html.

[21] S. Wei and V. Swaminathan. Cost effective video streaming using server push over http 2.0. In *Multimedia Signal Processing (MMSP), 2014 IEEE 16th International Workshop on*, pages 1–5. IEEE, 2014.

[22] S. Wei and V. Swaminathan. Low latency live video streaming over http 2.0. In *Proceedings of Network and Operating System Support on Digital Audio and Video Workshop*, page 37. ACM, 2014.

[23] S. Wei, V. Swaminathan, and M. Xiao. Power efficient mobile video streaming using http/2 server push. In *Multimedia Signal Processing (MMSP), 2015 IEEE 17th International Workshop on*, pages 1–6. IEEE, 2015.