SABRE: A client based technique for mitigating the buffer bloat effect of adaptive video flows

Ahmed Mansy
Georgia Institute of Technology
amansy@cc.gatech.edu

Bill Ver Steeg
Cisco
versteb@cisco.com

Mostafa Ammar
Georgia Institute of Technology
ammar@cc.gatech.edu

ABSTRACT

HTTP adaptive video streaming is an emerging technology that aims to deliver video quality to clients in a manner that accommodates available bandwidth and its fluctuations. In this scheme, a video stream is split at the server into small video files encoded at multiple bitrates. The video is composed at the client by downloading these files over HTTP and TCP. Although there are some efforts to standardize media representation for this technology, adaptation techniques remain an open area for development. Recently, an alarm was raised by a study about the interaction between TCP congestion control algorithms and large buffers on the Internet. Queuing delays when these buffers are full can reach several hundreds of milliseconds in a phenomenon that was dubbed *buffer bloat*. In this paper we use measurements on a testbed to demonstrate and quantify the buffer bloat effect of HTTP adaptive streaming. We show that in a typical residential setting a single video stream can easily cause queuing delays up to one second and even more hence seriously degrading the performance of other applications sharing the home network. We develop SABRE (Smooth Adaptive Bit RatE), a scheme that can be implemented by the client to mitigate this problem. We implemented SABRE in the VLC player. Using our testbed, we show that our technique can reduce buffer occupancy and significantly diminish the buffer bloat effect without affecting the experience of the video viewer.

Categories and Subject Descriptors

H.5.1 [**Information Systems**]: Multimedia Information Systems - *Video(e.g., tape, disk, DVI)*

General Terms

Design, Performance, Algorithms

Keywords

DASH, buffer bloat, TCP, client based technique

MMSys'13, February 26-March 1, 2013, Oslo, Norway.

1. INTRODUCTION

Dynamic Adaptive Streaming over HTTP (DASH) [13] is a new important standard for video streaming on the Internet. In this approach, a video stream is split into small video segments of equal length and each segment is encoded in multiple bitrates. The video server is basically a web server that hosts video files representing these segments along with a manifest file that describes the segments and their bitrates. When starting a streaming session, a DASH video client first downloads the manifest file to learn about the available bitrates. After that, it downloads video segments usually starting with the lowest available bitrate. While downloading video segments, the client estimates the available bandwidth and adapts among the different video profiles accordingly. Each segment is specified using an HTTP *get* request and is downloaded over HTTP running on top of TCP.

A DASH player usually has a playout buffer and it always tries to keep it full. When a client starts a DASH streaming session, the player downloads video segments as fast as possible until it fills the buffer; this phase is called the *initial buffering phase*. After that, the client downloads a new video chunk every n seconds, where n is usually equal to the length of the video segment (n is usually in the range from 2 to 10 seconds). The client behaves this way in order to keep the video buffer full; this is called the *steady state phase*. The client behavior in this phase is called the *On/Off* behavior. This is because it downloads a video segment (*On*) then waits for a while (*Off*), and so on.

Excessive buffering of network devices on the Internet is a well known problem which has been studied in different contexts [5, 11]. This problem was reintroduced recently by Gettys and Nichols in [8] under the name *buffer bloat*. In that study, the authors collected evidence to show that the Internet can suffer from significant congestion due to the existence of large buffers at different network devices. As a result, traffic can experience very high queuing delays that can reach several hundreds of milliseconds and sometimes even more than a second. High delays can be very harmful to many applications on the Internet such as VoIP, interactive games, and e-commerce.

The root cause of the *buffer bloat* problem is the way TCP (Transmission Control Protocol) works. In order for TCP to achieve the best throughput, it keeps a send buffer of approximately the bandwidth delay product (BDP) of the path between the source and the destination. This means

that the maximum number of *bytes in flight* TCP can have is equal to the BDP. In addition, TCP uses packet losses to detect congestion. When TCP detects packet loss, it realizes that the path is congested and backs off to a lower transmission rate. While it is important for TCP to detect packet loss in a timely manner, large network buffers can store a large number of packets before loss can occur and hence loss detection is significantly delayed. This causes TCP to overestimate the BDP and consequently send larger bursts of data that fill the large buffers and cause high delays.

The steady state behavior of DASH can be simply described as a periodic download of small files (video segments) over HTTP. Since HTTP runs over TCP, we expect DASH video flows to have a *buffer bloat* effect. To the best of our knowledge, this effect has not been measured or quantified by any previous studies.

In this paper, we make two contributions. First, we show through a set of experiments in a testbed that a single DASH stream can cause significant delays to other ongoing applications sharing the home network in a typical residential setting. Our setting considers the common case when the bottleneck link is in the home access link and large tail-drop buffers exist in residential routers.

In order to mitigate this problem we present as a second contribution a technique, SABRE (Smooth Adaptive Bit RatE), that enables a video client to smoothly download video segments from the server while not causing significant delays to other traffic sharing the link. Our scheme is based on a simple and effective idea that can be implemented in the application layer of any DASH player. The idea uses a technique to dynamically adjust the flow control TCP window (*rwnd*) in a DASH client. By doing that, we manage to control the burst size going from the server to the client and effectively reduce the average queue size of the home router. We implemented SABRE in the VLC DASH plugin [10] and evaluated it using testbed experiments. Our results show that SABRE can significantly improve queuing delay over traditional On/Off video players.

The rest of the paper is organized as follows. In section 2 we introduce the experimental setup with results showing the buffer bloat effect of DASH video flows. In section 3 we show experimentally how Active Queue Management (AQM), often cited as a solution to buffer bloat problems, is not a workable solution in this context. Our SABRE technique is described in section 4 along with some evaluation results. We present some results when two clients share the same bottleneck link in section 5. We conclude the paper in section 6.

2. THE BUFFER BLOAT EFFECT OF ABR FLOWS

In order to measure the significance of the buffer bloat effect of DASH video flows, we set up a testbed in the lab that mimics real world secnarios. In this section we describe this testbed along with the results of some experiments. These results show that HTTP adaptive video flows induce significant queuing delays that can reach hundreds of milliseconds. All results throughout this paper were obtained using this testbed.

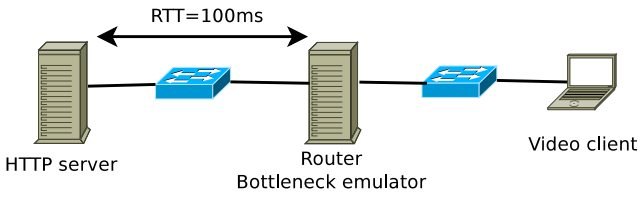


Figure 1: Experimental testbed

2.1 Experimental setup

Our testbed is shown in figure 1. The testbed consists of two workstations, two network switches, and a laptop. The workstations and the laptop are running operating system Ubuntu 12.04 LTS. The first workstation on the left acts as a remote HTTP video server and it runs a standard Apache webserver. The webserver hosts a video dataset that was obtained from a published DASH dataset [9]. The dataset includes a video of resolution 1280×1920 that is encoded into six different bitrates ranging from 2.04Mbps to 4.1 Mbps. The second worksation (in the middle) mimics a residential router that connects the client to the remote server. The laptop represents a video client that uses a VLC player equipped with a DASH plugin [10] to stream video from the server.

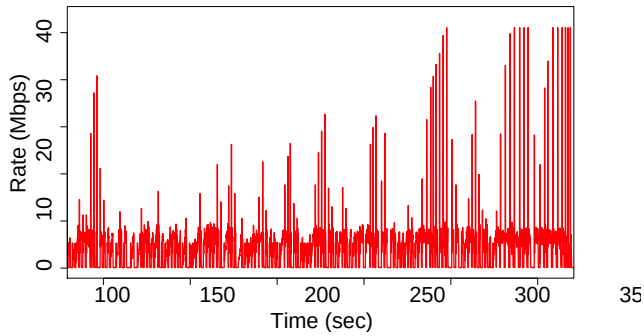
The linux traffic control tool *tc* is used on the router machine to emulate a bottleneck link between the client and the server. The bottleneck bandwidth is set to $6Mbps$, this is a common value in a residential DSL setting. The *tc* tool is also used to setup a tail-drop queue at the router of length 256 packets in order to emulate real residential gateways. The same tool will be used later to setup other Active Queue Management (AQM) techniques at the router. In addition, the *netem* tool is used to add a round trip time of $100ms$ between the router and the video server.

In this experiment we are interested in measuring the queuing delay experienced by VoIP traffic while a DASH streaming session is taking place. We emulate VoIP traffic by using *iperf* [1] to send UDP traffic from the video server to the client. We use UDP traffic of a constant bitrate of 80Kbps with small packets of 150 bytes each, this is similar to Skype voice traffic [6].

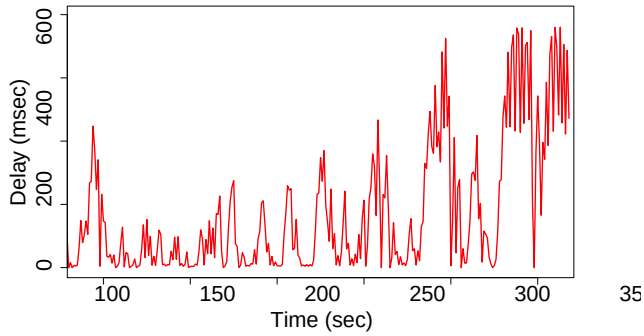
The one way queuing delay of UDP packets is measured in the following manner. *Wireshark* [3] is used to capture UDP traffic at both the router and the client. Assume $t_0^{(R)}$ and $t_i^{(R)}$ are the timestamps when the first and the i^{th} UDP packets were received at the router. Moreover, assume that $t_0^{(c)}$ and $t_i^{(c)}$ are the timestamps when the first and the i^{th} UDP packets were received at the client. Then the queuing delay of the i^{th} UDP packet can be computed using the formula

$$d(i) = (t_i^{(c)} - t_0^{(c)}) - (t_i^{(R)} - t_0^{(R)})$$

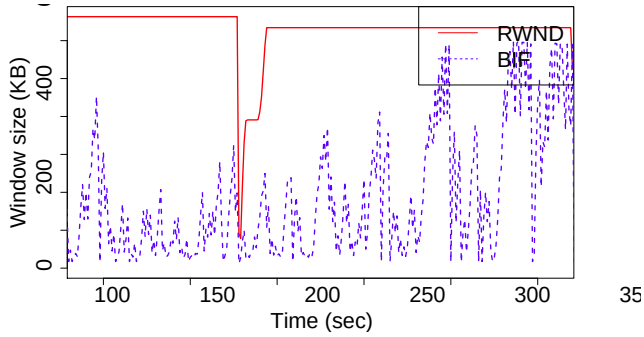
Note that computing the queuing delay in this way does not require synchronizing the clocks of both the client and the router machines.



(a) Data rate on the server link



(b) Queueing delay of UDP traffic



(c) Bytes-in-flight (BIF) and receiver window (*rwnd*)

Figure 2: On/Off video client with tail-drop queue at the router

2.2 Measuring buffer bloat

In this experiment we set the capacity of the bottleneck link to $6Mbps$. We first start UDP traffic then after five seconds we start streaming the video. In addition to computing queuing delay as described above, we use *Wireshark* [3] to capture incoming video traffic at the router. This enables us to compute the data rate on the link between the video server and the router, we call it the server link. In figures 2(b) and 2(a) we plot the queuing delay of UDP packets and data rate on the server link respectively. In these two figures we have a new sample every 100 milliseconds, meaning that we get 10 samples every second.

We can clearly see from the figures the correlation between the high data rate points in figure 2(a) and the high queuing delay points in figure 2(b). For example, during the time period from $t = 95$ to $t = 100$ we can see in figure 2(a) two large bursts of data with a rate that exceeds 30Mbps. During the same period we observe that queuing delay approaches $400ms$. The explanation is that the huge bursts of video data fill the queue at the router which causes UDP packets to experience long delays until the buffer gets drained. This behavior repeats multiple times at $t = 255, 290, 335$ and so on.

In order to understand why we see these large bursts of data we need to look at the receiver window (*rwnd*) returned by the client to the server, and the congestion window (*cwnd*) computed at the server. This is because the sender rate is governed by the value of $\min(rwnd, cwnd)$. We extract the *rwnd* values from the acknowledgement packets going from the client to the server. Since the actual value of *cwnd* can not be obtained without access to the server TCP code, we use the *bytes-in-flight* instead. The bytes-in-flight value is equal to the number of bytes that have been sent by the server but still awaiting acknowledgements from the client. This value is considered to be a lower bound on *cwnd*. Note that when bytes-in-flight equals *zero*, it does not necessarily mean that $cwnd = 0$, it could mean that no traffic was sent during the measurement period ($100ms$).

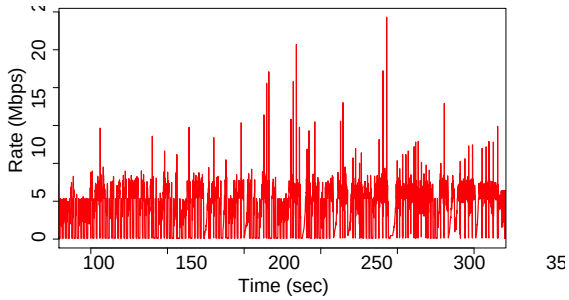
We plot both *rwnd* and bytes-in-flight (BIF) over time in figure 2(c). We observe that *rwnd* is almost a constant value of $650KB$ except at time $t = 180$ when it drops to *zero*. The reason it becomes *zero* is that the persistent TCP connection resets periodically around every three minutes. We suspect this is a default setting in the Apache web server as we observe this behavior repeatedly. We also observe that *rwnd* is always greater than the value of *bytes-in-flight*. This means that the burst size sent by the server is completely controlled by the value of the server congestion window (*cwnd*). On the other hand, the *bytes-in-flight* value varies widely over time and, as expected, the high data rate bursts in figure 2(a) correspond to high values of *bytes-in-flight* in figure 2(c).

Another observation here is that although the VLC player uses persistent TCP connections, the *cwnd* value does not grow continuously from video segment to the next as one may expect. The reason for that is the On/Off behavior of the video player. During the off period, the TCP connection becomes idle until it starts downloading the next video segment. If this off period is longer than the retransmission timeout (RTO), then according to TCP congestion control specification [2], *cwnd* gets reset to the value of the initial window (IW). The initial window is typically two TCP segments which is between 1000 and 3000 bytes.

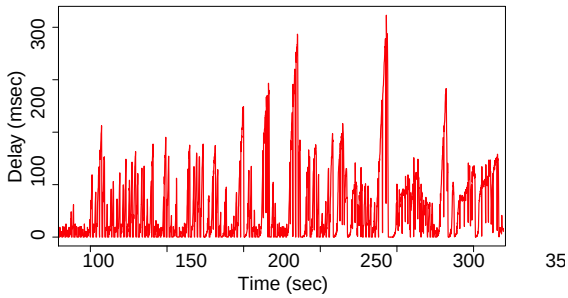
3. RANDOM EARLY DETECTION (RED)

Before presenting our client based technique, we first consider an Active Queue Management technique, specifically RED. RED is a technique that looked like it might be able to solve the buffer bloat problem but has proven to be difficult to manage and tune.

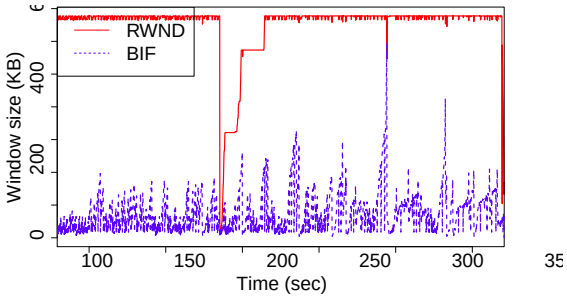
RED [7] computes an average queue size using a weighted moving average. In addition, RED is configured with two parameters, *minimum* and *maximum*. When the average queue size is less than *minimum*, no packets are marked to get discarded. When the average queue size is larger than *maximum*, all incoming packets are marked to get dropped.



(a) Data rate on the server link



(b) Queuing delay at the router



(c) Receiver window *rwnd*, and bytes-in-flight (BIF)

Figure 3: On/Off video client with RED queue at the router

When the queue size is between *minimum* and *maximum*, the probability of discarding a packet increases linearly with the queue size. Using this policy, RED guarantees that the queue size does not grow much over *maximum*.

We repeated the same experiment in section 2 after using the *tc* tool to replace the tail-drop queue with a RED queue. We set the max queue size to $200KB$ with the *minimum* and *maximum* as 20% and 80% of that value respectively. In figures 3(a) and 3(b) we plot the data rate on the server link and the queuing delay of UDP packets respectively.

Although we observe from figure 3(b) that RED manages to reduce queuing delay compared to the tail-drop queue (figure 2(b)), queuing delay still can be above $150ms$ for a large number of measurement intervals. This is still too much to be acceptable for a VoIP call. The reason queuing delay reach hundreds of milliseconds even when RED is used in the router, is that RED does not prevent the server from sending large bursts of data. This can clearly be seen from the data rate on the server link in figure 3(a). As explained in section 2, the reason for these large bursts is that *rwnd* usually stays at very large values and *cwnd* occasionally grows to large values as well, which results in sending large bursts of data. This can be seen from figure 3(c).

It is worth mentioning here that there could be another configuration for RED with different parameters that could produce better results. However, this is one of the main disadvantages of using RED. Tuning the algorithm to get the best performance is not an easy job [12]. Moreover, finding the set of parameters that would optimize the performance for one type of data flow does not mean that these parameters would work for all other applications. This is why we believe that the solution to the high queuing delay problem is by stopping the server from sending large bursts of data over short periods of time. That is because once the burst is on the wire, there is not much that can be done to prevent the queue from getting full. In the next section we present our solution to this problem.

4. SABRE

As mentioned earlier, the maximum burst TCP can send at any point is equal to $\min(cwnd, rwnd)$. Hence, one way to control the size of the burst is to control either *cwnd* or *rwnd*, or both. We know that a TCP sender uses ACKs and packet loss to increase and decrease the value of *cwnd* respectively. AQM algorithms introduce packet loss in the middle boxes in order to limit the growth of *cwnd*. On the other hand, it is very difficult for the client to introduce packet loss from the application space.

Alternatively, there are multiple ways the client can control the value of *rwnd*. It is important to mention here that the value of *rwnd* is a function of the empty space at the receive socket buffer at any point in time. This means that the size of the socket buffer represents an upper bound on the value of *rwnd*. Hence, one way to control the maximum value of *rwnd* is to set the value of the receive buffer at the client. This can be done using the *setsockopt* call with the option `SO_RCVBUF`. However, the DASH player may not be privileged to make the *setsockopt* call. This usually happens when the DASH player is implemented as a plugin to an existing video player which is the case for the VLC DASH player [10]. Another issue with this approach is that *setsockopt* can be used to set the buffer size only before establishing the connection. This means that a solution that dynamically uses *setsockopt* to set the buffer size will have to reset the connection everytime it needs to modify the buffer size. Resetting TCP connections frequently has many dis-

advantages. To name a few; it makes tracing network flows more difficult, it can be an overhead on the server, and it may require new key exchange if the flow is encrypted.

Another method to set the size of the receiver socket buffer is to use the Linux command *sysctl* to set the system parameter `net.ipv4.tcp_rmem`. However, this method has a system wide effect and it sets the maximum socket buffer for all TCP connections on the client machine which is undesirable.

Below we present SABRE, our technique to control the burst size from the application layer. First, in section 4.1 we introduce the technique for the unconstrained bandwidth case when the available bandwidth is constant and higher than the highest video bitrate. After that, in section 4.2 we develop the full-fledged scheme to work for the general case when there is variability in the available bandwidth.

4.1 The unconstrained constant bandwidth case

SABRE relies on three key techniques in its operation; HTTP pipelining, controlling download rate at the application, and a dual *backoff/refill* mode of operation. Below we describe in detail each of these techniques and the operation of SABRE in steady state.

HTTP pipelining. In steady state, an on/off video client requests a new video segment every n seconds where n is the segment length. Usually, the client finishes downloading a segment before submitting a request to download the next one. Due to that behavior, the receive socket buffer is always empty when the client starts downloading a new segment. In addition, *rwnd* is typically computed as a function of the available space in the receiver socket buffer. Although the exact function varies among different implementations of TCP, an empty or almost empty receiver socket buffer will always result in a large value of *rwnd*. This will usually cause the large data bursts we saw in section 2. In order to mitigate this problem we pipeline requests for multiple video segments. HTTP pipelining allows the client to send multiple `GET` requests to the server before having to wait for them to finish.

The rationale here is that pipelining enough video segments guarantees that the server will send enough data to always keep the client receive buffer full. We dynamically compute the number of segments to pipeline as follows: using the *getsockopt* with option `SO_RCVBUF`, we get the actual size of the receive buffer, call it $rcvbuf$ bytes. For a video segment of length s seconds and bitrate r bps, the average segment size will be $\frac{rs}{8}$ bytes. Hence the number of HTTP requests the client should pipeline is $1 + ceil(rcvbuf * \frac{8}{rs})$. In this formula, the second term is the number of segments to fill the socket buffer and the additional segment is the one being read by the video player.

Controlling download rate at the application. Filling the socket buffer does not guarantee avoiding large data bursts all the time. If the application reads from the socket buffer at a high rate, this will drain the buffer quickly which will cause *rwnd* to grow to a large value. If we take into consideration large round-trip times between the client and the server, *rwnd* could potentially grow to very large values. In order to solve this problem, the application has to control the rate in which it drains the socket buffer.

We know that the socket buffer gets drained by the *recv* API call. This means that every time *recv* is called, part of the socket buffer gets cleared and hence the value of *rwnd* may increase. As a result, controlling the rate in which the application calls *recv* will control the rate at which the socket buffer is drained and in turn will control the growth rate of *rwnd*.

As one may expect, traditional On/Off DASH players (including the VLC DASH player [10]) call *recv* as fast as possible. This causes the socket buffer to get drained as soon as any data arrives at the client. This in turn causes the value of *rwnd* to be always very large as we observed in figures 2(c) and 3(c). On the other hand, it is important to observe that the video player does not need to read data from the socket buffer at a rate higher than the video bitrate itself. This is why we compute the rate of the *recv* call so that the achieved download rate at the client at any point in time is not much higher than the video bitrate streamed by the client. We call this download rate the $target\_rate$.

In order to prevent *rwnd* from growing large, we distribute the *recv* calls uniformly over the segment download time where the latter is the same as the segment length in seconds. For example, if the $target\_rate$ is r bps, the segment length is s seconds, and the size of the buffer given to the *recv* call is buf bits, then the time between consecutive *recv* calls can be computed using the formula $t = \frac{s}{(rs)/(buf)} = \frac{buf}{r}$ seconds. This maintains a steady download rate close to r bps for a period of s seconds, while at the same time controlling the growth rate of *rwnd*.

Backoff/refill mode of operation. Remember, however, that a DASH video player estimates the available bandwidth on the path between the client and the server while downloading video segments. The client then uses the estimated bandwidth to decide whether it should switch to a higher or lower video quality or stay at the same video profile. Controlling the rate of the *recv* calls affects the estimated bandwidth by the client. In fact, if the rate of *recv* calls was computed to achieve a $target\_rate$ of r Mbps, we do not expect the client to estimate the available bandwidth to be higher than r Mbps. This means that the client will not be able to switch to higher video bitrates even if there exists enough bandwidth in the path between the server and the client.

In order to solve this problem, we modify the video player to operate in two modes; a *refill* mode and a *backoff* mode. The player enters the *refill* mode when its playout buffer level drops below a threshold value *refill_thresh*. In that mode the player targets a $target\_rate$ of $\lambda \times R_h$ where $\lambda > 1$ and R_h is the bitrate of the best video profile. The player does not need to target a higher download rate because its ultimate goal is to reach the best video profile. At the same time, targeting a lower download rate may cause the player to underestimate the available bandwidth and hence not reach the best video profile.

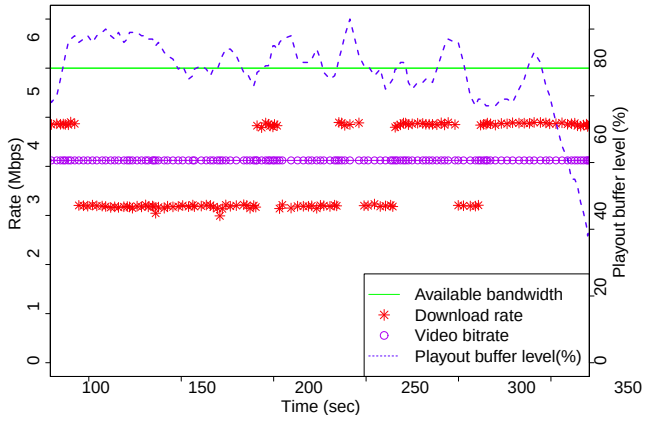


Figure 4: SABRE player: download rate, video bitrate, and level of playout buffer at a constant available bandwidth of 6Mbps

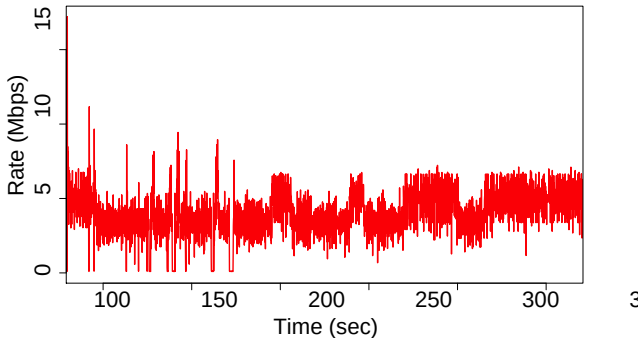
Once the level of the playout buffer exceeds another threshold *backoff_thresh*, the player enters the *backoff* mode. In this mode, the player aims at a *target_rate* of $\delta \times R$ where $0 < \delta < 1$ and R is the bitrate of the current video profile. The reason for the player to choose a download rate that is less than the video bitrate is to prevent over filling the playout buffer. The player stays in the *backoff* mode until the playout buffer gets to the *refill_thresh* and then it enters the *refill* mode again.

4.1.1 Experimental results

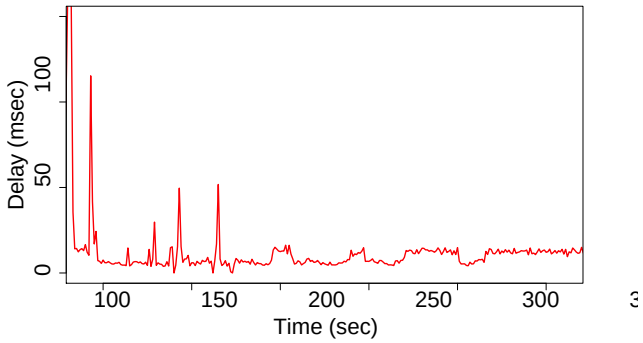
We implemented the above technique in the VLC DASH player [10]. In our implementation we set $\lambda = 1.2$ and $\delta = 0.8$. We repeated the experiment we did in section 2 while setting the bandwidth of the bottleneck link to 6 Mbps and we stream the video for 6 minutes. We consider the results of the steady state behavior starting from $t = 90$ seconds. Figure 4 shows the computed download rate, the requested video bitrate, and the level of the playout buffer. We can clearly see that the player is switching back and forth between the *backoff* and *refill* modes. For example, at time $t = 95$ the client enters the *backoff* mode and sets its *target_rate* to $0.8 \times 4.1 = 3.3$ Mbps. The client stays in this mode until $t = 190$, when the playout buffer drops below 85% then it enters the *refill* mode and sets it *target_rate* to $1.2 \times 4.1 = 4.92$ Mbps.

In figure 5 we plot the data rate on the server link, queuing delay, and the values of *rwnd* and bytes-in-flight over time. Compared to figure 2, we can see from figure 5(a) that bursts of high data rates do not exist anymore and this is why we do not see large queuing delays in figure 5(b). Looking at figure 5(c), we can clearly see that the value of bytes-in-flight is constantly being pushed down by the value of *rwnd*. We can also see that the value of *rwnd* is now close to 50KB most of the time. This is a huge reduction compared to figure 2(c) when *rwnd* used to be over 600KB. This is because HTTP pipelining is keeping the socket buffer full all the time which causes the *rwnd* to shrink to this low value.

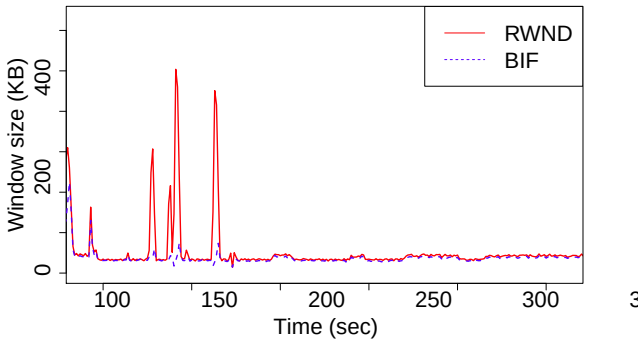
We can also observe that sometimes *rwnd* grows to large values for short periods of time, this can be seen in figure 5(c)



(a) Data rate on the server link



(b) Queuing delay



(c) Receiver window and bytes-in-flight

Figure 5: SABRE video client with tail-drop queue at the router

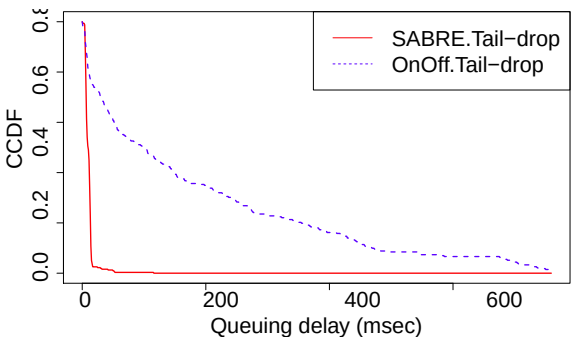


Figure 6: CCDF of queuing delay, On/Off player vs SABRE both using a tail-drop queue

between times $t = 130$ and $t = 170$. As mentioned earlier, the exact implementation for computing *rwnd* is operating system dependent and it is not clear to us why we observe such behavior. This transient behavior could possibly result in receiving large bursts of data which could increase queuing delay. In section 4.2 we present a technique to mitigate this problem and to avoid having large queuing delays even if the socket buffer drops for several seconds.

In order to compare the overall performance of the On/Off player versus SABRE in the unconstrained bandwidth case, we plot the CCDF of the queuing delay for both of them in figure 6. In this figure we see that SABRE manages to keep queuing delay below $50ms$ for almost 100% of the time. The On/Off player, on the other hand, causes queuing delay to exceed $100ms$ for about 50% of the time with 15% of the time being over $400ms$.

4.2 The general case

In real network conditions the available bandwidth changes frequently over time. In this section we describe the operation of the full-fledged SABRE scheme which uses the techniques described above in addition to adapting to changes in network conditions while maintaining low queuing delays.

Upon starting the video stream, the player enters the initial buffering phase. In this phase the player downloads video segments as fast as possible until it fills a playout buffer of 60 seconds. Once the buffer is full, the player enters the steady state phase, and this is when SABRE starts its operation. In both phases, the client computes the download rate of each video segment after it finishes downloading it. The is done by dividing the size of the segment (in bits) by the time it took the client to download it (in seconds). The client then computes a moving average of these values to estimate the available bandwidth.

If $D(i)$ is the download rate of segment i then the available bandwidth BW can be estimated using the formula

$$BW = \alpha BW + (1 - \alpha)D(i)$$

where $0 < \alpha < 1$ is a smoothing parameter. In our implementation we set $\alpha = 0.8$. Similar to [4], the player then uses BW to decide whether it should switch to a different video profile or stay at the current one. If R_i is the bitrate of the current segment and $BW < kR_i$, then the client switches to R_{i-1}, the next lower video profile. We use $k = 1.1$ as a slack parameter to compensate for variability in the computed download rate. If R_{i+1} is the bitrate of the next higher video profile and $BW > kR_{i+1}$ then the player switches to R_{i+1}. The rest of this section describes the operation of SABRE in steady state.

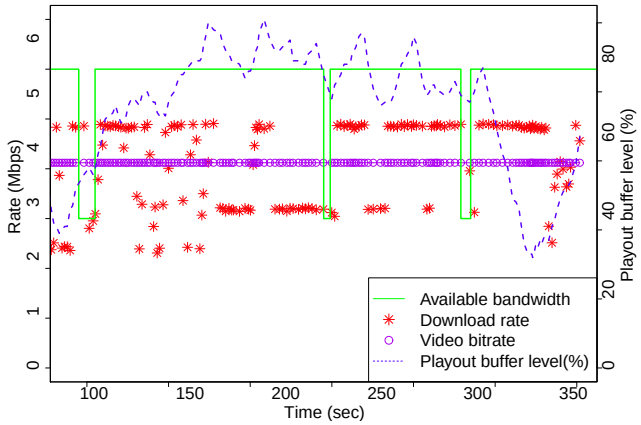
As described in section 4.1, SABRE pipelines video segment requests to keep the socket buffer full all the time. This means that the client is guaranteed to achieve its objective *target_rate* given that the available bandwidth is higher than that rate. When the network gets congested and the available bandwidth becomes less than *target_rate*, the socket buffer will get drained and the received download rate will be significantly less than *target_rate*. We identify a congestion event based on the condition $BW < \gamma \times target\_rate$, where $0 < \gamma < 1$ – we use $\gamma = 0.85$. When the socket buffer gets drained due to a congestion event, *rwnd* will increase rapidly which will cause queuing delay to increase significantly. This increment in delay reaches several hundreds of milliseconds and can last for several seconds. Once SABRE detects the congestion event, it can reduce its *target_rate* which will result in reducing *rwnd* and hence reduce the queue size. Due to space limitation we do not show results for this case.

In order to avoid these large delay spikes we need to be constantly monitoring the socket buffer occupancy level and act quickly when sudden changes happen. This can be achieved using the *ioctl* call with the `FIONREAD` option. This call returns the number of bytes that can be read at the socket buffer. Using this number together with the total socket buffer size (we get the latter from the *getsockopt* call), we can periodically compute the occupied part of the socket buffer. From our experiments, keeping a socket buffer occupancy level of 75% or more will guarantee to have small queuing delays.

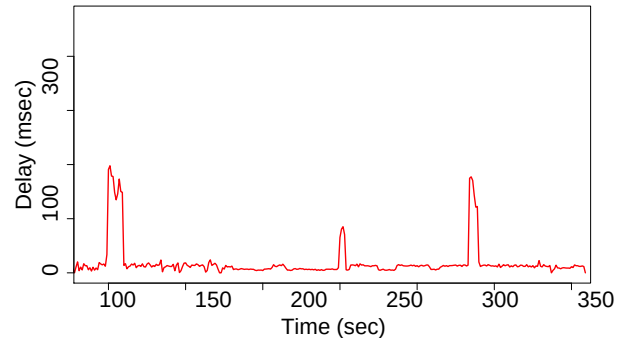
We compute the occupancy level of the socket buffer every $200ms$. This helps us to react fast when sudden variations happen in the available bandwidth. When the buffer level is detected to be lower than 75%, we temporarily reduce the rate of the *recv* call. Reducing the rate of the *recv* call allows the socket buffer to refill quickly until it gets back to the normal level. Once the socket buffer level gets to 75%, the rate of *recv* is resumed to its original value. Note that reducing the rate of *recv* calls can result in reducing the video segment download rate $D(i)$ computed by the application. This is because having a lower rate of *recv* calls means that the client will need more time to finish downloading the segment. Depending on how long SABRE has to reduce the rate of *recv*, $D(i)$ can vary from the *target_rate* (which can be $1.2R$ in the refill mode, where R is the video bitrate) to θR, where θ is the drop rate of *recv* – we use $\theta = 0.5$.

Drops in the socket buffer level can happen due to two different events; random drops like the ones observed in section 4.1.1 and drops due to changes in the available bandwidth. Although SABRE should not react to the first kind, it should reduce the requested video bitrate in the second. SABRE distingueshes between these two kinds of events using the following method. A drop event is considered significant only if it results in a segment download rate $D(i)$ that is less the video bitrate R. If SABRE detects consecutive significant events for a certain period of time, this event is considered a change in the available bandwidth, otherwise it is considered a random drop in socket buffer. In our implementation we set this period to 10 seconds.

When SABRE detects a drop in the available bandwidth it down-shifts to a lower video profile. However, since SABRE always keeps the socket buffer full, it can not estimate the new available bandwidth. Instead, SABRE uses a multiplicative-decrease additive-increase approach to find the best video profile that fits the new available bandwidth. Let R_i be the video bitrate at the time when SABRE decided to down-shift and R_0 be the lowest video profile. Then, when a drop in the available bandwidth is detected, SABRE will follow a multiplicative-decrease behavior and down-shift to the video profile $R_{i/2}$. SABRE then waits for a stabilization period

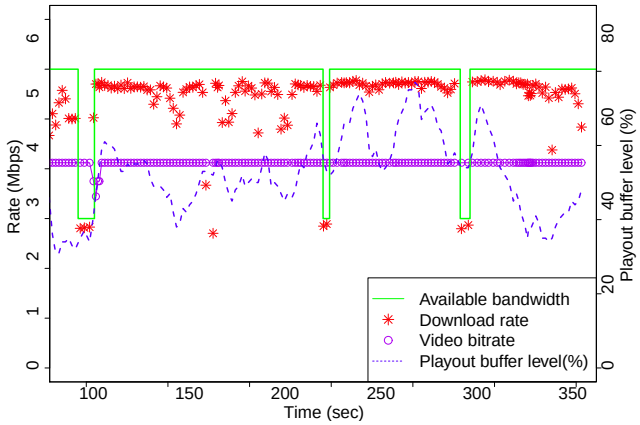


(a) Player adaptation to change in the available bandwidth

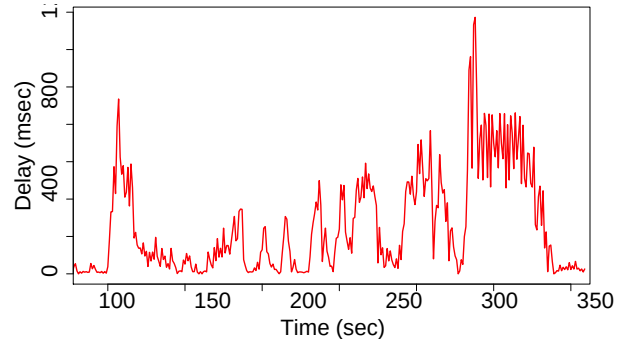


(b) Queuing delay

Figure 7: SABRE player with tail-drop queue and short duration congestion



(a) Player adaptation to change in the available bandwidth



(b) Queuing delay

Figure 8: On/Off player with tail-drop queue and short duration congestion

of *wait_to_probe* seconds to see whether or not it needs to down-shift to a lower profile. If SABRE does not detect any further drops in the available bandwidth, it starts probing to see whether it can achieve a higher video profile or not.

SABRE then follows an additive-increase behavior and up-shifts to the next video profile $R_{1+i/2}$. Using the same previous method, SABRE can detect whether or not the available bandwidth is enough to support the new video profile. Note here that detecting a drop in the available bandwidth while streaming a certain video profile is equivalent to detecting an up-shift to a profile that is higher than the available bandwidth. If SABRE manages to up-shift to the higher profile without detecting a drop in the available bandwidth, it reduces the value of *wait_to_probe* to half. The idea here is that the client should probe more aggressively when there is a lower chance of congestion. On the other hand, if SABRE detects that it has to down-shift again to a lower profile, it doubles the value of *wait_to_probe*. In our implementation, we set the default value of *wait_to_probe* to 16 seconds and allow it to reach a maximum of 32 seconds and a minimum of 4 seconds.

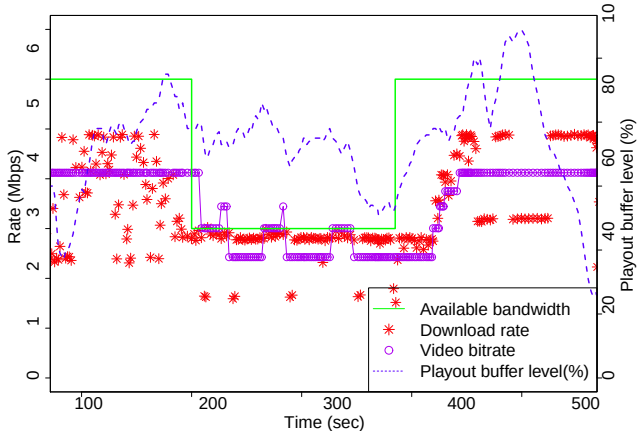
4.2.1 Experimental results

In this section we present results for two cases when the bottleneck link is congested. In the first case, the client experiences short-lived variations in the available bandwidth for only a few seconds. Our objective here is to study the effect of short variations on both the queuing delay and the adaptation logic. In the second case, the available bandwidth changes over longer periods of time in the order of several minutes.

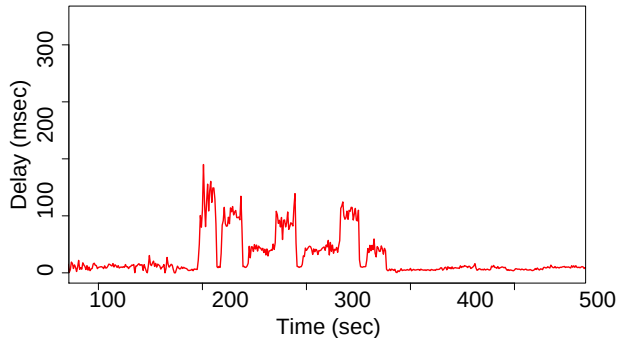
Short-lived variations

We repeated the experiment in section 4.1.1 with the following changes. We start the experiment with the available bandwidth set to 6 Mbps. The bandwidth of the bottleneck link is set to 3 Mbps at times $t = 95, 245, 330$ for the duration of $10, 4, 6$ seconds respectively. We performed this experiment twice; once using a SABRE client and the other using an On/Off client. Since we are interested in the steady state phase, we do not show results for the initial buffering phase. In figures 7 and 8 we show results for SABRE and On/Off clients respectively.

We can see from figure 7(a) that SABRE treated the three short variations in the available bandwidth as random drop events and it did not switch to a lower video profile. The behavior of the On/Off client was similar in figure 8(a) although it switched to lower profiles at time $t = 100$ when the drop in the available bandwidth lasted for 10 seconds. Since the On/Off client uses a moving average to estimate the available bandwidth, it needs to download multiple video



(a) Player adaptation to change in the available bandwidth



(b) Queuing delay

Figure 9: SABRE player with tail-drop queue and long duration congestion

segments with the new available bandwidth before it can act accordingly.

In figure 7(b) we plot queuing delay over time for the SABRE client. We can see that delay is always below $20ms$ except for the three events when the available bandwidth drops to 3 Mbps. In this case, reducing the rate of the *recv* calls prevents large delay spikes from happening, although delay still increased to about $200ms$. This happens in the three events and the lowest effect was at time $t = 245$ when the drop event lasted for only 4 seconds. On the other hand, we can see queuing delay for the On/Off client in figure 8(b). It is clear that droping the available bandwidth causes significant increase in delay. This can be seen at times $t = 100, 335$ when queuing delay jumps to $900ms$ and $1300ms$ respectively.

Long-lived variations

For this case, we repeated the experiment in section 4.1.1 with the following changes. We start the experiment with the available bandwidth set to 6 Mbps from time $t = 0$ to $t = 190sec$. After that, we change the available bandwidth to 3 Mbps from $t = 190$ to $t = 380sec$, then we set it back to 6 Mbps until the end of the experiment. We performed this experiment for both the SABRE client and the On/Off client. We show results for the SABRE and On/Off clients in figures 9 and 10 respectively.

In figure 9(a) we can see that SABRE takes less that 10 seconds to detect a change in the available bandwidth at time $t = 200$. Once change is detected, SABRE applies the multiplicative-decrease policy and down-shifts from $R_5 = 4.1$ Mbps to $R_2 = 3.1$ Mbps. Since the new available bandwidth is 3 Mbps which is very close to R_2, SABRE thinks it can up-shift to a higher profile. At time $t = 220$ SABRE decides to up-shift to the next profile $R_3 = 3.4$ Mbps. SABRE then detects that the available bandwidth is not enough for the new profile and down-shifts to $R_1 = 2.45$ Mbps at time $t = 230$. After a stabilization period of $wait\_to\_probe = 32$ seconds, SABRE up-shifts to $R_2 = 3.1$ Mbps and then up-shifts again to $R_3 = 3.4$ Mbps. This behavior repeats until the available bandwidth increases to 6 Mbps at time $t = 380$. Since SABRE has to wait for $wait\_to\_probe$ every time before up-shifting to a higher profile, SABRE takes additional time to recover to the highest video profile $R_5 = 4.1$ Mbps. In this case SABRE reached R_5 at time $t = 440$ which means it took an additional 60 seconds after the available bandwidth changed to 6 Mbps.

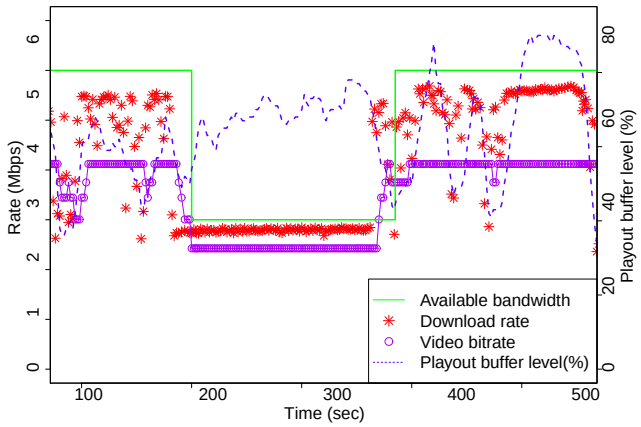
On the other hand, it took the On/Off player only about 10 seconds after the available bandwidth became 6 Mbps to recover to the highest profile R_5 (see figure 10(a)). However, this comes with a very expensive price in terms of queuing delay. We can see from figure 10(b) that during the congestion period from $t = 190$ to $t = 380$ queuing delay jumps dramatically over one second while in figure 9(b) SABRE manages to keep the maximum delay below $200ms$ and about $100ms$ on average.

We also plot the CCDF of queuing delay for both SABRE and the On/Off player in figure 11. It is clear that while SABRE manages to keep queuing delay less that $100ms$ for about 90% of the time, an On/Off player causes queuing delay to exceed $200ms$ about 60% of the time. This delay can be very disturbing to other applications sharing the same bottleneck link with the video flow.

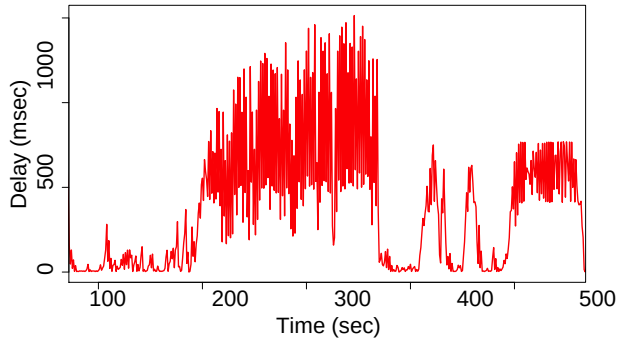
5. TWO VIDEO PLAYERS

In this section we present some experimental results when two clients share the same bottleneck link to the video server. This typically happens when two persons at the same home stream different videos. In this study we are interested in two things; the buffer bloat effect of multiple adaptive video flows, and the interaction between the adaptation algorithms in the two clients. Below we present results for three cases: two On/Off clients, one On/Off and one SABRE, and two SABRE clients.

In all three experiments we first start one of the two clients, wait for 20 seconds, and then start the second one. Both clients run for 300 seconds. Again, we ignore the first 100 seconds of the experiment since we are only interested in the steady state behavior of the two clients. The bandwidth of the bottleneck link is set to 6 Mbps. Ideally, each client should converge to a video profile that occupies its own fair share of the bandwidth. Since the fair share is 3 Mbps, the ideal video bitrate for each of the two clients should be 2.45 Mbps. Below we present the observed results for all three experiments.



(a) Player adaptation to change in the available bandwidth



(b) Queuing delay

Figure 10: On/Off player with tail-drop queue and long duration congestion

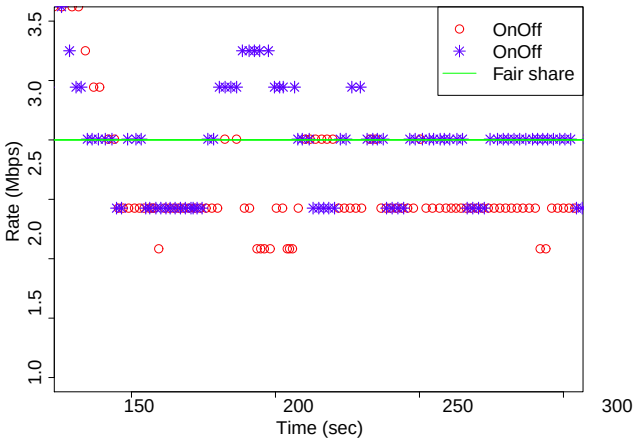


Figure 12: Bitrate adaptation when two On/Off players share a bottleneck link of 6 Mbps

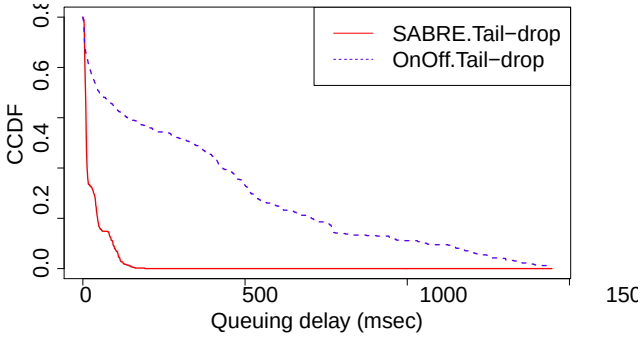


Figure 11: CCDF of queuing delay, On/Off player vs SABRE both using a tail-drop queue and long duration congestion

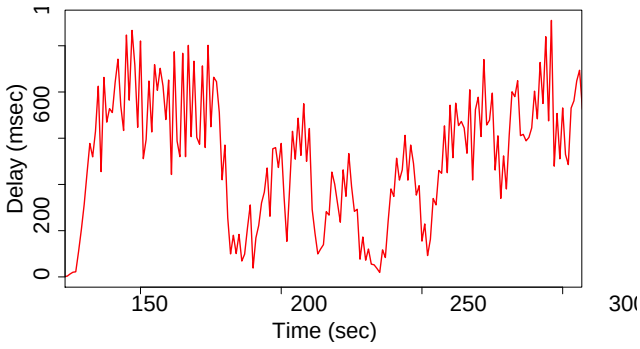


Figure 13: Queuing delay when two On/Off players share a bottleneck link of 6 Mbps

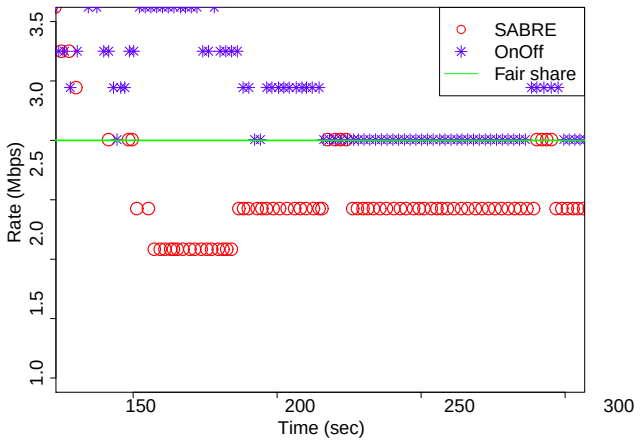


Figure 14: Bitrate adaptation when two players, On/Off and SABRE, share a bottleneck link of 6 Mbps

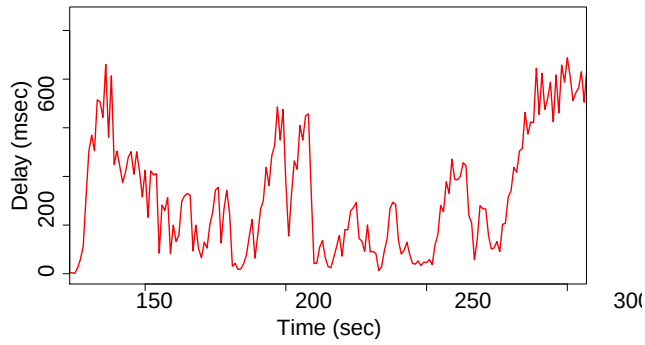


Figure 15: Queuing delay when two players, On/Off and SABRE, share a bottleneck link of 6 Mbps

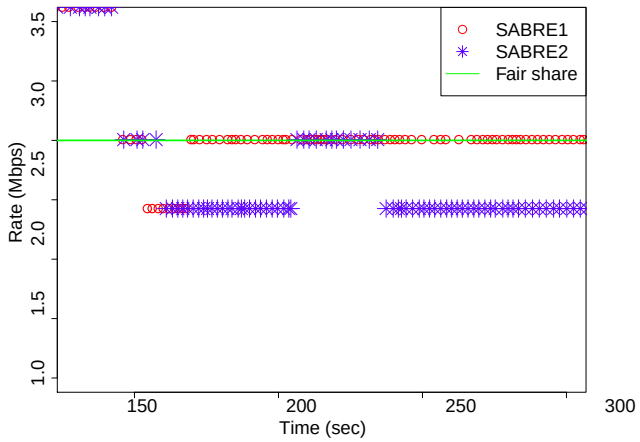


Figure 16: Bitrate adaptation when two SABRE players share a bottleneck link of 6 Mbps

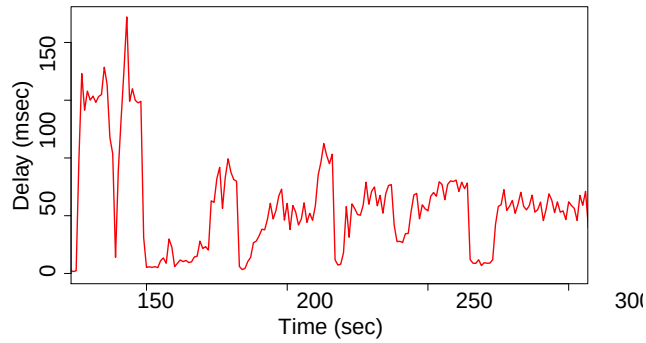


Figure 17: Queuing delay when two SABRE players share a bottleneck link of 6 Mbps

Two On/Off clients. Figure 12 shows the requested video bitrates by the two clients over a period of three minutes. We can observe that sometimes a client can overestimate the available bandwidth. This in turn can lead the client to request a video bitrate that is higher than its fair share. This can be seen at times $t = 180$ and $t = 230$ when one of the two clients requests a bitrate of 3.4 Mbps.

This behavior is similar to what was observed in [4]. The reason behind this behavior is the *off* periods of On/Off players. During an `Off` period, the `On` client can overestimate the available bandwidth because it thinks it is the only one using the link. In figure 13 we plot the queuing delay caused by the two On/Off video flows. We can see that adding two flows produces much higher delays than what was observed with a single flow in figure 2(b). The delay sometimes reaches one second and is over $500ms$ for about 50% of the time. Having a one way queuing delay close to a second makes it almost impossible to use the bottleneck link for anything else.

An On/Off client and a SABRE client. The interaction between an On/Off client and a SABRE client sharing a bottleneck link is very important to study. The On/Off client always probes the link aggressively for the available bandwidth. On the other hand, SABRE follows a much less aggressive behavior. It is important to study and understand whether one of them can cause performance degradations to the other. In figure 14 we plot the requested video bitrates by both players over time.

We can see that after the SABRE client detects a congestion event it manages to stabilize at a video bitrate of 2.45 Mbps. The client then periodically tries to shift to the next video bitrate (3.1 Mbps) but it always fails and goes back to the previous bitrate (2.45 Mbps). The On/Off client, on the other hand, settles at a video bitrate of 3.1 Mbps. It occasionally over-estimates its share of the bandwidth and shifts to higher bitrate. However, this shift does not last for a long period and the client falls back to 3.1 Mbps. It is clear that the On/Off client abuses the conservative behavior of the SABRE client and settles for a video bitrate higher than the one used by the SABRE client.

In figure 15 we plot the queuing delay caused by the two video flows. We can make two observations from this figure. First, although the delay is high and sometimes reaches $800ms$, the combination of a SABRE player and an On/Off player achieves a lower queuing delay than having two On/Off clients. Second, a single On/Off player can cause high queu-

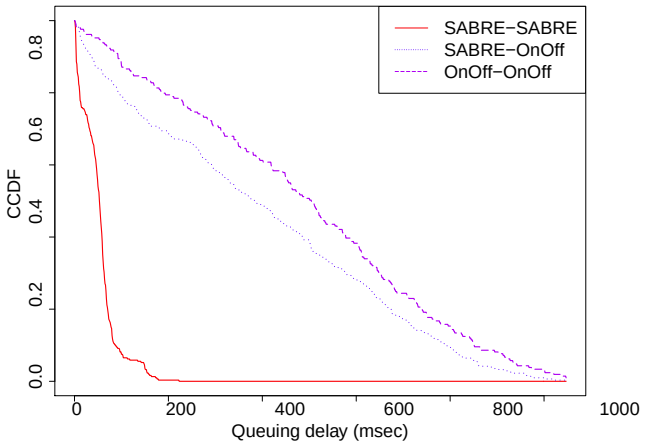


Figure 18: CCDF of queuing delay for two clients sharing a bottleneck link of 6 Mbps. Three cases: two On/Off clients, one On/Off and one SABRE, and two SABRE clients

ing delays even if SABRE players are sharing the bottleneck bandwidth with it.

Two SABRE clients. We can see from figure 16 that the two SABRE clients are competing over the video profile $R_2 = 3.1$ Mbps. Since the bottleneck link has a bandwidth of 6 Mbps, it can not accomodate two video flows of 3.1 Mbps each. At time $t = 145$, both clients converged to R_2 but not for a long time. They both detected that the available bandwidth is not enough to accomodate their video profiles and they both down-shifted to $R_1 = 2.45$ Mbps. Later, one of the two clients up-shifts to R_2 and manages to stay there, while the other client periodically up-shifts then down-shifts again. In our experiment, the client that managed to stay at the higher video profile (R_2) had more memory and processing power than the other client. We suspect this is an important factor in deciding which of the two clients will win with the higher video profile.

In order to characterize the buffer bloat effect in the three experiments, we plot the CCDF of queuing delay for the three experiments in figure 18. We can clearly see that two SABRE players are much better for the network than having even a single On/Off player. More specifically, the SABRE players manage to keep queuing delay below $100ms$ for about 95% of the time. On the other hand, whenever one On/Off player gets in the way we get queuing delay over $200ms$ for about 70% of the time.

6. CONCLUDING REMARKS

HTTP adaptive video streaming is being adopted by major content providers as the standard for streaming video on the Internet. With the wide spread use of this technology among residential users, it is important to make sure that it does not affect their Internet experience in a negative way. Recent studies show that Internet users can suffer from buffer bloat due the interaction between TCP and large buffers on the Internet. These studies raise the question whether HTTP streaming could have such a harmful effect on residential Internet users.

In this paper we use testbed measurements to show that, indeed, HTTP adaptive video streaming can be harmful to other applications sharing the same residential network. Our results show that even a single video stream can cause up to one second of queuing delay and it even gets worse when the home link is congested. We also show that AQM techniques, a widely believed solution to this problem, do not manage to eliminate large queuing delays.

In addition, we introduce SABRE, a client based technique that can be implemented in the video player to mitigate this problem. We implemented SABRE in the VLC DASH player. Using testbed experiments, we show that SABRE manages to significantly reduce queuing delays while not affecting the user viewing experience. We also conduct experiments with two clients sharing the home network to study the interaction between SABRE and other traditional players. Our results show that SABRE can coexist with traditional streaming players without having any performance penalties.

7. REFERENCES

[1] Iperf. `http://iperf.sourceforge.net`.
[2] TCP Congestion Control. `http://tools.ietf.org/html/rfc5681`.
[3] Wireshark. `http://www.wireshark.org`.
[4] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. Begen. What happens when http adaptive streaming players compete for bandwidth? NOSSDAV '12, pages 89–94. ACM, 2012.
[5] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *SIGCOMM '04*, pages 281–292, New York, NY, USA, 2004. ACM.
[6] D. Bonfiglio, M. Mellia, M. Meo, and D. Rossi. Detailed analysis of skype traffic. *IEEE Transactions on Multimedia*, 11(1):117–127, Jan. 2009.
[7] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.
[8] J. Gettys and K. Nichols. Bufferbloat: dark buffers in the internet. *Commun. ACM*, 55(1):57–65, Jan. 2012.
[9] S. Lederer, C. Müller, and C. Timmerer. Dynamic adaptive streaming over http dataset. In *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, pages 89–94, New York, NY, USA, 2012. ACM.
[10] C. Müller and C. Timmerer. A test-bed for the dynamic adaptive streaming over http featuring session mobility. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pages 271–276, New York, NY, USA, 2011. ACM.
[11] R. S. Prasad, C. Dovrolis, and M. Thottan. Router buffer sizing for tcp traffic and the role of the output/input capacity ratio. *IEEE/ACM Transactions on Networking*, 17(5):1645–1658, Oct. 2009.
[12] P. Ranjan, E. Abed, and R. La. Nonlinear instabilities in tcp-red. *IEEE/ACM Transactions on Networking*, 12(6):1079–1092, Dec. 2004.
[13] T. Stockhammer. Dynamic adaptive streaming over http –: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pages 133–144, New York, NY, USA, 2011. ACM.