Signal Processing: *Image Communication* 27 (2012) 271–287

Contents lists available at SciVerse ScienceDirect

Signal Processing: *Image Communication*

journal homepage: www.elsevier.com/locate/image

An experimental evaluation of rate-adaptive video players over HTTP

Saamer Akhshabi [a,*], Sethumadhavan Narayanaswamy [a], Ali C. Begen [b], Constantine Dovrolis [a]

[a] *College of Computing, Georgia Institute of Technology, United States*
[b] *Video and Content Platforms, Research and Advanced Development, Cisco Systems, United States*

ARTICLE INFO

Available online 14 October 2011

Keywords:
Experimental evaluation
Adaptive streaming
Adaptive player
AdapTech Streaming
Video streaming over HTTP

ABSTRACT

Adaptive (video) streaming over HTTP is gradually being adopted by content and network service providers, as it offers significant advantages in terms of both user-perceived quality and resource utilization. In this paper, we first focus on the rate-adaptation mechanisms of adaptive streaming and experimentally evaluate two major commercial players (Smooth Streaming and Netflix) and one open-source player (Adobe's OSMF). We first examine how the previous three players react to persistent and short-term changes in the underlying network available bandwidth. Do they quickly converge to the maximum sustainable bitrate? We identify major differences between the three players and significant inefficiencies in each of them. We then propose a new adaptation algorithm, referred to as *AdapTech Streaming*, which aims to address the problems with the previous three players. In the second part of the paper, we consider the following two questions. First, what happens when two adaptive video players compete for available bandwidth in the bottleneck link? Can they share that resource in a stable and fair manner? And second, how does adaptive streaming perform with live content? Is the player able to sustain a short playback delay, keeping the viewing experience "live"?

1. Introduction

Video has long been viewed as the "next killer application".[1] Over the last 20 years, the various instances of packet video have been thought of as demanding applications that would never work satisfactorily over best-effort IP networks. That pessimistic view actually led to the creation of novel network architectures and QoS mechanisms, which were not deployed in a large-scale, though. Eventually, over the last 3–4 years video-based applications, and video streaming in particular, have become utterly popular generating more than half of the aggregate Internet traffic. Perhaps, surprisingly though, video streaming today runs over IP without any specialized support from the network. This has become possible through the gradual development of highly efficient video compression methods, the penetration of broadband access technologies, and the development of *adaptive video players* that can compensate for the unpredictability of the underlying network through sophisticated rate-adaptation, playback buffering, and error recovery and concealment methods.

Another conventional wisdom has been that video streaming would never work well over TCP, due to the throughput variations caused by TCP's congestion control

* Corresponding author.
E-mail addresses: s.akhshabi@gatech.edu (S. Akhshabi), sethumadhavan3@gatech.edu (S. Narayanaswamy), abegen@cisco.com (A.C. Begen), constantine@gatech.edu (C. Dovrolis).

[1] A shorter version of this paper initially appeared in ACM MMSys 2011 conference [1]. This longer version includes new results in Sections 3.3, 4.3 and 8, while Section 6 is new.

0923-5965/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.image.2011.10.003

and the potentially large retransmission delays. As a consequence, most of the earlier video streaming research has assumed that the underlying transport protocol is UDP (or RTP over UDP), which considerably simplifies the design and modeling of adaptive streaming applications. In practice, however, two points became clear in the last few years. First, TCP's congestion control mechanisms and reliability requirement do not necessarily hurt the performance of video streaming, especially if the video player is able to adapt to large throughput variations. Second, the use of TCP, and of HTTP over TCP in particular, greatly simplifies the traversal of firewalls and NATs.

The first wave of HTTP-based video streaming applications used the simple *progressive download* method, in which a TCP connection simply transfers the entire movie file as quickly as possible. The shortcomings of that approach are many, however. One major issue is that all clients receive the same encoding of the video, despite the large variations in the underlying available bandwidth both across different clients and across time for the same client. This has recently led to the development of a new wave of HTTP-based streaming applications that we refer to as *adaptive streaming over HTTP* (For a general overview of video streaming protocols and adaptive streaming, refer to [2]). Several players, such as Microsoft's Smooth Streaming, Adobe OSMF, as well as the players developed or used by Netflix, Move Networks and others, use this approach. In adaptive streaming, the server keeps multiple profiles of the same video, encoded in different bitrates and quality levels. Further, the video object is partitioned in *chunks* or *fragments*, typically a few seconds long. A player can then request different chunks at different encoding bitrates, depending on the underlying network conditions. Notice that it is the player that decides what bitrate to request for any chunk, improving server-side scalability. Another benefit of this approach is that the player can control its playback buffer size by dynamically adjusting the rate at which new chunks are requested.

Adaptive streaming over HTTP is a new technology. It is not yet clear whether the existing commercial players perform well, especially under dynamic network conditions. Further, the complex interactions between TCP's congestion control and the application's rate-adaptation mechanisms create a "nested double feedback loop"—the dynamics of such interacting control systems can be notoriously complex and hard to predict. As a first step toward understanding and improving such video streaming mechanisms, this paper experimentally evaluates two major commercial video players over HTTP (Microsoft's Smooth Streaming [17] and the player used by Netflix [15]) and one open-source player (Adobe's OSMF [16]).

In the first part of the paper (Sections 3–5), we examine how the previous three players react to persistent and short-term variations in the underlying network available bandwidth. Do they quickly converge to the maximum sustainable bitrate? We identify major differences between the three players and significant inefficiencies in each of them. Then, in Section 6, we propose a new adaptation algorithm, referred to as *AdapTech Streaming* and implemented in Adobe's OSMF v1.5 player, which aims to address the issues with the previous three players.

In the second part of the paper (Sections 7 and 8), we consider the following two questions. First, what happens when two adaptive video players compete for available bandwidth in the bottleneck link? Can they share that resource in a stable and fair manner? And second, how does adaptive streaming perform with live content? Is the player able to sustain a short playback delay, keeping the viewing experience "live"?

1.1. Paper outline

In Section 2, we describe our experimental approach, the various tests we perform for each player, and the metrics we focus on. Sections 3–5 focus on the Smooth Streaming, Netflix, and OSMF players, respectively. The proposed rate adaptation algorithm, *AdaptTech Streaming*, is described in Section 6. Section 7 focuses on the competition effects that take place when two adaptive players share the same bottleneck. Section 8 focuses on live video using the Smooth Streaming player. In Section 9, we review the related work on adaptive video streaming. We summarize what we learn for each player and conclude the paper in Section 10.

2. Methodology and metrics

In this section, we give an overview of our experimental methodology and describe the metrics we focus on. The host that runs the various video players also runs a packet sniffer (Wireshark [14]) and a network emulator (DummyNet [20]). Wireshark allows us to capture and analyze offline the traffic from and to the HTTP server. DummyNet allows us to control the *downstream available bandwidth* (also referred to as *avail-bw*) that our host can receive. That host is connected to the Georgia Tech campus network through a Fast Ethernet interface. When we do not limit the avail-bw using DummyNet, the video players always select the highest rate streams; thus, when DummyNet limits the avail-bw to relatively low bitrates (1–5 Mbps) we expect that it is also the downstream path's end-to-end bottleneck.

In the following, we study the throughput-related metrics:

1. The *avail-bw* refers to the bitrate of the bottleneck that we emulate using DummyNet. The TCP connections that transfer video and audio streams cannot exceed (collectively) that bitrate at any point in time.
2. The 2-*sec connection throughput* refers to the download throughput of a TCP connection that carries video or audio traffic, measured over the last 2 s.
3. The *running average of a connection's throughput* refers to a running average of the 2-sec connection throughput measurements. If $A(t_i)$ is the 2-sec connection throughput in the ith time interval, the running

average of the connection throughput is

$$\hat{A}(t) = \begin{cases} \delta\hat{A}(t_{i-1}) + (1-\delta)A(t_i) & i > 0 \\ A(t_0) & i = 0 \end{cases}$$

In the experiments, we use $\delta = 0.8$.

4. The *(audio or video) chunk throughput* refers to the download throughput for a particular chunk, i.e., the size of that chunk divided by the corresponding download duration. Note that, if a chunk is downloaded in every 2 s, the chunk throughput can be much higher than the 2-sec connection throughput in the same time interval (because the connection can be idle during part of that time interval). As will be shown later, some video players estimate the avail-bw using chunk throughput measurements.

We also estimate the *playback buffer size* at the player (measured in seconds), separately for audio and video. We can accurately estimate the playback buffer size for players that provide a timestamp (an offset value that indicates the location of the chunk in the stream) in their HTTP chunk requests. Suppose that two successive, say video, requests are sent at times t_1 and t_2 ($t_1 < t_2$) with timestamps t'_1 and t'_2 ($t'_1 < t'_2$), respectively (all times measured in seconds). The playback buffer duration in seconds for video at time t_2 can be then estimated as

$$B(t_2) = [B(t_1) - (t_2 - t_1) + (t'_2 - t'_1)]^+$$

where $[x]^+$ denotes the maximum of x and 0. This method works accurately because, as will be clear in the following sections, the player requests are not pipelined: a request for a new chunk is sent only after the previous chunk has been fully received.

We test each player under the same set of avail-bw conditions and variations. In the first round of tests, we examine the behavior of a player when the avail-bw is not limited by DummyNet; this "blue-sky" test allows us to observe the player's start-up and steady-state behavior—in the same experiments we also observe what happens when the user skips to a future point in the video clip. In the second round of tests, we apply persistent avail-bw variations (both increases and decreases) that last for tens of seconds. Such variations are common in practice when the cross traffic in the path's bottleneck varies significantly due to arriving or departing traffic from other users. A good player should react to such variations by decreasing or increasing the requested bitrate. In the third round of tests, we apply positive and negative spikes in the path's avail-bw that last for just few seconds—such variations are common in 802.11 WLANs for instance. For such short-term drops, the player should be able to maintain a constant requested bitrate using its playback buffer. For short-term avail-bw increases, the player could be conservative and stay at its current rate to avoid unnecessary bitrate variations. In these experiments, we use different sets of video content from the Web specific to each player examined here. We do not encode the content and control the servers ourselves. As a result, we do not test the different players with a consistent set of video content. Since we are not interested in the actual video quality, the content is not primarily important in producing these results. However, we repeat the experiments several times for each player with different sets of video content and we observe consistent results for the same player. Due to space constraints, we do not show results from all these experiments for each player; we select only those results that are more interesting and provide new insight.

All experiments were performed on a Windows Vista Home Premium version 6.0.6002 laptop with an Intel(R) Core(TM)2 Duo P8400 2.26 GHz processor, 3.00 GB physical memory, and an ATI Radeon Graphics Processor ($0 \times 5C4$) with 512 MB dedicated memory.

3. Microsoft smooth streaming

In the following experiments, we use Microsoft Silverlight Version 4.0.50524.0. In a Smooth Streaming manifest file, the server declares the available audio and video bitrates and the resolution for each content (among other information). The manifest file also contains the duration of every audio and video chunk. After the player has received the manifest file, it generates successive HTTP requests for audio and video chunks. Each HTTP request from the player contains the name of the content, the requested bitrate, and a timestamp that points to the beginning of the corresponding chunk. This timestamp is determined using the per-chunk information provided in the manifest. The following is an example of a Smooth Streaming HTTP request:

```
GET (..)/BigBuckBunny720p.ism/
QualityLevels(2040000)/
Fragments(video=400000000) HTTP/1.1
```

In this example, the requested bitrate is 2.04 Mbps and the chunk timestamp is 40 s.

The Smooth Streaming player maintains two TCP connections with the server. At any point in time, one of the two connections is used for transferring audio and the other for video chunks. Under certain conditions, however, the player switches the audio and video streams between the two connections—it is not clear to us when/how the player takes this decision. This way, although at any point in time one connection is transferring video chunks, over the course of streaming, both connections get the chance to transfer video chunks. The benefit of such switching is that neither of the connections would stay idle for a long time, keeping the server from falling back to slow-start. Moreover, the two connections would maintain a large congestion window.

Sometimes the player aborts a TCP connection and opens a new one—this probably happens when the former connection provides very low throughput. Also, when the user jumps to a different point in the stream, the player aborts the existing TCP connections, if they are not idle, and opens new connections to request the appropriate chunks. At that point, the contents of the playback buffer are flushed.

In the following experiments we watch a sample video clip ("Big Buck Bunny") provided by Microsoft at the IIS

Web site:

http://www.iis.net/media/experiencesmoothstreaming

The manifest file declares eight video encoding bitrates between 0.35 Mbps and 2.75 Mbps and one audio encoding bitrate (64 Kbps). We represent an encoding bitrate of r Mbps as P_r, (e.g., $P_{2.75}$). Each video chunk (except the last) has the same duration: $\tau = 2$ *s*. The audio chunks are approximately of the same duration.

3.1. *Behavior under unrestricted avail-bw*

Fig. 1 shows the various throughput metrics, considering only the video stream, in a typical experiment without restricting the avail-bw using DummyNet. $t=0$ corresponds to the time when the Wireshark capture starts. Note that the player starts from the lowest encoding bitrate and it quickly, within the first 5–10 s, climbs to the highest encoding bitrate. As the per-chunk throughput measurements indicate, the highest encoding bitrate ($P_{2.75}$) is significantly lower than the avail-bw in the end-to-end path. The player upshifts to the highest encoding profile from the lowest one in four transitions. In other words, it seems that the player avoids large jumps in the requested bitrate (more than two successive bitrates)—the goal is probably to avoid annoying the user with sudden quality transitions, providing a dynamic but smooth watching experience.

Another important observation is that during the first 4 s after the user clicked Play, the player asks for video chunks much more frequently than once every τ seconds (see Fig. 2). Further analysis of the per-chunk interarrivals and download times shows that the player operates in one of two states: `Buffering` and `Steady-State`. In the former, the player requests a new chunk as soon as the previous chunk was downloaded. Note that the player does *not* use HTTP pipelining—it does not request a chunk if the previous chunk has not been fully received. In `Steady-State`, on the other hand, the player requests a new chunk either τ seconds after the previous chunk was requested (if it took less than τ seconds to download that chunk) or as soon as the previous chunk was received (otherwise). In other words, in the `Buffering-State` the player aims to maximize its chunk request rate so that it can build up a target playback buffer as soon as possible. In `Steady-State`, the player aims to maintain a constant playback buffer, requesting one chunk every τ seconds (recall that each chunk corresponds to τ seconds of content). We estimated the target video playback buffer size, as described in Section 2, to be about 30 s. The time it takes to reach `Steady-State` depends on the avail-bw—as the avail-bw increases, it takes less time to accumulate the 30-sec playback buffer. In this experiment, it took the player 4 s to reach the `Steady-State`. We have consistently observed that the player does not sacrifice quality, requesting low-bitrate encodings, to fill up its playback buffer sooner. Another interesting observation is that the player does not request a video bitrate whose frame resolution (as declared at the manifest file) is larger than the resolution of the display window.

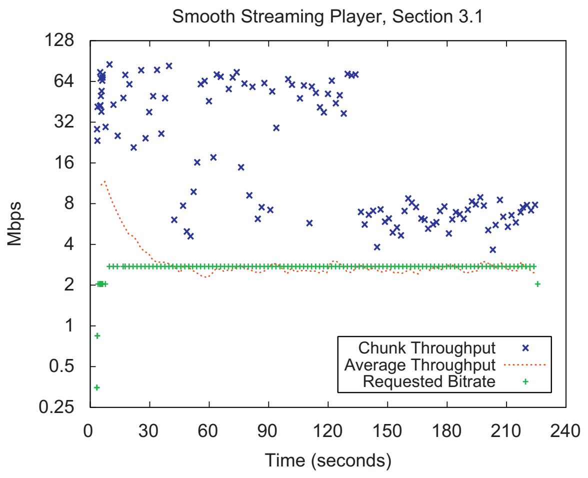

Fig. 1. Per-chunk throughput, average TCP throughput and the requested bitrate for video traffic under unrestricted avail-bw conditions. Playback starts at around $t=5$ s, almost 2 s after the user clicked Play.

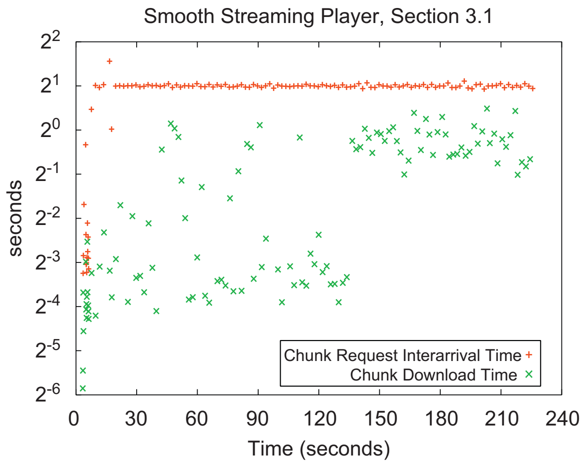

Fig. 2. Interarrival and download times of video chunks under unrestricted avail-bw conditions. Each chunk is 2 s long.

3.2. *Behavior of the audio stream*

Audio chunks are of the same duration with video chunks, at least in the movies we experimented with. Even though audio chunks are much smaller in bytes than video chunks, the Smooth Streaming player does not attempt to accumulate a larger audio playback buffer than the corresponding video buffer (around 30 s). Also, when the avail-bw drops, the player does not try to request audio chunks more frequently than video chunks (it would be able to do so). Overall, it appears that the Smooth Streaming player attempts to keep the audio and video stream download processes as much in sync as possible.

3.3. *Behavior under persistent avail-bw variations*

In this section, we summarize a number of experiments in which the avail-bw goes through four significant and persistent transitions, as shown in Fig. 3. First, note

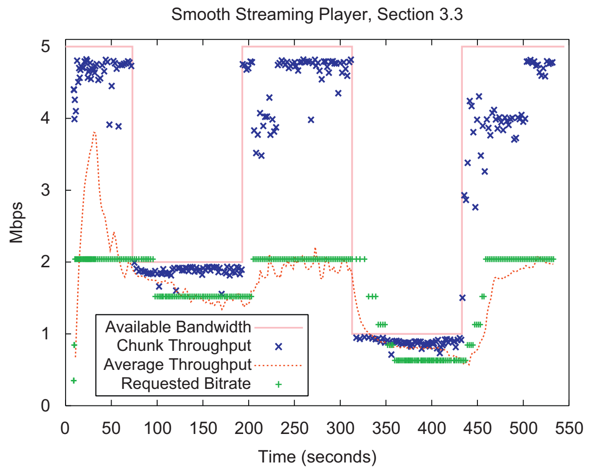

Fig. 3. Per-chunk throughput, average TCP throughput and the requested bitrate for the video traffic under persistent avail-bw variations. Playback starts at around $t=10$ s, almost 3 s after the user clicked Play.

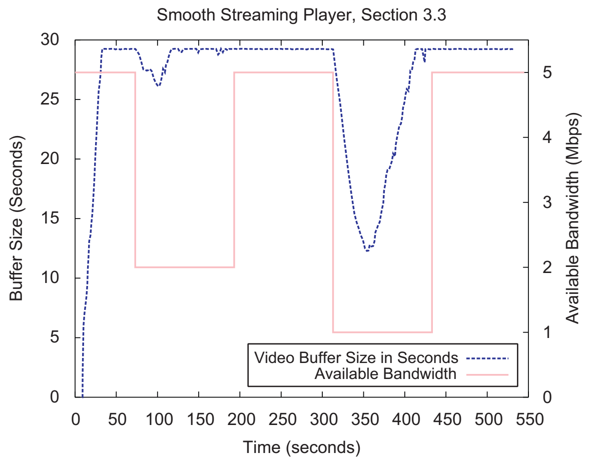

Fig. 4. Video playback buffer size in seconds under persistent avail-bw variations.

that, as expected, the per-chunk throughput is never higher than the avail-bw. Instead, the per-chunk throughput tracks quite well the avail-bw variations for most of the time; part of the avail-bw, however, is consumed by audio chunks and, more importantly, TCP throughput can vary significantly after packet loss events.

We next focus on the requested video bitrate as the avail-bw changes. Initially, the avail-bw is 5 Mbps and the player requests the $P_{2.04}$ profile because it is constrained by the resolution of the display window (if we were watching the video in full-screen mode, the player would request the highest $P_{2.75}$ profile). The playback buffer (shown in Fig. 4) has reached its 30 s target by $t=40$ s and the player is in `Steady-State`.

At time $t=73$ s, the avail-bw is dropped to 2 Mbps—that is not sufficient for the $P_{2.04}$ encoding because we also need some capacity for the audio traffic and for various header overheads. The player reacts by switching to the next lower profile ($P_{1.52}$) but after some significant delay (almost 25 s). During that time period, the playback buffer has decreased by only 3 s (the decrease is not large because the avail-bw is just barely less than the cumulative requested traffic). The large reaction delay indicates that the player does *not* react to avail-bw changes based on the latest per-chunk throughput measurements. Instead, it averages those per-chunk measurements over a longer time period so that it acts based on a smoother estimate of the avail-bw variations. The playback buffer size returns to its 30 s target after the player has switched to the $P_{1.52}$ profile.

The avail-bw increase at $t=193$ s is quickly followed by an appropriate increase in the requested encoding bitrate. Again, the switching delay indicates that the Smooth Streaming player is conservative, preferring to estimate reliably the avail-bw (using several per-chunk throughput measurements) instead of acting opportunistically based on the latest chunk throughput measurement.

The avail-bw decrease at $t=303$ s is even larger (from 5 Mbps to 1 Mbps) and the player reacts by adjusting the requested bitrate in four transitions. The requested bitrates are not always successive. After those transitions, the request bitrate converges to an appropriate value $P_{0.63}$, much less than the avail-bw. It is interesting that the player could have settled at the next higher bitrate ($P_{0.84}$)—in that case, the aggregate throughput (including the audio stream) would be 0.94 Mbps. That is too close to the avail-bw (1 Mbps), however. This implies that Smooth Streaming is conservative: it prefers to maintain a safety margin between the avail-bw and its requested bitrate. We think that this is wise, given that the video bitrate can vary significantly around its *nominal encoding value* due to the variable bitrate (VBR) nature of video compression.

Another interesting observation is that the player avoids large transitions in the requested bitrate—such quality transitions can be annoying to the viewer. Also, the upward transitions are faster than the downward transitions—still, however, it can take several tens of seconds until the player has switched to the highest sustainable bitrate.

Fig. 5 shows the throughput of the two TCP connections used by the Smooth Streaming player. It also

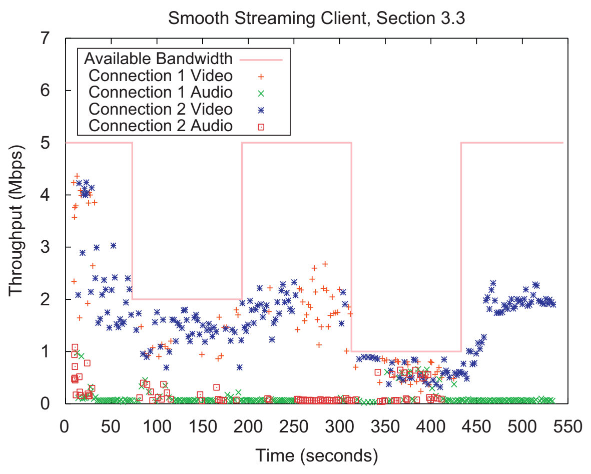

Fig. 5. Throughput of the two TCP connections carrying audio and video chunks.

illustrates whether a connection is used for video or audio chunks.

We measure throughput as follows. We obtain individual throughput measurements for intervals between two successive requests. If two successive video requests are sent at times t_1 and t_2, and b bits are downloaded in that period, then the throughput of the TCP connection at time t_2 is $b/(t_2-t_1)$ and is a *video throughput measurement*. Note that in the throughput measurements, we do not consider TCP/IP header overheads (TCP payload sizes are 1260 bytes).

Fig. 5 shows that at any point in time, the player uses one of the connections to download video and the other to download audio. The player does not download two chunks of the same type simultaneously on two different connections. The figure also shows that the player often switches the audio and video downloads between the two TCP connections. As expected, the audio download duration is much smaller than that for video. During the time period between two successive requests, the connection that transfers audio stays idle. Switching audio and video between two connections helps the connection from staying idle too long, avoiding slow-start restart (RFC 5681). It also helps both connections to maintain a large congestion window.

3.4. Behavior under short-term avail-bw variations

In this section, we summarize a number of experiments in which the avail-bw goes through positive or negative "spikes" that last for only few seconds, as shown in Figs. 6 and 8. The spikes last for 2 s, 5 s and 10 s, respectively. Such short-term avail-bw variations are common in practice, especially in 802.11 WLAN networks. We think that a good adaptive player should be able to compensate for such spikes using its playback buffer, without causing short-term rate adaptations that can be annoying to the user.

Fig. 6 shows the case of positive spikes. Here, we repeat the three spikes twice, each time with a different

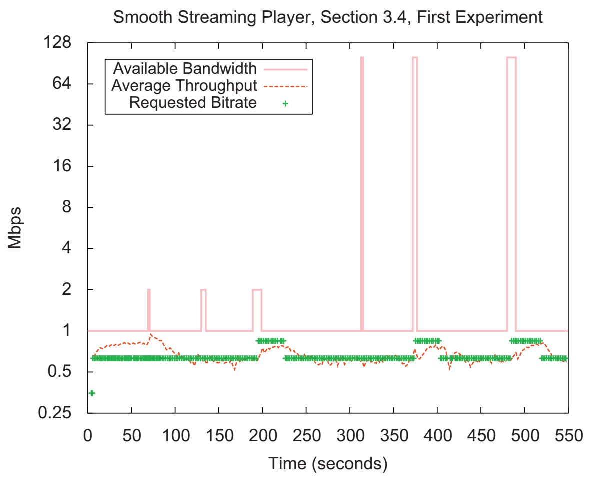

Fig. 6. Average TCP throughput and the requested bitrate for the video traffic under positive avail-bw spikes. Playback starts at around $t=7$ s, almost 4 s after the user clicked Play.

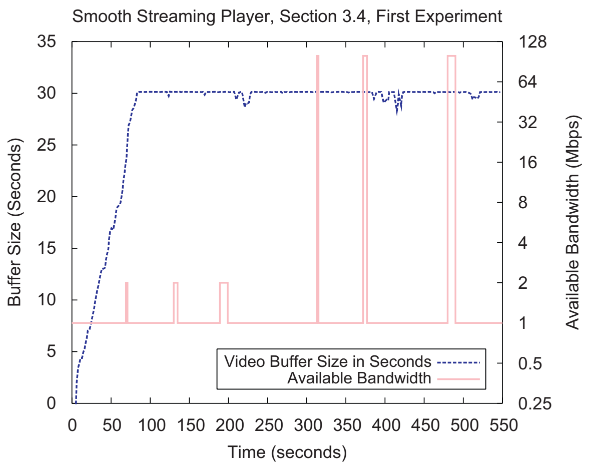

Fig. 7. Video playback buffer size in seconds under positive avail-bw spikes.

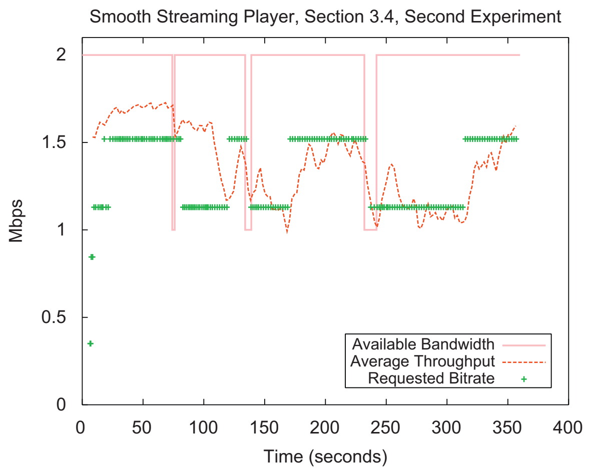

Fig. 8. Average TCP throughput and the requested bitrate for the video traffic under negative avail-bw spikes. Playback starts at around $t=9$ s, almost 3 s after the user clicked Play.

increase magnitude. The Smooth Streaming player ignores the 2-sec spikes and the smaller 5-sec spike. On the other hand, it reacts to the 10-s spikes by increasing the requesting video bitrate. Unfortunately, it does so too late (sometimes after the end of the spike) and for too long (almost till 40 s after the end of the spike). During that interval, the player is more vulnerable to freeze events (see Fig. 7). This experiment confirms that the player reacts, not to the latest chunk download throughput, but to a smoothed estimate of those measurements that can be unrelated to the current avail-bw conditions.

Figs. 8 and 9 show similar results in the case of negative spikes. Here, the spikes reduce the avail-bw from 2 Mbps to 1 Mbps. The player reacts to all three spikes, even the spike that lasts for only 2 s. Unfortunately, the player reacts too late and for too long: it requests a lower bitrate after the end of each negative spike and it stays at that lower bitrate long for 40–80 s. During those periods, the user would unnecessarily experience a lower video quality.

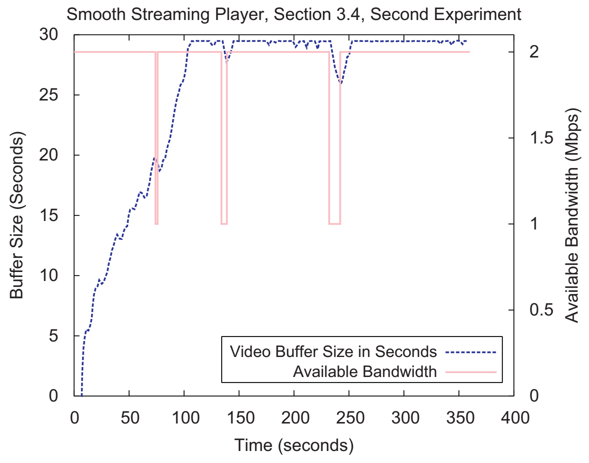

Fig. 9. Video playback buffer size in seconds under negative avail-bw spikes.

4. Netflix player

The Netflix player uses Microsoft's Silverlight for media representation, but a different rate-adaptation logic. The Netflix player also maintains two TCP connections with the server, and it manages these TCP connections similarly with the Smooth Streaming player. As will become clear, however, the Netflix player does not send audio and video chunk requests at the same pace. Also, the format of the manifest file and requests are different. Further, most of the initial communication between the player and server, including the transfer of the manifest file, is done over SSL. We decrypted the manifest file using a Firefox plugin utility called Tamper Data [4] that acts as a man-in-the-middle in Firefox. Video and audio chunks are delivered in wmv and wma formats, respectively. An example of a Netflix chunk request follows:

```
GET /sa2/946/1876632946.wmv
/range/2212059-2252058?token=1283923056
_d6f6112068075f1fb60cc48eab59ea55&random
=1799513140 HTTP/1.1
```

Netflix requests do not correspond to a certain time duration of audio or video. Instead, each request specifies a range of bytes in a particular encoding profile. Thus, we cannot estimate the playback buffer size as described in Section 2. We can only approximate that buffer size assuming that the actual encoding rate for each chunk is equal to the corresponding nominal bitrate for that chunk (e.g., a range of 8 Mb at the $P_{1.00}$ encoding profile corresponds to 8 s worth of video)—obviously this is only an approximation but it gives us a rough estimate of the playback buffer size.

After the user clicks the PLAY button, the player starts by performing some TCP transfers, probably to measure the capacity of the underlying path. Then it starts buffering audio and video chunks, but without starting the playback yet. The playback starts either after a certain number of seconds or when the buffer size reaches a target point. If that buffer is depleted at some point, the Netflix player prefers to stop the playback, showing a message that the player is adjusting to a slower connection. The playback resumes when the buffer size reaches a target point.

In the following experiments we watch the movie "Mary and Max". The manifest file provides five video encoding bitrates between 500 Kbps and 3.8 Mbps and two audio encoding bitrates (64 Kbps and 128 Kbps).

4.1. Behavior under unrestricted avail-bw

Fig. 10 shows the various throughput metrics, considering only the video stream, in a typical experiment without using DummyNet to restrict the avail-bw. The interarrival of video chunk requests and the download times for each video chunk are shown in Fig. 11. In this experiment, the playback started about 13 s after the user

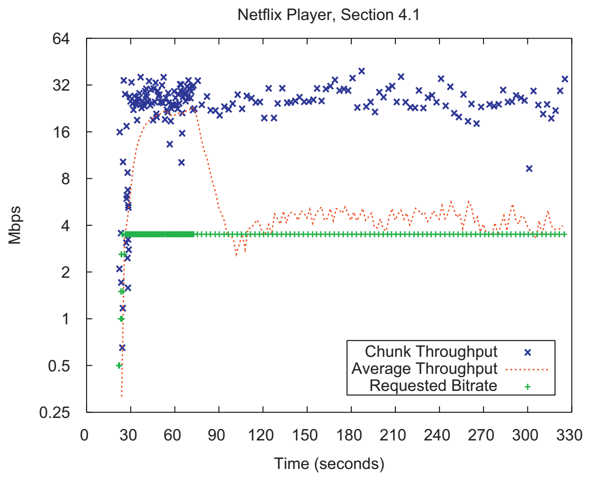

Fig. 10. Per-chunk throughput, average TCP throughput and the requested bitrate for video traffic under unrestricted avail-bw conditions. Playback starts at around $t=24$ s, almost 16 s after the user clicked PLAY.

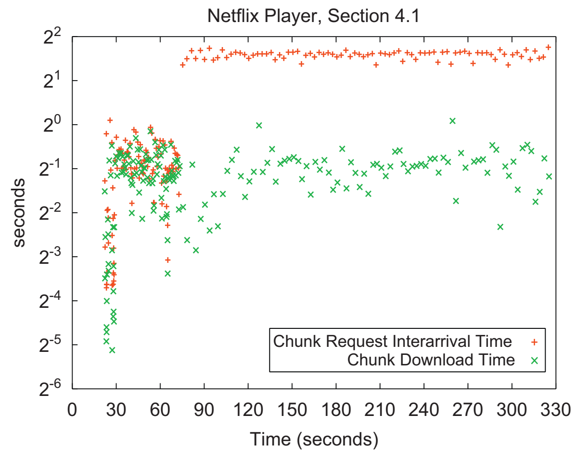

Fig. 11. Interarrival and download times of video chunks under unrestricted avail-bw conditions.

clicked Play. The playback delay can be much larger depending on the initial avail-bw (even up to few minutes). During this interval, several security checks are also performed before the player starts buffering and the playback begins [13]. For the first few chunks, the player starts from the lowest encoding bitrate and requests a number of chunks from all the available bitrates. Then, the player stays at that highest bitrate for the duration of the experiment.

During the first 55 s of streaming, until $t=75$ s, the player is clearly in the `Buffering` state: it requests a new video chunk right after the previous chunk has been downloaded. The achieved TCP throughput in this path is about 30 Mbps, allowing the player to quickly accumulate a large playback buffer. We estimated the size of the playback buffer at the end of the `Buffering` state ($t=75$ s) at about 300 s worth of video—this is an order of magnitude larger than the playback buffer size we observed for Smooth Streaming.

When the player switches to `Steady-State`, video chunks are requested almost every 3 s, with significant variation, however.

4.2. Behavior of the audio stream

Audio chunks in the Netflix player are significantly larger than the ones in Smooth Streaming. Specifically, an audio chunk is typically 30 s long. Thus, after the player has reached `Steady-State`, a new audio chunk is requested every 30 s. Further, it appears that this player does not attempt to keep the audio and video stream download processes in sync; it can be that the audio playback buffer size is significantly larger than the video playback buffer size.

4.3. Behavior under persistent avail-bw variations

Fig. 12 shows the various throughput-related metrics in the case of persistent avail-bw variations. As in the experiment with unrestricted avail-bw, the player first requests few chunks at all possible encodings. Within the first 40 s it converges to the highest sustainable bitrate ($P_{1.50}$) for that avail-bw (2 Mbps). It should be noted that in this experiment the player never leaves the `Buffering-State` (based on analysis of the video chunk request interarrivals).

When the avail-bw drops to 1 Mbps, the player reacts within about 20 s, which implies that its avail-bw estimator is based on a smoothed version of the underlying per-chunk throughput, as opposed to the instantaneous and latest such measurement. It is interesting that the selected profile at that phase ($P_{1.00}$) is not sustainable, however, because it is exactly equal to the avail-bw (some avail-bw is consumed by audio traffic and other header overheads). Thus, the playback buffer size slowly decreases, forcing the player between 320 and 400 s to occasionally switch to the next lower bitrate. This observation implies that the Netflix player prefers to utilize a certain high bitrate even when the avail-bw is insufficient, as long as the player has accumulated more than a certain playback buffer size. We make the same observation from 450 to 500 s. During that

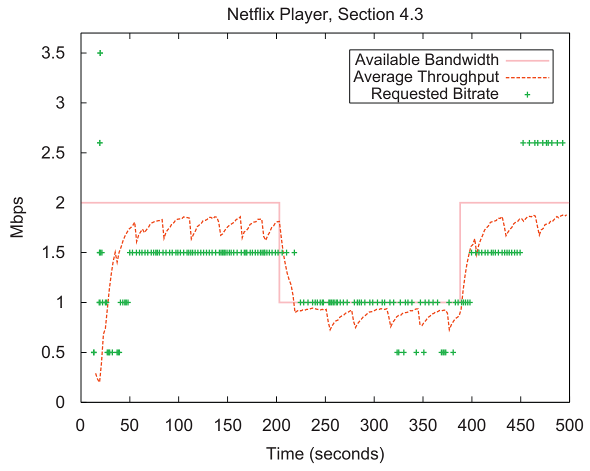

Fig. 12. Average TCP throughput and the requested bitrate for the video traffic under persistent avail-bw variations. Playback starts at around $t=20$ s, almost 13 s after the user clicked Play.

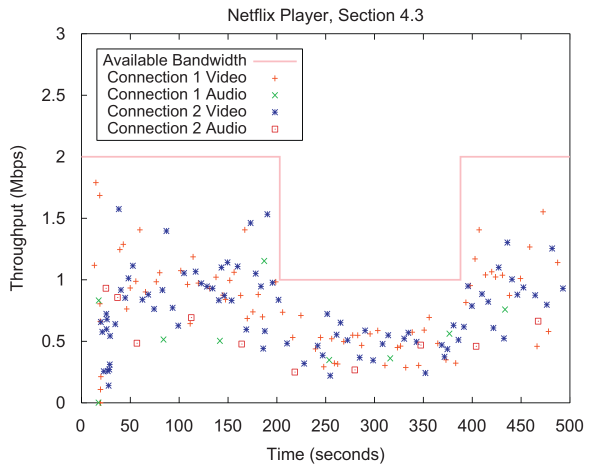

Fig. 13. Throughput of the two TCP connections carrying audio and video chunks.

interval, the player switches to a profile ($P_{2.60}$) that is much higher than the avail-bw (2 Mbps). The player can do this, without causing any problems, because it has accumulated a sufficiently large playback buffer size at that point.

In summary, it appears that the Netflix player is more aggressive than Smooth Streaming, trying to deliver the highest possible encoding rate even when the latter is more than the avail-bw, as long as the playback buffer size is sufficiently large.

Fig. 13 shows the throughput of the two TCP connections used by the Netflix player. Unlike Smooth Streaming, the Netflix player requests an audio chunk every 30 s while in `Steady-State`. The chunk itself contains 30 s worth of audio, which is much larger than the chunk duration in the Smooth Streaming player. The player mostly switches the audio chunk requests between the two TCP connections everytime it sends a new audio request. During the time periods that no audio requests are sent, the player uses both connections to request video chunks.

4.4. Behavior under short-term avail-bw variations

Fig. 14 shows how the Netflix player reacts to positive avail-bw spikes, while Fig. 15 shows how it reacts to negative avail-bw spikes. We cannot compare directly the results of these experiments to the corresponding experiments with Smooth Streaming, because the video encoding profiles are different between the movies in these two experiments. As in the case of persistent avail-bw variations, we observe that the Netflix player is rather aggressive, reacting to large increases in avail-bw even if they are short-lived. As opposed to Smooth Streaming, which often reacts too late and for too long, Netflix reacts faster, while the spike is still present, even though the reaction can still last much longer after the spike.

On the other hand, in the experiment with negative avail-bw spikes, the Netflix player does not switch to a lower bitrate. It prefers to compensate for the lack of avail-bw using the large playback buffer size that it has previously accumulated.

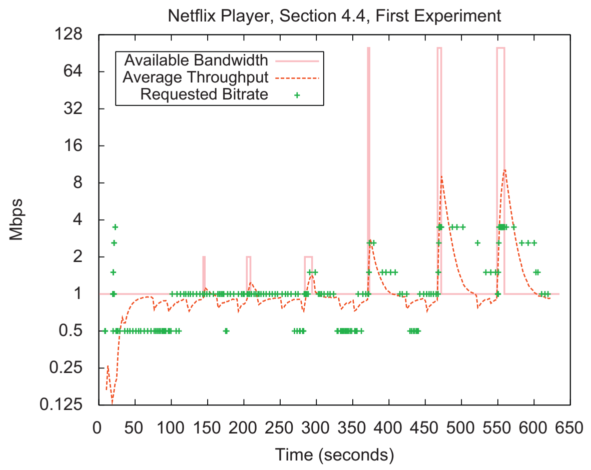

Fig. 14. Average TCP throughput and the requested bitrate for the video traffic under positive avail-bw spikes. Playback starts at around $t=20$ s, almost 16 s after the user clicked PLAY.

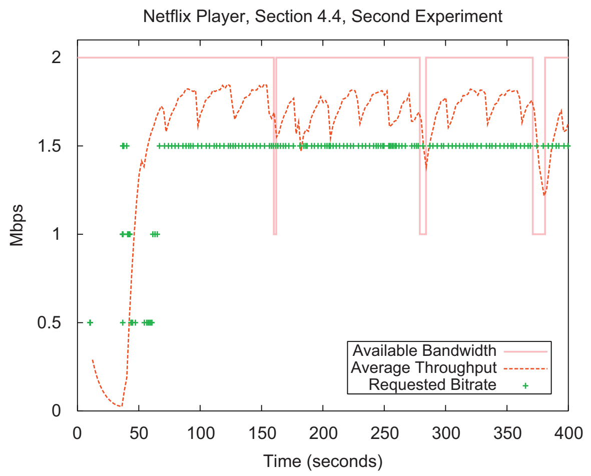

Fig. 15. Average TCP throughput and the requested bitrate for the video traffic under negative avail-bw spikes. Playback starts at around $t=37$ s, almost 31 s after the user clicked PLAY.

5. Adobe OSMF

We have repeated the same set of tests with Adobe's sample OSMF player, using Flash version 10.1.102.64 with player version WIN 10.1.102.64 and OSMF library version 1.0. In the following experiments, we watch a movie trailer ("Freeway") provided by Akamai's HD-video demo Web site for Adobe HTTP Dynamic Streaming:

http://wwwns.akamai.com/hdnetwork/demo/flash/zeri/

Note that the player used in this Web site was not built specifically to showcase HTTP Dynamic Streaming.

The manifest file declares eight encoding bitrates for this trailer between 0.25 Mbps and 3.00 Mbps. In this player, each server file (F4F format) contains one segment of the movie. A segment can contain multiple chunks. The duration of each chunk is determined by the server. The HTTP requests that the player generates include a chunk number instead of a timestamp or a byte range. An example of an OSMF HTTP request follows:

```
GET /content/inoutedit-mbr
/inoutedit_h264_3000Seg1-Frag5 HTTP/1.1
```

Note that the requested bitrate is shown in the request (3000 Kbps) together with the requested segment and chunk numbers. The player maintains one TCP connection with the server and receives all chunks through this connection. The player might shut down the connection and open a new one if the user jumps to a different point in the stream and the connection is not idle. According to information provided by the player itself, the target playback buffer seems to be less than 10 s.

Fig. 16 shows that initially the player requests one chunk at the lowest available profile and then quickly climbs to the largest possible one. But, it does not converge to that profile and continues switching between profiles occasionally. When the avail-bw is dropped to 2 Mbps at $t=139$ s, the player fails to converge to the highest sustainable profile ($P_{1.7}$). Instead, it keeps switching between profiles, often using the lowest and highest ones ($P_{0.25}$ and $P_{3.00}$). The user observes several dropped frames and freeze events, which is an indication of a depleted playback buffer.

We have also conducted the same set of experiments with the latest version of the OSMF player obtained from the following Web site.

http://sourceforge.net/adobe/osmf/home/

Fig. 17 shows the various throughput related metrics in an experiment with the OSMF player version 1.5 under persistent avail-bw variations. We see a very similar issue here. The player makes similar problematic rate switchings and gets into oscillation.

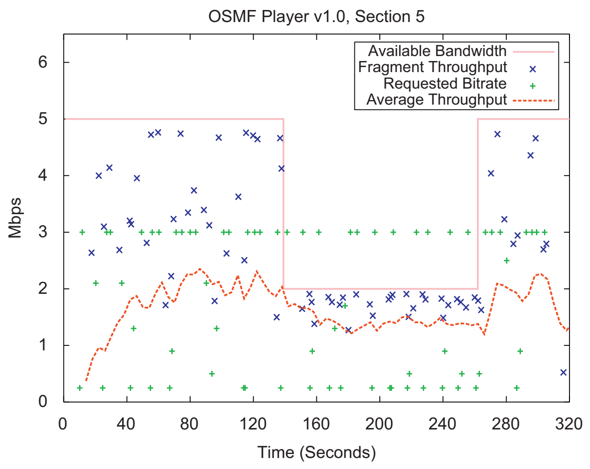

Fig. 16. Per-chunk throughput, the requested bitrate for the video traffic and average TCP throughput under persistent avail-bw variations. Playback starts at around $t=13$ s, almost 3 s after the user clicked Play.

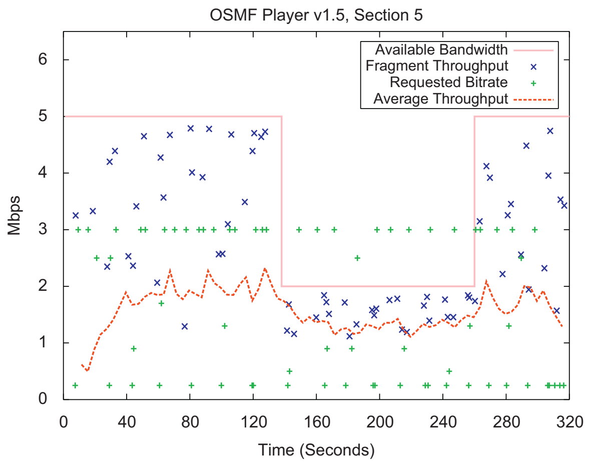

Fig. 17. Per-chunk throughput, the requested bitrate for the video traffic and average TCP throughput under persistent avail-bw variations. Playback starts at around $t=11$ s, almost 4 s after the user clicked Play.

To summarize, we have observed that the OSMF player fails to converge to a sustainable bitrate especially when the avail-bw is smaller than or very close to the highest available bitrate of the media. Instead, it usually oscillates between the lowest and highest bitrates. The default rate-adaptation algorithm seems to be tuned for short variations in the avail-bw. We do not describe here the rest of the experiments we performed with it, because they simply confirm that the default rate-adaptation algorithm deployed in the OSMF player does not function properly under our test scenarios.

6. AdapTech streaming player

We now propose a new adaptation algorithm, referred to as `AdapTech Streaming`, which aims to address the observed issues with the previous three players. We have implemented `AdapTech` using Adobe's OSFM player v1.5. In the following, we first describe our design objectives, the *AdapTech* algorithm, and then present experimental results under persistent and short-term avail-bw variations.

6.1. Objectives and adaptation algorithm

The design objectives behind *AdapTech* are:

- Start the video playback quickly—the initial playback delay should be much shorter than the maximum delay in the playback buffer.
- The top-priority should be to avoid playback buffer underflows and video freezes.
- Provide smooth bitrate transitions so that the user does not notice large video quality fluctuations.
- Detect *persistent* avail-bw increases, and when that is the case switch to the highest sustainable video bitrate.
- Avoid reacting to positive or negative short-term avail-bw spikes.

In the following we explain how we achieve the previous objectives. First, just like Smooth Streaming, the player has two states: `Buffering` and `Steady-State`. In the former, the player builds up its playback buffer as fast as possible by requesting a new chunk immediately after the previous has been downloaded. The player switches to `Steady-State` when the buffer size, $\beta(t)$, reaches a predefined threshold β_{max}. In `Steady-State`, the player requests a new chunk every time a previous chunk has been removed from the playback buffer (e.g., in every 2 s, if that is the chunk duration). Thus, in *Steady State* $\beta(t)$ is almost constant. We set β_{max} to 30 s. The player starts in the `Buffering-State` with the lowest possible profile. The playback starts when $\beta(t)$ reaches a threshold θ_1. So, the initial playback delay is typically much shorter than β_{max}-the exact duration of the initial playback delay depends on the ratio of the network avail-bw to the lowest video bitrate, and on θ_1.

The player decides the next requested profile when the previous chunk has been received. The player maintains two throughput related metrics. The first is the throughput of the last downloaded chunk, denoted as A. The second is a smooth estimate of the avail-bw using an exponential running average of the per-chunk TCP throughput measurements. If $A(i)$ is the throughput of the ith chunk, the running average $\hat{A}$ is

$$\hat{A} = \begin{cases} \delta\hat{A}(i-1)+(1-\delta)A(i) & i>1 \\ A(1) & i=1 \end{cases}$$

We use $\delta=0.8$.

Assuming that the available profiles are sorted in terms of increasing bitrate, and the bitrate of the ith profile is b_i, the indices of the two candidate profiles ϕ_1 and ϕ_2 (hereafter for simplicity also referred to as candidate profiles) are given by

$$\phi_1 = \max\{i : b_i < c \times A\}$$

$$\phi_2 = \max\{i : b_i < c \times \hat{A}\}$$

where c is a slack parameter $(0<c<1)$ that is necessary due to the variability of the video encoding rate—we use

$c=0.8$. ϕ_1 is the index of the candidate profile based on the most recent throughput measurement. It is used to quickly detect major changes in the avail-bw but it can be noisy. ϕ_2 is the index of the candidate profile based on the running average of the per-chunk throughput measurements. ϕ_2 is less noisy but it can be slow in detecting avail-bw variations. The index of the current profile is denoted by ϕ_{cur}. If there is no profile that satisfies either of the previous conditions, the index of the candidate profile is set to 1 (i.e. the smallest available profile).

As previously stated, our first priority is to avoid playback buffer depletion and video freezes. To do so, the player reacts quickly if the buffer level drops below a certain critical threshold. In parallel however, we want to avoid overreacting to short-term drops (spikes) of the avail-bw. To meet both goals, we use two thresholds, θ_1 and θ_2 ($0<\theta_1<\theta_2<\beta_{max}$). If $\beta(t)>\theta_2$, we do not reduce the video bitrate. This avoids overreacting to negative spikes in the avail-bw. When $\theta_1<\beta<\theta_2$, the buffer occupancy is considered low and we decrease the requested bitrate, by one profile at each chunk, as long as $\phi_1<\phi_{cur}$. This reaction is fast because it is based on the most recent throughput measurement (instead of the avail-bw running average). Finally, if $\beta<\theta_1$, the player switches to "panic mode" and requests the smallest available bitrate b_1.

On the other hand, the bitrate increase logic is as follows. If $\beta(t)>\theta_2$, the buffer size is *not* critically low and so we increase the requested video bitrate if the following two conditions are also met. First, we check whether during the last T seconds the candidate profile ϕ_2, which is based on the long-term throughput average, has been higher than ϕ_{cur}. Second, we check whether ϕ_1, which is based on the most recent throughput measurement, is also higher than ϕ_{cur}. These two conditions aim to avoid short-term avail-bw increases that can trigger unnecessary (and potentially annoying) video bitrate transitions. When both conditions are met, the flag *can-switch-up* is enabled. If $\beta(t)$ is between θ_1 and θ_2, we increase the video bitrate by one profile only if the candidate profile ϕ_1 is higher than ϕ_{cur}; this condition is typically true while the player is recovering from an earlier buffer size reduction.

In summary, the next requested video profile then is determined as follows:

```
if $\theta_2 < \beta(t) < \beta_{max}$ then
  if $\phi_2 > \phi_{cur}$ and can-switch-up is TRUE then
    Increase the requested bitrate by one profile.
  end if
els if $\theta_1 < \beta(t) < \theta_2$ then
  if $\phi_1 < \phi_{cur}$ then
    Decrease the requested bitrate by one profile
  els if $\phi_1 > \phi_{cur}$ then
    Increase the requested bitrate by one profile
  end if
  els if $\beta(t) < \theta_1$ then
  "Panic mode": switch to the lowest available profile.
end if
```

Note that we have limited the maximum requested bitrate difference between two successive chunks to one profile (unless if in "panic mode"). The goal is to avoid major video quality fluctuations that can be annoying to the user. Obviously, this constraint can make the player slower to converge to the appropriate candidate bitrate when the avail-bw changes. In the following experiments we set $T=15$ s, $\theta_1=10$ s, $\theta_2=20$ s.

6.2. Behavior under persistent avail-bw variations

In the following we present some representative results with the `AdapTech Streaming` player using the video ("Big Buck Bunny") provided by the Wowza Media server for Flash HTTP Streaming [18]:

```
http://184.72.239.149/vod/smil:bigbuckbunny.smil/
manifest.f4m
```

The manifest file declares four video encoding bitrates between 0.45 Mbps and 1.47 Mbps. Each video chunk except the last has the same duration of $\tau=3$ s. Note that in Adobe's HTTP Dynamic Streaming, the audio chunks accompany the video chunks and have the same duration.

Fig. 18 shows the various throughput related metrics in the case of persistent avail-bw variations. The playback buffer size for the same experiment is shown in Fig. 19. The experiment starts with 3 Mbps avail-bw. After at least 10 s worth of video has been buffered, the video playback starts and the player quickly switches to the highest available profile ($P_{1.47}$), increasing the requested bitrate by one profile at each chunk. The player switches to `Steady-State` when the playback buffer size reaches 30 s.

When the avail-bw drops to 1 Mbps at $t=90$ s, the playback buffer size starts decreasing until, at time $t=166$ s, it goes below 20 s. The player then reacts by requesting the next chunk at a lower profile ($P_{1.07}$) at $t=170$ s. By decreasing the requested profile and switching to `Buffering-State`, the player manages to build up its buffer before it goes much lower than 20 s. When the avail-bw increases back to 3 Mbps, it takes the player 10 s to detect that the increase is persistent (i.e., $\phi_2>\phi_{cur}$) and 15 s later, the player starts increasing the requested bitrate by one profile per chunk. A similar situation takes place when the avail-bw drops down for a second time.

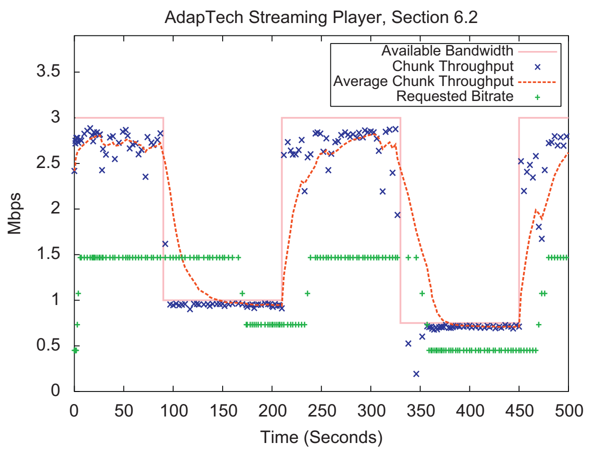

Fig. 18. Per-chunk throughput, average chunk throughput and the requested bitrate for the video traffic under persistent avail-bw variations.

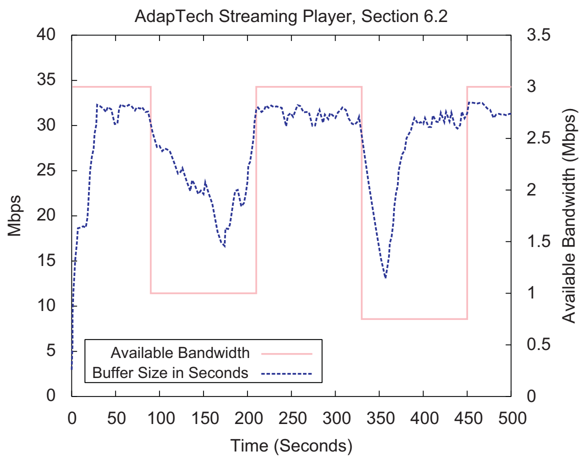

Fig. 19. Video playback buffer size in seconds under persistent avail-bw variations.

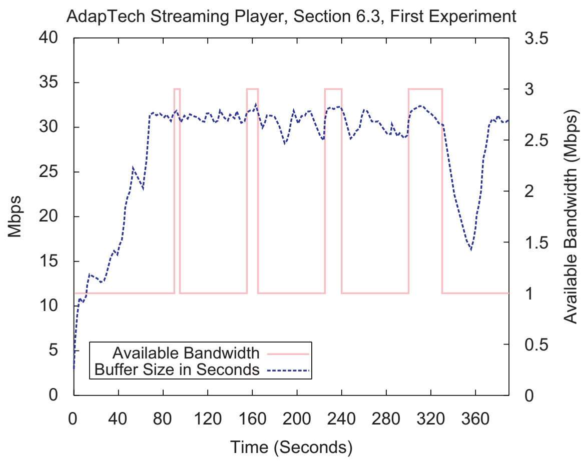

Fig. 21. Video playback buffer size in seconds under positive avail-bw spikes.

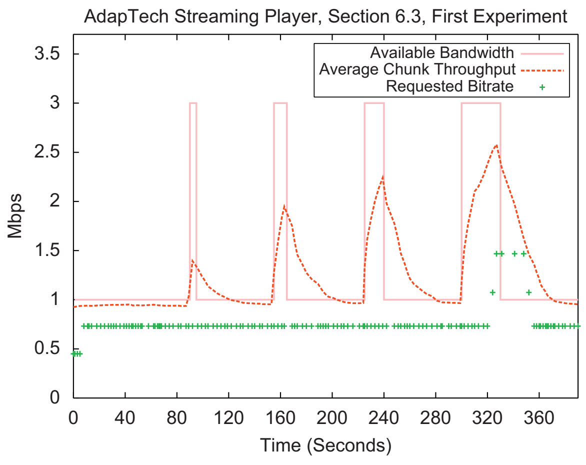

Fig. 20. Average chunk throughput and the requested bitrate for the video traffic under positive avail-bw spikes.

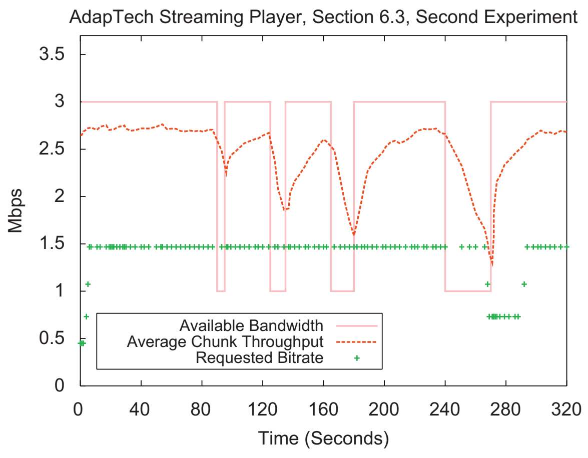

Fig. 22. Average chunk throughput and the requested bitrate for the video traffic under negative avail-bw spikes.

The difference is that the playback buffer size decreases much faster this time and so the player reacts much faster too.

6.3. *Behavior under short-term avail-bw variations*

In this section, we demonstrate how our player reacts to positive and negative spikes of different durations. Our focus here is to increase the duration of the spikes until we observe the player reacting. Fig. 20 shows how the player reacts to positive spikes in the avail-bw, while Fig. 21 shows the buffer size for the same experiment. The spikes last for 5, 10, 15, and 30 s and have a magnitude of 2 Mbps. The player does not react to the three shorter spikes but it reacts to the 30-sec spike by increasing the requested profile. The magnitude of the spike and the value of the parameter T determine the duration of the shortest spike that the player would react to. The value of T represents a tradeoff between reacting quickly to avail-bw increases and avoiding overreactions to short-term positive avail-bw spikes. In the case of $T=15$, the player would react to the spike at least 15 s after the latter started. This would require the spike to last for at least that long.

Fig. 22 shows how the player reacts to negative spikes in the avail-bw, while Fig. 23 shows the buffer size for the same experiment. The spikes last for 5, 10, 15, and 30 s. and have a magnitude of 2 Mbps. For the three shorter spikes, the playback buffer size never goes below $\theta_2=20$ s and the player does not react. The 30-sec spike is long enough to reduce the playback buffer below 20 s. Once this happens, the player quickly starts reducing the requested profile. The value of θ_2 represents a tradeoff between reacting quickly to avail-bw decreases and avoiding overreactions to short-term negative avail-bw spikes.

7. Two competing players

Suppose that two adaptive HTTP streaming players share the same bottleneck. This can happen, for instance, when people in the same house watch two different

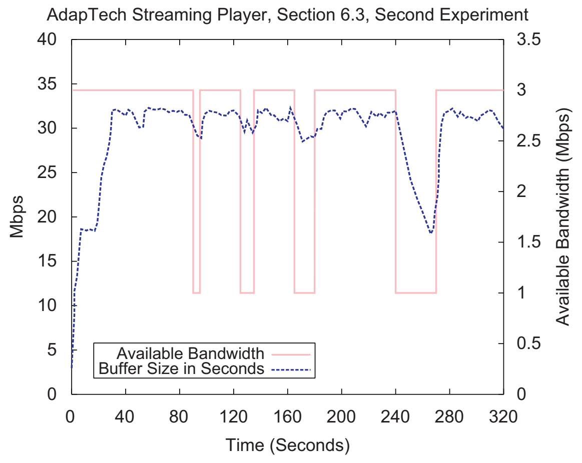

Fig. 23. Video playback buffer size in seconds under negative avail-bw spikes.

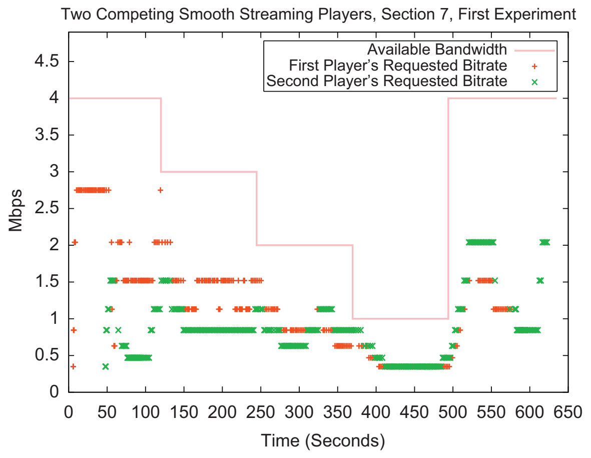

Fig. 24. Two Smooth Streaming players compete for avail-bw. The players start the playback at around $t=7$ s and $t=57$ s, respectively.

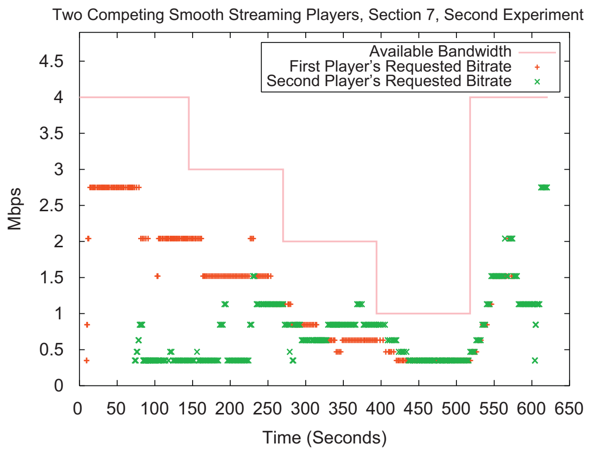

Fig. 25. Two Smooth Streaming players compete for avail-bw. The players start the playback at around $t=12$ s and $t=77$ s, respectively.

movies—in that case the shared bottleneck is probably the residential broadband access link. Another example of such competition is when a large number of users watch the same live event, say a football game. In that case the shared bottleneck may be an edge network link. There are many questions in this context. Can the players share the avail-bw in a stable manner, without experiencing oscillatory bitrate transitions? Can they share the avail-bw in a fair manner? How does the number of competing streams affect stability and fairness? How do different adaptive players compete with each other? And how does a player compete with TCP bulk transfers (including progressive video downloads)? In this section, we only "touch the surface" of these issues, considering a simple scenario in which two identical players (Smooth Streaming) compete at a bottleneck in which the avail-bw varies between 1 and 4 Mbps. The idea is that, if we observe significant problems even in this simple scenario, we should also expect similar issues in more complex scenarios.

In the following, we present results from two experiments. It should be noted that such experiments are fundamentally non-reproducible: there is always some stochasticity in the way players share the bottleneck's avail-bw. However, our observations, at a qualitative level, are consistent across several similar experiments.

Fig. 24 shows the avail-bw variations in the first experiment, together with the requested bitrates from the two players. The second player starts about 1 min after the first one. Until that point, the first player was using the highest profile ($P_{2.75}$). After the second player starts, the two players could have shared the 4 Mbps bottleneck by switching to $P_{1.52}$, however, they do not. Instead, the second player oscillates between lower profiles. When the avail-bw drops to 3 Mbps or 2 Mbps, the oscillations continue for both players. The only stable period during this experiment is when the avail-bw is limited to 1 Mbps: in that case both players switch to the lowest profile $P_{0.35}$ simply because there is no other bitrate that is sustainable for both players. Interestingly, when the avail-bw increases to 4 Mbps the two players start oscillating in a synchronized manner: when they both switch to $P_{2.04}$ the shared bottleneck is congested. It seems that both players observe congestion at the same time, and they react in the same manner lowering their requested bitrate at about the same time.

The previous experiment reveals some interesting points about Smooth Streaming. First, it seems that the avail-bw estimation method in that player considers only time periods in which chunks are actually downloaded—there is probably no estimation when the player's connections are idle. So, if two players X and Y share the same bottleneck, and Y is idle while X downloads some chunks, X can overestimate the avail-bw. Second, it appears that the Smooth player does not use randomization in the rate-adaptation logic. Previous studies have shown that a small degree of randomization was often sufficient to avoid synchronization and oscillations [7].

Fig. 25 shows the results for another run. Here, the second player stays at the lowest possible bitrate for about 150 s after it starts streaming, while the first player uses the remaining avail-bw with the highest sustainable bitrate. This is clearly a very unfair way to share the bottleneck link. It should be noted that this unfairness is *unrelated* to TCP's well-known unfairness toward connections with large round-trip times (RTT). In this experiment, both connections have the same RTT. The unfairness here is not generated by TCP's congestion control, but by the offered load that each application (video player) requests. The second player estimates the avail-bw to be much lower, and it does not even try to increase its requested bitrate. If it had done so, it would likely be able to obtain a higher bitrate forcing the first player to a lower bitrate. It appears, however, that the Smooth player's rate-adaptation algorithm does not include such bandwidth-sharing objectives. Our preliminary investigation in this paper indicates that there are possible issues with adaptive streaming when multiple players compete for avail-bw. Further investigation would require a larger number of players (possibly of different types) and more control over the adaptive streaming system including the video content, servers, and routers. This however, is outside the scope of the current paper.

8. Smooth live streaming

We are also interested in the similarities and differences between live and on-demand adaptive video streaming. What is the playback delay in the case of live streaming? Does the player react differently to avail-bw variations when it streams live content? And how does the player react when the playback buffer becomes empty? Does it skip chunks so that it maintains a small playback delay, or does it increase the playback delay aiming to show all chunks? We explored these questions with the Smooth Streaming player. In the first experiment, we used the live video feed from the Home Shopping Network (HSN) web site:

`http://www.hsn.com/hsn-tv_at-4915_xa.aspx?nolnav=1.`

Fig. 26 shows the various throughput metrics and the requested video bitrate, while Fig. 27 shows the estimated video playback buffer size (in seconds). A first important difference with on-demand streaming is that the initial playback buffer size is about 8 s; significantly lower than the typical playback buffer sizes we observed in on-demand Smooth Streaming sessions. By the way, even though this playback delay may sound too large for live content, note that even cable TV networks usually enforce a similar delay (referred to as "profanity delay") to avoid broadcasting inappropriate scenes.

A second important point, which is not shown in the previous graphs but it can be observed by the timestamps of the HTTP requests, is that the player initially requests chunks that correspond to about 8 s in the past. This way, it can start displaying the video without having to wait for a long initial playback delay; of course, what the user watches then happened at least 8 s ago. As in the case of on-demand streaming, the initial chunk request rate (while the player is in the `Buffering-State`) is higher, requesting a new chunk as soon as the last chunk is received.

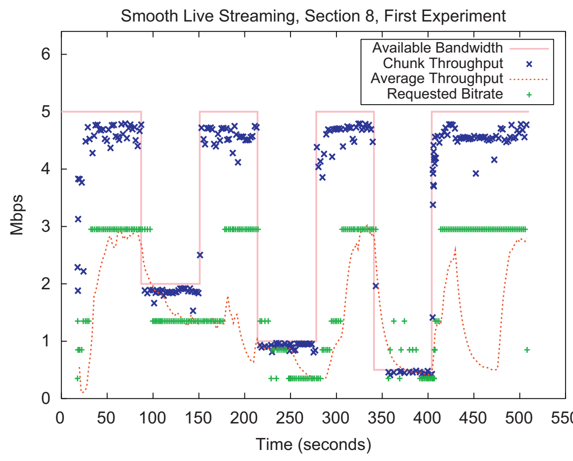

Fig. 26. Per-chunk throughput, the requested bitrate for the video traffic and average TCP throughput for live video streaming. Playback starts at around $t=20$ s.

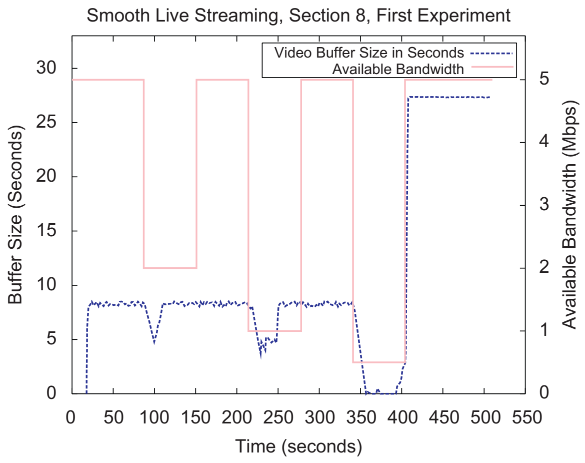

Fig. 27. Video playback buffer size in seconds for live video streaming.

Other than the previous two points, it appears that the Smooth Streaming player does not react to avail-bw variations any different with live content than with on-demand content. Note that the persistent avail-bw decreases and increases are followed by similar bitrate adjustments as in Section 3.

Another interesting point is what happens when the playback buffer becomes empty. This happens in this experiment at around $t=360$ s, when the avail-bw was decreases to 500 Kbps. During that event the playback buffer remaines practically empty for about 40 s. The player still receives some chunks during that period but they do not increase the playback buffer size by more than a chunk. The player is late to switch to a sufficiently lower bitrate, and so several chunks are requested at bitrates ($P_{1.40}$ and $P_{0.80}$) that are higher than the avail-bw. The end-result is that those chunks take too long to download, the buffer becomes depleted, and the playback stalls for about 27 s.

Arguably, it is reasonable to expect for live streaming that the player could skip some chunks that take too long to download, jumping to a later point in the video stream. The Smooth Streaming implementation for this particular Web site does not do so, however. It appears that the player aims to show every single chunk. Consequently, the playback delay can increase, gradually staying more and more behind the live broadcast. Indeed, in this case after the avail-bw increased to 5 Mbps, the playback buffer size increases to almost 27 s, which is comparable to the typical playback delay of on-demand content using Smooth Streaming.

We performed a similar experiment with another live video streaming provider, to examine if there are differences between servers. In this experiment we used the IIS Smooth Streaming Showcase Website:

```
http://www.microsoft.com/silverlight/
iis-smooth-streaming/demo/
```

It is interesting that the player behaves very differently when streaming live video from this server. Specifically, Fig. 28 shows the requested bitrate and avail-bw. The experiment starts with 5 Mbps avail-bw—we drop it to 10 Kbps at time $t=72$ s, effectively stopping the player from downloading any chunks for 60 s. After increasing the avail-bw back to 5 Mbps, the player has two options. It can either continue asking and displaying chunks sequentially—falling behind the live content by more than 60 s. Or, the player can skip around 60 s of content and keep the viewing experience “almost-live”.

The player continues requesting chunks sequentially at the lowest available bitrate until it downloads all chunks upto the most recent. In the meanwhile, the player starts also downloading chunks at higher bitrates based on to the current avail-bw. We have confirmed that the player decodes and presents these higher quality chunks. At the same time a `Go Live` button becomes active. We press the button at around time 180 s. The player immediately switches to live content, displaying the lower quality chunks. It also stops downloading the older chunks, and starts increasing the requested bitrate of the live chunks

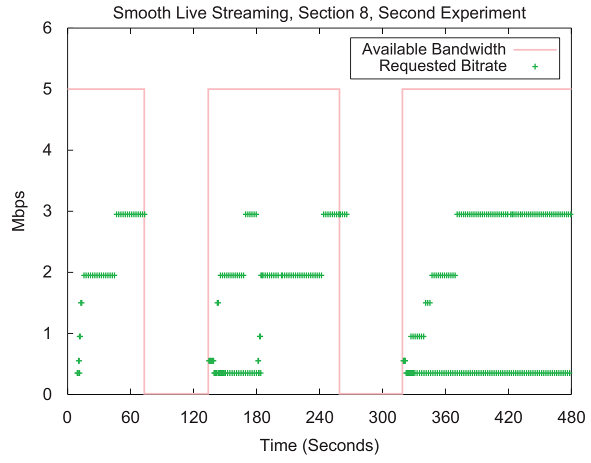

Fig. 28. The requested bitrate for the video traffic, Smooth Live Streaming Showcase Website.

over time. Similarly, in the interval $t=259$ s to $t=319$ s, we drop the avail-bw to 10 Kbps. This time, after increasing the avail-bw, we do not press the `Go Live` button. The experiment shows that the player continues asking for live chunks at the lowest available bitrate.

9. Related work

Even though there is extensive previous work on rate-adaptive video streaming over UDP, transport of rate-adaptive video streaming over TCP, and HTTP in particular, presents unique challenges and has not been studied so much in the past.

A good overview of multi-bitrate video streaming over HTTP is given by Zambelli [26], focusing on Microsoft’s IIS Smooth Streaming. Begen et al. [2] provide an overview of video streaming technologies over the Web including adaptive streaming over HTTP. Stockhammer [23] provides an overview of the Dynamic Adaptive Streaming over HTTP (DASH) specifications (available from 3GPP and in draft version from MPEG).

Recently, some related work aims to evaluate the performance of video streaming systems over HTTP and TCP. Cicco et al. [3] investigate the performance of the Akamai HD Network for Dynamic Streaming for Flash over HTTP. Kuschnig et al. [9] evaluate and compare three server-side rate-control algorithms for adaptive TCP streaming of H.264/SVC video. The same authors [11] have presented an experimental analysis of the use of multiple parallel HTTP-based request-response streams for video streaming. Sánchez et al. [21] show the benefits of using the Scalable Video Coding (SVC) in a DASH environment. Wang et al. [25] have developed discrete-time Markov models to investigate the performance of TCP for both live and stored media streaming. They show that TCP provides good streaming performance when the achievable TCP throughput is roughly twice the media bitrate, with only a few seconds of startup delay. Goel et al. [8] show that the latency at the application layer, which occurs as a result of throughput-optimized TCP implementations, can be minimized by dynamically tuning TCP’s send buffer.

There is also some recent work proposing new rate adaptation algorithms for adaptive streaming over HTTP, or improving existing players. Kuschnig et al. have proposed a receiver-driven transport mechanism that uses multiple HTTP streams and different priorities for certain parts of the media stream [10]. Tullimas et al. [24] propose a receiver-driven TCP-based method for video streaming over the Internet, called *MultiTCP*, aimed at providing resilience against short-term bandwidth fluctuations and controlling the sending rate by using multiple TCP connections. Evensen et al. [6] present a client-side request scheduler that distributes requests over multiple heterogeneous interfaces simultaneously. Liu et al. [12] present a rate adaptation algorithm that deploys a step-wise increase or aggressive-decrease method. Cicco et al. [5] propose a Quality Adaptation Controller (QAC) for live adaptive video streaming designed based on control theory and they compare their scheme with Akamai’s adaptive video streaming. Qiu et al. [19] use an optimization algorithm called Intelligent Bitrate Switching based Adaptive Video Streaming (ISAVS) for HTTP adaptive streaming. Schierl et al. [22] present the use of Priority based Media Delivery (PMD) for SVC to overcome link interruptions and channel bitrate reductions in mobile networks.

10. Conclusions

We conducted an experimental evaluation of two commercial and one open source adaptive HTTP streaming players, focusing on how they react to persistent and short-term avail-bw variations. We found major differences in the players and significant inefficiencies in each of them. Aiming at addressing those issues, we also proposed a new rate adaptation algorithm called `AdapTech Streaming`. In the following, we conclude by giving a summary of our findings for each player.

The Smooth Streaming player is quite effective under unrestricted avail-bw as well as under persistent avail-bw variations. It quickly converges to the highest sustainable bitrate, while it accumulates at the same time a large playback buffer requesting new chunks (sequentially) at the highest possible bitrate. This player is rather conservative in its bitrate switching decisions. First, it estimates the avail-bw by smoothing the per-chunk throughput measurements, introducing significant delays in the rate-adaptation logic. Second, it avoids large bitrate changes that could be annoying to the viewer. On the negative side, the Smooth Streaming player reacts to short-term avail-bw spikes too late and for too long, causing either sudden drops in the playback buffer or unnecessary bitrate reductions. Further, our experiments with two competing Smooth Streaming players indicate that the rate-adaptation logic is not able to avoid oscillations, and it does not aim to reduce unfairness in bandwidth sharing. The Live Smooth Streaming player behaves similarly, except that the playback buffer is initially shorter and the player starts requesting chunks from the recent past. Interestingly, we observed major differences in the behavior of this player across different servers. In some servers, the user is given the option to `Go Live`, jumping to an almost-live point in the stream.

The Netflix player is similar to Smooth Streaming (they both use Silverlight for the media representation). However, we observed that the former showed some important differences in its rate-adaptation behavior, becoming more aggressive than the latter and aiming to provide the highest possible video quality, even at the expense of additional bitrate changes. Specifically, the Netflix player accumulates a very large buffer (up to few minutes), it downloads large chunks of audio in advance of the video stream, and it occasionally switches to higher bitrates than the avail-bw as long as the playback buffer is almost full. It shares, however, the previous shortcomings of Smooth Streaming.

The OSMF player often fails to converge to an appropriate bitrate even after the avail-bw has stabilized. This player has been made available so that developers will customize the code including the rate-adaptation algorithm for HTTP Dynamic Streaming for their use cases. We do not summarize any other experiment here.

The goal of the `AdapTech Streaming` player is to address the observed issues with the three previous players, focusing on a single instance of the player streaming on-demand content. As such, the main design objectives are to detect persistent and short-term avail-bw variations in a timely manner in order to provide smooth bitrate transitions and avoid video freezes. We show that the player is able to achieve the stated goals by maintaining a short-term and a long-term estimate of the avail-bw while making different decisions based on the buffer size.

Overall, it is clear that the existing adaptive HTTP streaming players are still at their infancy. The technology is new and it is still not clear how to design an effective rate-adaptation logic for a complex and demanding application (video streaming) that has to function on top of a complex transport protocol (TCP). The interactions between these two feedback loops (rate-adaptation logic at the application layer and TCP congestion control at the transport layer) are not yet understood well.

References

[1] S. Akhshabi, A.C. Begen, C. Dovrolis, An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http, ACM MMSys, 2011.
[2] A.C. Begen, T. Akgul, M. Baugher, Watching video over the web, part I: streaming protocols, IEEE Internet Computing 15 (2) (2011) 54–63.
[3] L. De Cicco, S. Mascolo, An experimental investigation of the Akamai adaptive video streaming, in: Proceedings of USAB WIMA, 2010.
[4] Tamper Data, ⟨https://addons.mozilla.org/en-US/firefox/addon/tamper-data/⟩.
[5] L. De Cicco, S. Mascolo, V. Palmisano, Feedback control for adaptive live video streaming, ACM MMSys, 2011.
[6] K. Evensen, D. Kaspar, C. Griwodz, P. Halvorsen, A. Hansen, P. Engelstad, Improving the performance of quality-adaptive video streaming over multiple heterogeneous access networks, ACM MMSys, 2011.
[7] R. Gao, C. Dovrolis, E. Zegura, Avoiding oscillations due to intelligent route control systems, in: Proceedings of IEEE INFOCOM, 2006.
[8] A. Goel, C. Krasic, J. Walpole, Low-latency adaptive streaming over TCP, ACM TOMCCAP 4 (3) (2008) 1–20.
[9] R. Kuschnig, I. Kofler, H. Hellwagner, An evaluation of TCP-based rate-control algorithms for adaptive Internet streaming of H.264/SVC, in: Proceedings of ACM MMSys, 2010.
[10] R. Kuschnig, I. Kofler, H. Hellwagner, Improving Internet video streamilng performance by parallel TCP-based request-response streams, in: Proceeidngs of IEEE CCNC, 2010.
[11] R. Kuschnig, I. Kofler, H. Hellwagner, Evaluation of http-based request-response streams for internet video streaming, ACM MMSys, 2011.
[12] C. Liu, I. Bouazizi, M. Gabbouj, Rate adaptation for adaptive http streaming, ACM MMSys, 2011.
[13] Pomelo LLC, Analysis of Netflix's security framework for 'Watch Instantly' service, Pomelo, LLC Tech Memo, 2009, ⟨http://pomelollc.files.wordpress.com/2009/04/pomelo-tech-report-netflix.pdf⟩.
[14] A. Orebaugh, G. Ramirez, J. Burke, J. Beale, Wireshark and Ethereal Network Protocol Analyzer Toolkit, Syngress Media Inc, 2007.
[15] Netflix Player, ⟨https://www.netflix.com⟩.
[16] OSMF Player, ⟨http://www.osmf.org⟩.
[17] Smooth Streaming Player, ⟨http://www.iis.net/download/SmoothClient⟩.
[18] Wowza Media Systems Flash HTTP Streaming preview, ⟨http://www.wowzamedia.com/forums/content.php?111-Flash-HTTP-Streaming-preview-AddOn-package⟩.
[19] X. Qiu, H. Liu, D. Li, S. Zhang, D. Ghosal, B. Mukherjee, Optimizing http-based adaptive video streaming for wireless access networks, IEEE IC-BNMT, 2010.
[20] L. Rizzo, Dummynet: a simple approach to the evaluation of network protocols, SIGCOMM CCR 27 (1) (1997) 31–41.

[21] Y. Sánchez de la Fuente, T. Schierl, C. Hellge, T. Wiegand, D. Hong, D. De Vleeschauwer, W. Van Leekwijck, Y. Le Louédec, idash: improved dynamic adaptive streaming over http using scalable video coding, ACM MMSys, 2011.
[22] T. Schierl, Y. Sanchez de la Fuente, R. Globisch, C. Hellge, T. Wiegand, Priority-based media delivery using svc with rtp and http streaming, Multimedia Tools and Applications (2010) 1–20.
[23] T. Stockhammer, Dynamic adaptive streaming over http—standards and design principles, ACM MMSys, 2011.
[24] S. Tullimas, T. Nguyen, R. Edgecomb, S.-C. Cheung, Multimedia streaming using multiple TCP connections, ACM TOMCCAP 4 (2) (2008) 1–20.
[25] B. Wang, J. Kurose, P. Shenoy, D. Towsley, Multimedia streaming via TCP: an analytic performance study, ACM TOMCCAP 4 (2) (2008) 1–22.
[26] A. Zambelli, IIS smooth streaming technical overview, Microsoft Corporation, 2009, 〈http://download.microsoft.com/download/4/2/4/4247C3AA-7105-4764-A8F9-321CB6C765EB/IIS_Smooth_Streaming_Technical_Overview.pdf〉.