ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/221636668

An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP

Conference Paper · February 2011

DOI: 10.1145/1943552.1943574 · Source: DBLP

CITATIONS
557

READS
853

3 authors, including:

Ali C. Begen
Ozyegin University
133 PUBLICATIONS **3,972** CITATIONS

SEE PROFILE

Constantine Dovrolis
Georgia Institute of Technology
157 PUBLICATIONS **11,101** CITATIONS

SEE PROFILE

All content following this page was uploaded by Constantine Dovrolis on 11 January 2015.

The user has requested enhancement of the downloaded file.

An Experimental Evaluation of Rate-Adaptation Algorithms in Adaptive Streaming over HTTP

Saamer Akhshabi
College of Computing
Georgia Institute of Technology
sakhshab@cc.gatech.edu

Ali C. Begen
Video and Content Platforms Research and Advanced Development
Cisco Systems
abegen@cisco.com

Constantine Dovrolis
College of Computing
Georgia Institute of Technology
dovrolis@cc.gatech.edu

ABSTRACT

Adaptive (video) streaming over HTTP is gradually being adopted, as it offers significant advantages in terms of both user-perceived quality and resource utilization for content and network service providers. In this paper, we focus on the rate-adaptation mechanisms of adaptive streaming and experimentally evaluate two major commercial players (Smooth Streaming, Netflix) and one open source player (OSMF). Our experiments cover three important operating conditions. First, how does an adaptive video player react to either persistent or short-term changes in the underlying network available bandwidth? Can the player quickly converge to the maximum sustainable bitrate? Second, what happens when two adaptive video players compete for available bandwidth in the bottleneck link? Can they share the resources in a stable and fair manner? And third, how does adaptive streaming perform with live content? Is the player able to sustain a short playback delay? We identify major differences between the three players, and significant inefficiencies in each of them.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems

General Terms

Performance, Measurement, Algorithms

Keywords

Experimental evaluation, adaptive streaming, rate-adaptation algorithm, video streaming over HTTP

1. INTRODUCTION

Video has long been viewed as the "next killer application". 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 three to four 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 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 recent players, such as Microsoft's Smooth

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MMSys'11, February 23–25, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0517-4/11/02 ...$10.00.

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 maintains multiple profiles of the same video, encoded in different bitrates and quality levels. Further, the video object is partitioned in *fragments*, typically a few seconds long. A player can then request different fragments at different encoding bitrates, depending on the underlying network conditions. Notice that it is the player that decides what bitrate to request for any fragment, 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 fragments 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 towards understanding and improving such video streaming mechanisms, this paper experimentally evaluates two commercial adaptive video players over HTTP (Microsoft's Smooth Streaming and the player used by Netflix) and one open source player (Adobe OSMF). Our experiments cover three important operating conditions. First, how does an adaptive video player react to either persistent or short-term changes in the underlying network available bandwidth? Can the player quickly converge to the maximum sustainable bitrate? Second, 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 third, how does adaptive streaming perform with live content? Is the player able to sustain a short playback delay? We identify major differences between the three players, and significant inefficiencies in each of them.

1.1 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 in depth in the past. A good overview of multi-bitrate video streaming over HTTP was given by Zambelli [17], focusing on Microsoft's IIS Smooth Streaming. Adobe has provided an overview of HTTP Dynamic Streaming on the Adobe Flash platform [1]. Cicco et al. [3] experimentally investigated the performance of the Akamai HD Network for Dynamic Streaming for Flash over HTTP. They studied how the player reacted to abrupt changes in the available bandwidth and how it shared the network bottleneck with a greedy TCP flow. Kuschnig et al. [9] evaluated and compared three server-side rate-control algorithms for adaptive TCP streaming of H.264/SVC video. The same authors have proposed a receiver-driven transport mechanism that uses multiple HTTP streams and different priorities for certain parts of the media stream [10]. The end-result is to reduce throughput fluctuations, and thus, improve video streaming over TCP. Tullimas et al. [15] also proposed 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. Hsiao et al. [8] proposed a method called Receiver-based Delay Control (RDC) to avoid congestion by delaying TCP ACK generation at the receiver based on notifications from routers. Wang et al. [16] developed discrete-time Markov models to investigate the performance of TCP for both live and stored media streaming. Their models provide guidelines indicating the circumstances under which TCP streaming leads to satisfactory performance. For instance, 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. [7] showed that the latency at the application layer, which occurs as a result of throughput-optimized TCP implementations, could be minimized by dynamically tuning TCP's send buffer. They developed an adaptive buffer-size tuning technique that aimed at reducing this latency. Feng et. al [5] proposed and evaluated a priority-based technique for the delivery of compressed prerecorded video streams across best-effort networks. This technique uses a multi-level priority queue in conjunction with a delivery window to smooth the video frame rate, while allowing it to adapt to changing network conditions. Prangl et al. [13] proposed and evaluated a TCP-based perceptual QoS improvement mechanism. Their approach is based on media content adaptation (transcoding), applied at the application layer at the server. Deshpande [4] proposed an approach that allowed the player to employ single or multiple concurrent HTTP connections to receive streaming media and switch between the connections dynamically.

1.2 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, 4 and 5 focus on the Smooth Streaming, Netflix, and OSMF players, respectively. Section 6 focuses on the competition effects that take place when two adaptive players share the same bottleneck. Section 7 focuses on live video using the Smooth Streaming player. We summarize what we learn for each player and conclude the paper in Section 8.

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 [12]) and a network emulator (DummyNet [14]). 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 various 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 two seconds.
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 i'th 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) fragment throughput* refers to the download throughput for a particular fragment, i.e., the size of that fragment divided by the corresponding download duration. Note that, if a fragment is downloaded in every two seconds, the fragment 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 fragment 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 fragment in the stream) in their HTTP fragment 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 fragment is sent only after the previous fragment 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. 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 (0x5C4) 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 fragment. After the player has received the manifest file, it generates successive HTTP requests for audio and video fragments. 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 fragment. This timestamp is determined using the per-fragment 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 fragment 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 fragments. 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 fragments, over the course of streaming, both connections get the chance to transfer video fragments. 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 fragments. 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 fragment (except the

last) has the same duration: τ=2 s. The audio fragments are approximately of the same duration.

3.1 Behavior under Unrestricted avail-bw

Figure 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 seconds, climbs to the highest encoding bitrate. As the per-fragment 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.

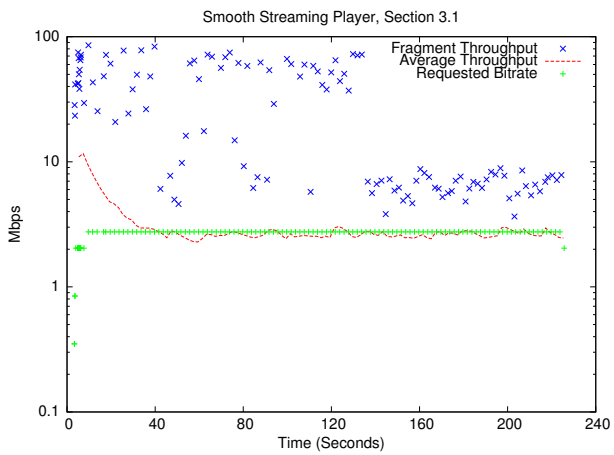

Figure 1: Per-fragment throughput, average TCP throughput and the requested bitrate for video traffic under unrestricted avail-bw conditions. Playback starts at around t=5 s, almost 2 s after the user clicked PLAY**.**

Another important observation is that during the initial time period, the player asks for video fragments much more frequently than once every τ seconds. Further analysis of the per-fragment 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 fragment as soon as the previous fragment was downloaded. Note that the player does *not* use HTTP pipelining - it does not request a fragment if the previous fragment has not been fully received. In `Steady-State`, on the other hand, the player requests a new fragment either τ seconds after the previous fragment was requested (if it took less than τ seconds to download that fragment) or as soon as the previous fragment was received (otherwise). In other words, in the `Buffering` state the player aims to maximize its fragment 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 fragment every τ seconds (recall that each fragment corresponds to τ seconds of content). We estimated the target video playback buffer size, as described in Section 2, to be about 30 seconds. 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-second playback buffer. 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.

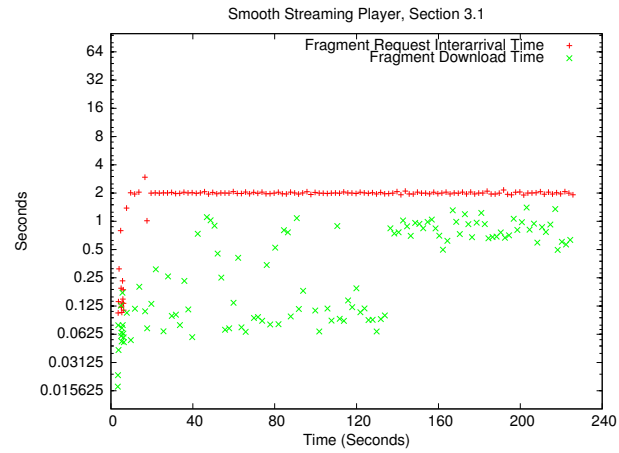

Figure 2: Interarrival and download times of video fragments under unrestricted avail-bw conditions. Each fragment is two seconds long.

3.2 Behavior of the Audio Stream

Audio fragments are of the same duration with video fragments, at least in the movies we experimented with. Even though audio fragments are much smaller in bytes than video fragments, 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 fragments more frequently than video fragments (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 Figure 3. First, note that, as expected, the per-fragment throughput is never higher than the avail-bw. Instead, the per-fragment throughput tracks quite well the avail-bw variations for most of the time; part of the avail-bw, however, is consumed by audio fragments 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 Figure

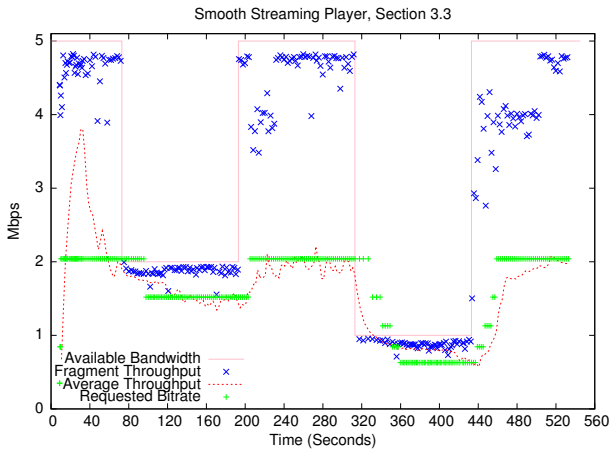

Figure 3: Per-fragment throughput, average TCP throughput and the requested bitrate for the video traffic under persistent avail-bw variations. Playback starts at around t=10 s, almost 3 s after the user clicked PLAY**.**

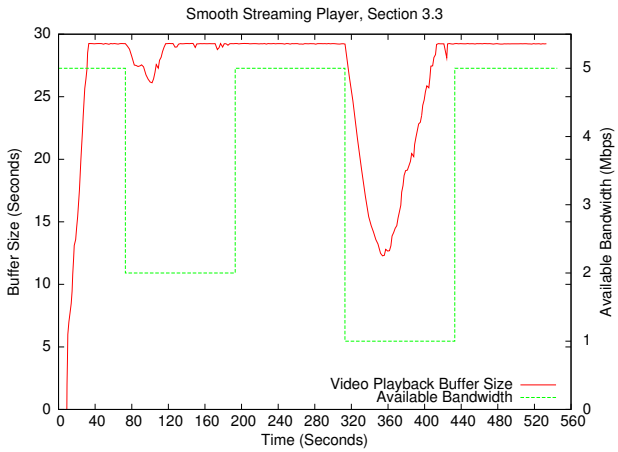

Figure 4: Video playback buffer size in seconds under persistent avail-bw variations.

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 seconds). During that time period, the playback buffer has decreased by only 3 seconds (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-fragment throughput measurements. Instead, it averages those per-fragment 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-fragment throughput measurements) instead of acting opportunistically based on the latest fragment 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.

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 Figures 5 and 7. 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.

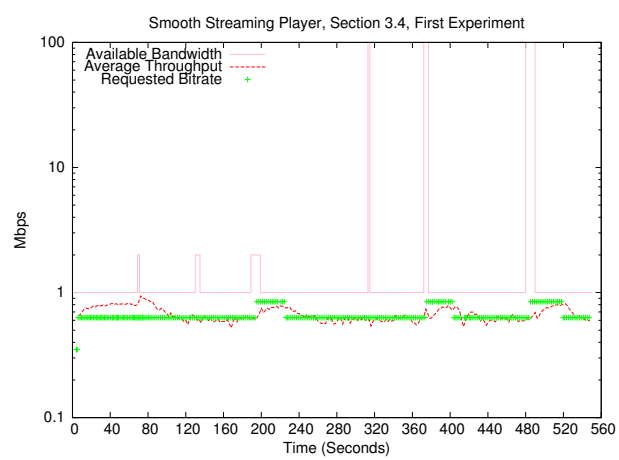

Figure 5: Average TCP throughput and the requested bitrate for the video traffic under positive avail-bw spikes. Playback starts at around t=7 s, almost 4 s after the user clicked PLAY**.**

Figure 5 shows the case of positive spikes. Here, we repeat the three spikes twice, each time with a different

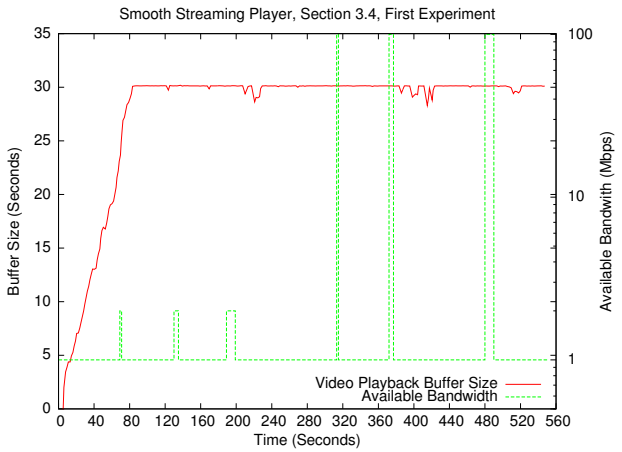

Figure 6: Video playback buffer size in seconds under positive avail-bw spikes.

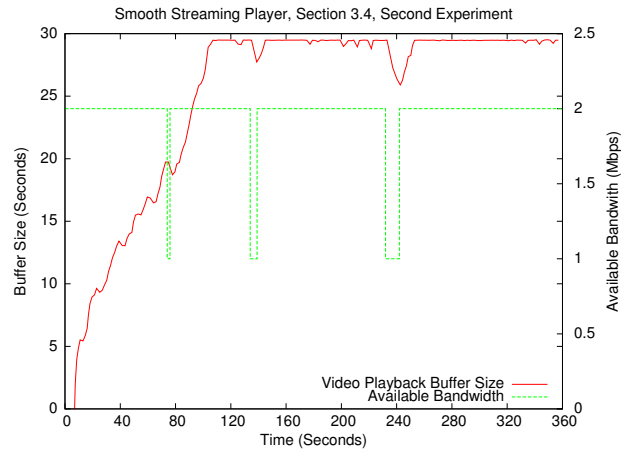

Figure 8: Video playback buffer size in seconds under negative avail-bw spikes.

increase magnitude. The Smooth Streaming player ignores the 2-second spikes and the smaller 5-second spike. On the other hand, it reacts to the 10-second 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 the time periods that the requested bitrate is higher than the avail-bw, the playback buffer size obviously shrinks, making the player more vulnerable to freeze events (See Figure 6). This experiment confirms that the player reacts, not to the latest fragment download throughput, but to a smoothed estimate of those measurements that can be unrelated to the current avail-bw conditions.

Figures 7 and 8 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.

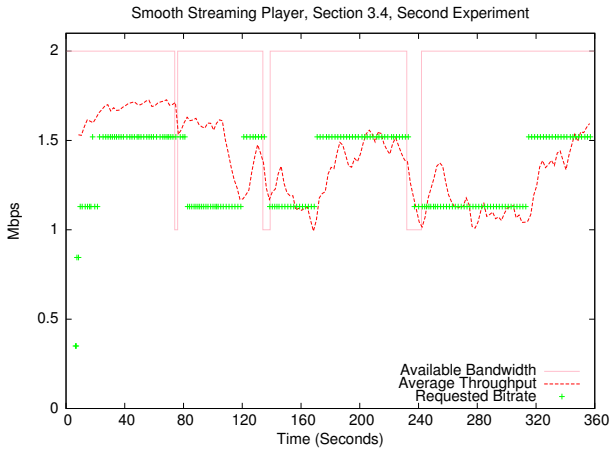

Figure 7: Average TCP throughput and the requested bitrate for the video traffic under negative avail-bw spikes. Playback starts at around t=9 s, almost 3 s after the user clicked PLAY.

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 fragment 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 that accesses the corresponding private key in Firefox. Video and audio fragments are delivered in wmv and wma formats, respectively. An example of a Netflix fragment 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 fragment is equal to the corresponding nominal bitrate for that fragment (e.g., a range of 8 Mb at the $P_{1.00}$ encoding profile corresponds to 8 seconds 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 fragments, 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

Figure 9 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 fragment requests and the download times for each video fragment are shown in Figure 10. In this experiment, the playback started about 13 s after the user 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 [11]. For the first few fragments, the player starts from the lowest encoding bitrate and requests a number of fragments from all the available bitrates. Then, the player stays at that highest bitrate for the duration of the experiment.

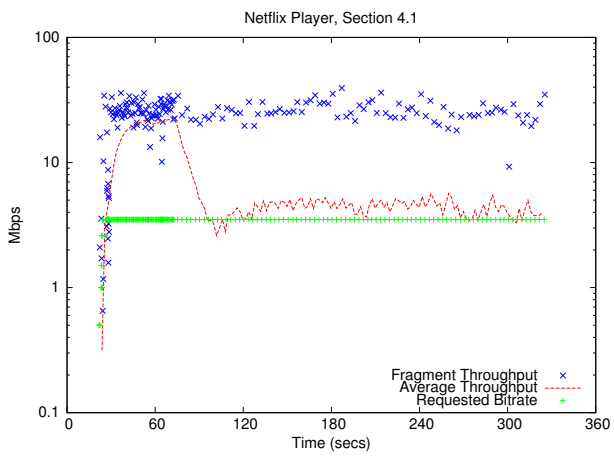

Figure 9: Per-fragment throughput, average TCP throughput and the requested bitrate for video traffic under unrestricted avail-bw conditions. Playback starts at around t=24 s, almost 16 s after the user clicked PLAY.

During the first 55 s of streaming, until t=75 s, the player is clearly in the `Buffering` state: it requests a new video fragment right after the previous fragment 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 fragments are requested almost every three seconds, with significant variation, however.

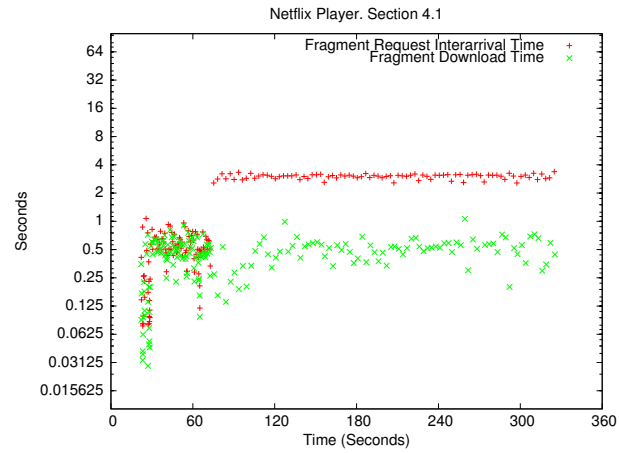

Figure 10: Interarrival and download times of video fragments under unrestricted avail-bw conditions.

4.2 Behavior of the Audio Stream

Audio fragments in the Netflix player are significantly larger than the ones in Smooth Streaming. Specifically, an audio fragment is typically 30 s long. Thus, after the player has reached `Steady-State`, a new audio fragment 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

Figure 11 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 fragments 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 fragment 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-fragment 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 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.

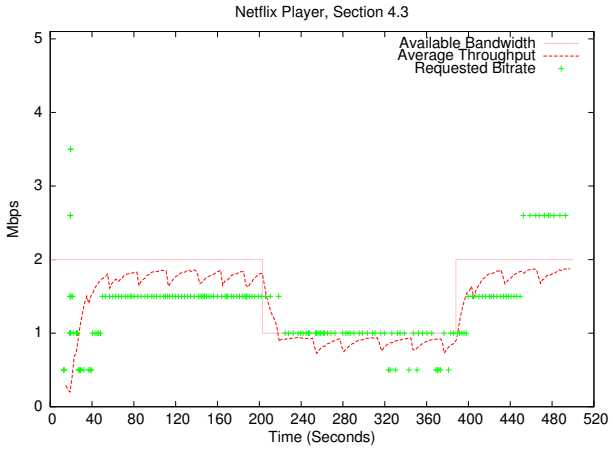

Figure 11: Average TCP throughput and the requested bitrate for the video traffic under persistent avail-bw variations. Playback starts at around t=20 s, almost 13 s after the user clicked PLAY**.**

4.4 Behavior under Short-term avail-bw Variations

Figure 12 shows how the Netflix player reacts to positive avail-bw spikes, while Figure 13 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.

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

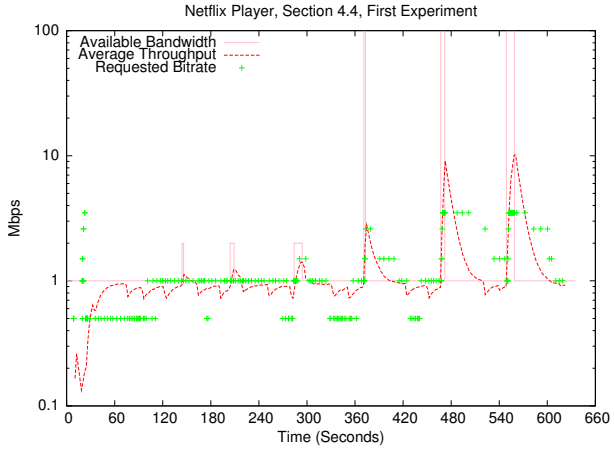

Figure 12: Average TCP throughput and the requested bitrate for the video traffic under positive avail-bw spikes. Playback starts at around t=20 s, almost 16 s after the user clicked PLAY**.**

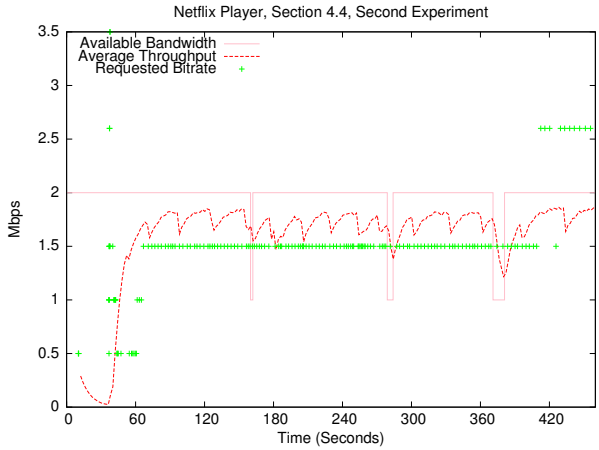

Figure 13: Average TCP throughput and the requested bitrate for the video traffic under negative avail-bw spikes. Playback starts at around t=37 s, almost 31 s after the user clicked PLAY**.**

player, each server file (F4F format) contains one segment of the movie. A segment can contain multiple fragments. The duration of each fragment is determined by the server. The HTTP requests that the player generates include a fragment 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 fragment numbers. The player maintains one TCP connection with the server and receives all fragments 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 seconds.

Figure 14 shows that initially the player requests one fragment 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.

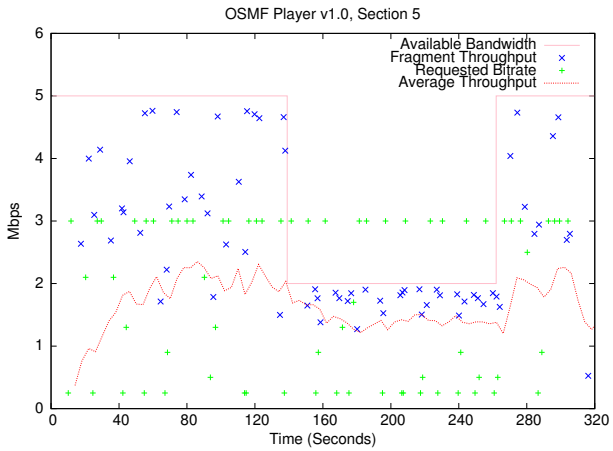

Figure 14: Per-fragment throughput, the requested bitrate for the video traffic and average TCP throughput under persistent avail-bw variations. Playback starts at around t=13 s, almost 3 s after the user clicked PLAY**.**

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/`

Figure 15 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.

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.

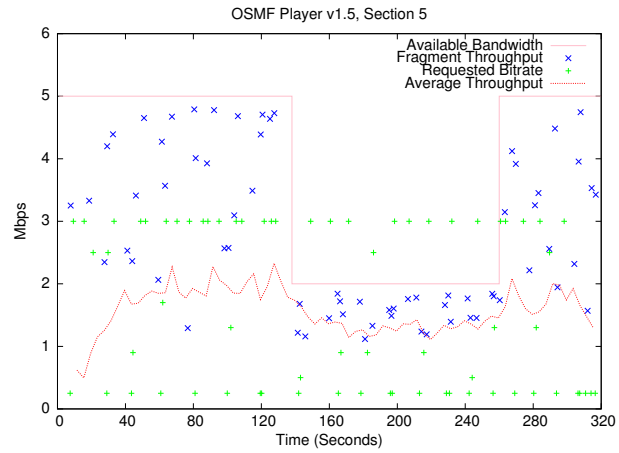

Figure 15: Per-fragment throughput, the requested bitrate for the video traffic and average TCP throughput under persistent avail-bw variations. Playback starts at around t=11 s, almost 4 s after the user clicked PLAY**.**

6. 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 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 1-4 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.

Figure 16 shows the avail-bw variations in the first experiment, together with the requested bitrates from the two players. The second player starts about one minute 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.

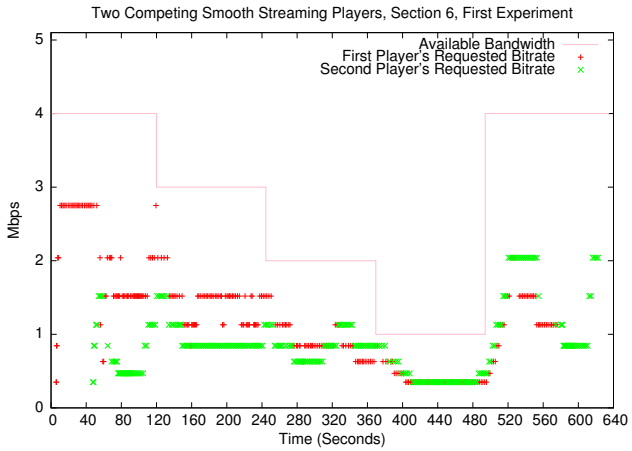

Figure 16: Two Smooth Streaming players compete for avail-bw. The players start the playback at around t=7 s and t=57 s, respectively.

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 fragments 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 fragments, 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 [6].

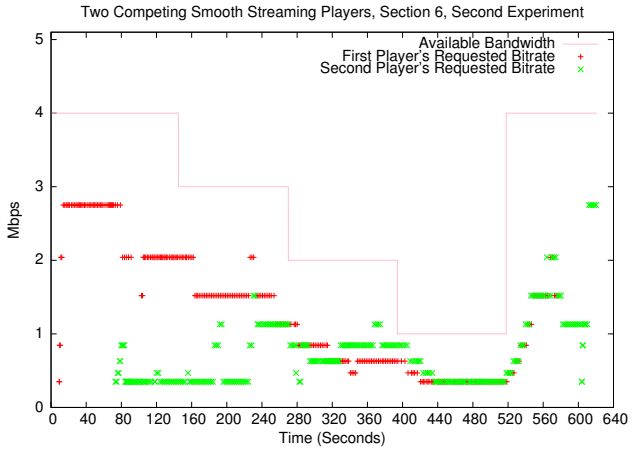

Figure 17: Two Smooth Streaming players compete for avail-bw. The players start the playback at around t=12 s and t=77 s, respectively.

Figure 17 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 towards 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.

7. 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 fragments so that it maintains a small playback delay, or does it increase the playback delay aiming to show all fragments? We explored these questions with the Smooth Streaming player. In the following experiments, 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`.

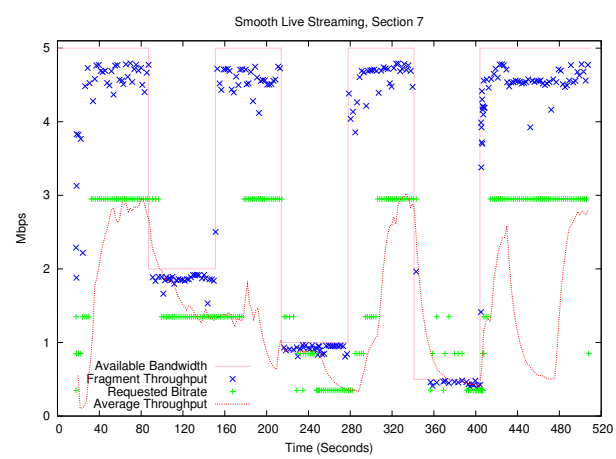

Figure 18: Per-fragment throughput, the requested bitrate for the video traffic and average TCP throughput for live video streaming. Playback starts at around t=20 s.

Figure 18 shows the various throughput metrics and the requested video bitrate, while Figure 19 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,

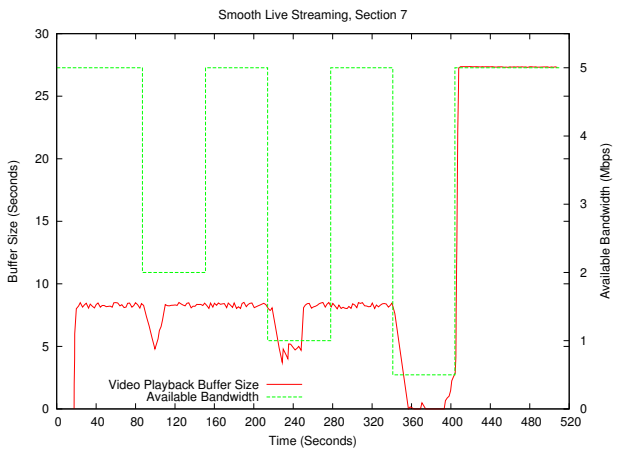

Figure 19: Video playback buffer size in seconds for live video streaming.

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 fragments 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 fragment request rate (while the player is in the `Buffering` state) is higher, requesting a new fragment as soon as the last fragment is received.

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 happened in this experiment at around t=360 s, when the avail-bw was decreased to 500 Kbps. During that event the playback buffer remained practically empty for about 40 s. The player still receives some fragments during that period but they would not increase the playback buffer size by more than a fragment. The player was late to switch to a sufficiently lower bitrate, and so several fragments were requested at bitrates ($P_{1.40}$ and $P_{0.80}$) that were higher than the avail-bw. The end-result was that those fragments took too long to download, the buffer became depleted, and the playback stalled for about 27 s.

Arguably, it is reasonable to expect for live streaming that the player could skip some fragments 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 fragment. 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 increased to almost 27 s, which is comparable to the typical playback delay of on-demand content using Smooth Streaming.

8. CONCLUSIONS

We conducted an experimental evaluation of three commercial adaptive HTTP streaming players, focusing on how they react to persistent and short-term avail-bw variations. Here, we summarize our findings for each player and conclude the paper.

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 fragments (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-fragment 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 fragments from the recent past.

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 case. We do not summarize any other experiment here.

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. In future work, we plan to focus on these issues and design improved rate-adaptation mechanisms.

9. REFERENCES

[1] Adobe. HTTP Dynamic Streaming on the Adobe Flash Platform. *Adobe Systems Incorporated*, 2010. http://www.adobe.com/products/httpdynamicstreaming/pdfs/httpdynamicstreaming_wp_ue.pdf.

[2] A. C. Begen, T. Akgul, and M. Baugher. Watching video over the Web, part I: streaming protocols. *To appear in IEEE Internet Comput.*, 2011.

[3] L. De Cicco and S. Mascolo. An Experimental Investigation of the Akamai Adaptive Video Streaming. In *Proc. of USAB WIMA*, 2010.

[4] S. Deshpande. Adaptive timeline aware client controlled HTTP streaming. In *Proc. of SPIE*, 2009.

[5] W. Feng, M. Liu, B. Krishnaswami, and A. Prabhudev. A priority-based technique for the best-effort delivery of stored video. In *Proc. of MMCN*, 1999.

[6] R. Gao, C. Dovrolis, and E. Zegura. Avoiding oscillations due to intelligent route control systems. In *Proc. of IEEE INFOCOM*, 2006.

[7] A. Goel, C. Krasic, and J. Walpole. Low-latency adaptive streaming over TCP. *ACM TOMCCAP*, 4(3):1–20, 2008.

[8] P.-H. Hsiao, H. T. Kung, and K.-S. Tan. Video over TCP with receiver-based delay control. In *Proc. of ACM NOSSDAV*, 2001.

[9] R. Kuschnig, I. Kofler, and H. Hellwagner. An evaluation of TCP-based rate-control algorithms for adaptive Internet streaming of H.264/SVC. In *Proc. of ACM MMSys*, 2010.

[10] R. Kuschnig, I. Kofler, and H. Hellwagner. Improving Internet video streamilng performance by parallel TCP-based request-response streams. In *Proc. of IEEE CCNC*, 2010.

[11] 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.

[12] A. Orebaugh, G. Ramirez, J. Burke, and J. Beale. *Wireshark and Ethereal network protocol analyzer toolkit*. Syngress Media Inc, 2007.

[13] M. Prangl, I. Kofler, and H. Hellwagner. Towards QoS improvements of TCP-based media delivery. In *Proc. of ICNS*, 2008.

[14] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM CCR*, 27(1):31–41, 1997.

[15] S. Tullimas, T. Nguyen, R. Edgecomb, and S.-C. Cheung. Multimedia streaming using multiple TCP connections. *ACM TOMCCAP*, 4(2):1–20, 2008.

[16] B. Wang, J. Kurose, P. Shenoy, and D. Towsley. Multimedia streaming via TCP: An analytic performance study. *ACM TOMCCAP*, 4(2):1–22, 2008.

[17] 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.

View publication stats