Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard

Te-Yuan Huang Nikhil Handigol Brandon Heller Nick McKeown Ramesh Johari
Stanford University
{huangty,nikhilh,brandonh,nickm,ramesh.johari}@stanford.edu

ABSTRACT

Today's commercial video streaming services use dynamic rate selection to provide a high-quality user experience. Most services host content on standard HTTP servers in CDNs, so rate selection must occur at the client. We measure three popular video streaming services – Hulu, Netflix, and Vudu – and find that accurate client-side bandwidth estimation above the HTTP layer is hard. As a result, rate selection based on inaccurate estimates can trigger a feedback loop, leading to undesirably variable and low-quality video. We call this phenomenon the *downward spiral effect*, and we measure it on all three services, present insights into its root causes, and validate initial solutions to prevent it.

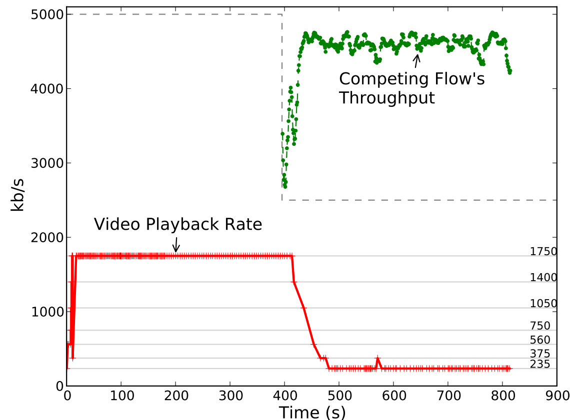


Figure 1: (Service A) A video starts streaming at 1.75Mb/s over a 5Mb/s network. After 395 seconds, a second flow starts (from the same server). The video could stream at 1.75Mb/s (given its fair share of 2.5Mb/s), but instead drops down to 235kb/s.

1. INTRODUCTION

Video streaming is a huge and growing fraction of Internet traffic, with Netflix and Youtube alone accounting for over 50% of the peak download traffic in the US [17]. Several big video streaming services run over HTTP and TCP (*e.g.* Hulu, Netflix, Vudu, YouTube) and stream data to the client from one or more third-party commercial CDNs (*e.g.* Akamai, Level3 or Limelight). Streaming over HTTP has several benefits: It is standardized across CDNs (allowing a portable video streaming service), it is well-established (which means the CDNs have already made sure service can reach through NATs to end-hosts), and cheap (the service is simple, commoditized and the CDNs compete on price). These benefits have made possible the huge growth in affordable, high-quality movie and TV streaming, for our viewing delight.

When video is streamed over HTTP, the video service provider relies on TCP to find the available bandwidth and choose a video rate accordingly. For example, if a client estimates that there is 1.5Mb/s available in the network, it might request the server to stream video compressed to 1.3Mb/s (or the highest video rate available at or below 1.5Mb/s). The video streaming service provider must walk a tightrope: If they pick a video rate that is *too high*, the viewer will experience annoying rebuffering events; if they pick a streaming rate that is *too low*, the viewer will experience poor video quality. In both cases, the experience degrades, and user may take their viewing elsewhere [9]. It is therefore important for a video streaming service to select the highest safe video rate.

This paper describes a measurement study of three popular HTTP-based video streaming services (Hulu, Netflix, and Vudu) to see how well they pick the video rate. According to the latest Consumer Reports [20], Netflix is the most popular video streaming provider, while Vudu is one of the most satisfying streaming services. Hulu is also popular for its TV content. Our results show that when the home network is quiet, all three do a good job and pick a rate close to the maximum possible given the available network bandwidth. But all three services do a poor job when there is competing traffic (*e.g.* from another video stream, a backup, or a large download). Figure 1 shows a typical example in which the video is streamed at only 1/7th of its fair share of the available network bandwidth.

We attribute the problem to the difficulty of selecting the best video rate from "above" HTTP. We will present evidence that all three video streaming services underestimate available bandwidth because of interactions between the video playback buffer, HTTP and TCP's congestion control algorithm. In some cases the cause is on-off scheduling when the playback buffer fills, causing TCP to re-enter slow-start (Section 5.1). In all cases, it appears that when the bit rate is reduced, the client has less accurate data upon which to estimate the available bandwidth. When bandwidth estimates are scant or noisy, it is perhaps safer to stream at a lower rate and avoid wreaking havoc in home networks. On the other hand, given how well TCP shares bandwidth among long-lived flows, the video service provider could safely send at a higher rate and rely on TCP to be a good neighbor.

We explore the bandwidth estimation problem in each of the three video services. Throughout the paper we refer to the services by the names A, B and C. All of our results can be reproduced by observing the services externally, so the data is not confidential. However, we refer to the services as A, B, and C to stress that the paper is not a comparison of the services; rather, we wish to show that any HTTP-based streaming service must wrestle with the problem of picking a streaming rate based on observations above HTTP.

Our measurements lead us to recommendations, which we demonstrate by trace-driven emulations of the video services: ensure that TCP achieves its fair share by using sufficiently large HTTP requests, take advantage of a large playback buffer (a client typically holds several minutes of video), do not focus on keeping the playback buffer full (this will naturally fall foul of TCP dynamics), be less conservative in trying higher video rates, and rely on TCP to prevent the video stream from taking more than its fair share.

In what comes next we describe how the three video services work, then demonstrate in Section 4 how badly they perform in the presence of a competing flow. In Section 5 we explain why each service picks a much lower video rate than necessary and explain how this condition depends on the interaction between HTTP, TCP and conservatism. We describe ways to pick rates more accurately in Section 6 and validate rate selection changes. In Section 7 we describe related work, then conclude in Section 8.

2. HOW VIDEO STREAMING WORKS

To understand why the video streaming services all suffer from the same problem, we first need to understand how they work. The information will later be useful for deploying a local proxy as well as for enabling our own emulator to download video segments from the CDNs directly.[1] We start out with the similarities before describing unique aspects to each. All three video streaming services share the following features:

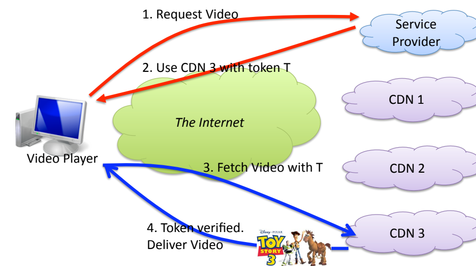


Figure 2: Common architecture of video streaming services over HTTP.

1. Video content is streamed over HTTP from third party CDN providers.
2. The video service runs on several different clients (*e.g.* web browser plugin, game console, TV).
3. Watching a video has two phases: authentication and streaming (see Figure 2).
4. Authentication (over HTTPS) : When a client requests a video, the service provider authenticates the user account and directs the client to a CDN holding the video. The video service provider tells the client what video streaming rates are available and issues a token for each rate.
5. Streaming: The client picks a video rate and requests a video by presenting a token (as a credential) to the designated CDN.
6. Picking a video rate: The client starts up with a pre-configured starting video rate. The client monitors the arriving traffic to pick the video rate. Because the service runs over "vanilla HTTP", the client makes the decision and the CDN server is not involved in picking the video rate.[2]

Although the three services are very similar, they differ from each other in some important ways.

Service A: Our measurements for Service A are based on a web-browser client. The Service A client sends HTTP "byte range" requests to the CDN, requesting four second chunks of video over a persistent TCP connection. Unless it switches to a new rate, the client reads the whole video from the same server. The client

[1]Even though we are able to download the video segments, only the legitimate player would be able to play the video.
[2]Services that control both the server and the client (*e.g.* YouTube) can involve the server and the client when picking a rate.

Provider	Platform	Download Strategy
Service A	Web Browser	Segment-by-segment download (Persistent connection)
Service B	Sony PlayStation 3	Segment-by-segment download (New connection)
Service C	Sony PlayStation 3	Progressive download (Open-ended download)

Table 1: Summary of the download strategies of the three services.

Provider	HTTP Request Format	Available Playback Rates (kb/s)
Service A	GET /*filename*/byte_range?token	235, 375, 560, 750, 1050, 1400, 1750(SD), 2350, 3600(HD)
Service B	GET /filename/clip_num?br=*bitrate*&token	650, 1000, 1500, 2000, 2500, 3200
Service C	GET /*filename*?token	SD: 1000, 1500, 2000 HD: 3000, 4500, 6750, 9000

Table 2: Summary of the available playback rates for each of the three services.

always starts out requesting the lowest video rate, continuously estimates the available bandwidth, and only picks a higher rate if it believes it can sustain it.

Service B: Service B's desktop client runs over a proprietary protocol, thus our measurements for Service B are based on its client on the Sony PlayStation 3 (PS3), which operates over HTTP. The Service B's PS3 client also requests one segment at a time, but each segment is stored as a seperate file and each request for a new segment is sent over a new TCP connection. Each request is for about eight seconds of video. The video rate is specified in the HTTP GET request; it starts at one of the two lowest playback rates, and steps up as it detects more bandwidth is available.

Service C: Our measurements for Service C are also based on its PS3 client, which has access to a broader range of video qualities compare to its desktop client. The Service C's PS3 client sends an open-ended HTTP request; *i.e.*, it requests the whole file in one go. To change video rate, the client must reset the TCP connection and request a new filename. While Service A controls the occupancy of the playback buffer by varying the rate at which they request new segments, Service B and Service C rely on the TCP receive window: when the playback buffer is full, TCP reduces the receive window to slow down the server.

Table 1 and Table 2 summarize the three services.

3. MEASUREMENT SETUP

In this section we describe our experimental setup and measurement approach. In particular, we describe how to control bottleneck bandwidth (including the use of a local proxy to eliminate network path variation); how to measure the playback rate; and the competing traffic we employ.

3.1 Bandwidth Control and Proxy

Our experiments measure the behavior of video streams from Services A, B and C when they compete with other TCP flows. To create a controlled environment, where we can set the bottleneck link rate, we make all the video streams pass through a rate limiter between the CDN and the client. Figure 3 shows how we place a NetFPGA machine [14] in-line to limit bandwidth. In most cases we limit the bandwidth to 5Mb/s, and the buffer size to 120kbit, sufficient to sustain 100% throughput with a 4–20 ms RTT. We use these configuration parameters throughout the paper, unless stated otherwise. The competing flow always passes through the NetFPGA too, and it shares the bottleneck link with the video stream.

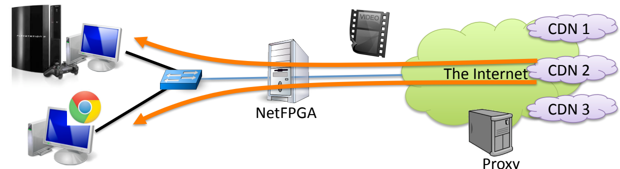


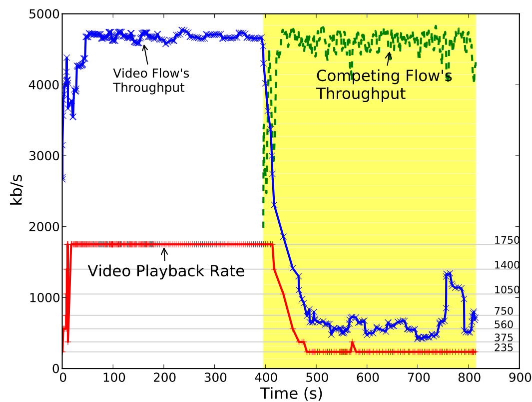
Figure 3: Our experimental setup includes a NetFPGA bandwidth limiter and a proxy.

In instances where we want a tightly controlled experiment we download videos from a local proxy instead of the CDN, so as to eliminate any variance caused by the location of the CDN server, or in the Internet paths.

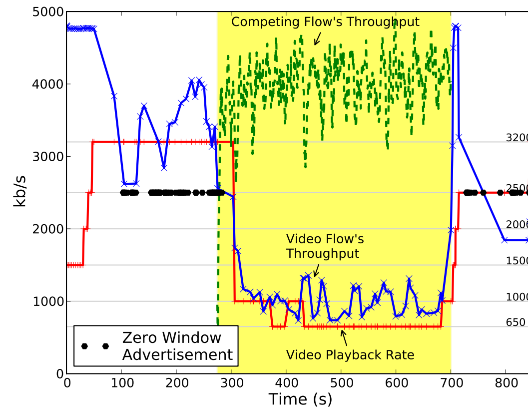
3.2 Measuring the Video Playback Rate

To understand the system dynamics, we need to know what video playback rate the client picks. Because this information is not externally visible, we have to deduce the mapping between the filenames requested by the client, and the playback rate of the video inside the file. We developed a different technique for each service.

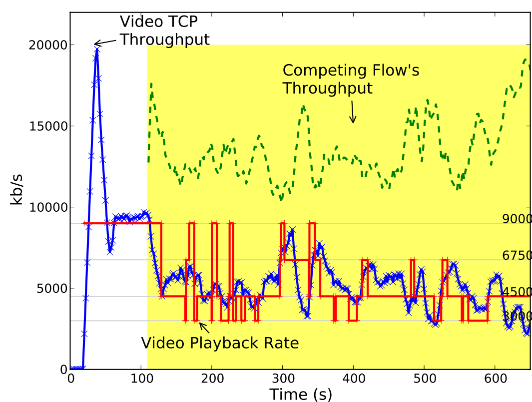
Service A: To figure out the mapping between filenames and the corresponding video rates, we first extract tokens from our traces and get the size of each file (via HTTP). We divide the file size by the duration of the video to get the rough video playback rate. The Service A client provides a debug window for users to monitor the current video playback rate, and we use this to validate our mapping.



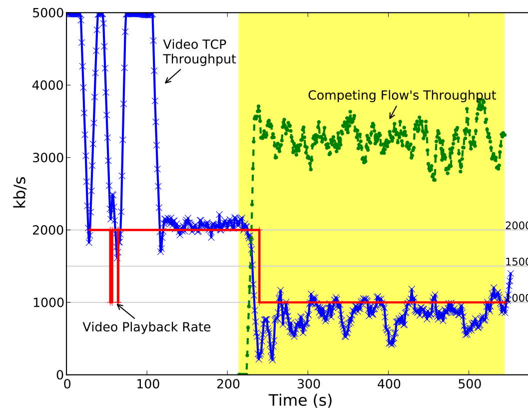
(a) Service A. Network bottleneck set to 5Mb/s. RTT from client to server is 20ms.



(b) Service B. Network bottleneck set to 5Mb/s. RTT from client to server is 20ms.



(c) Service C HD. Network bottleneck set to 22Mb/s.



(d) Service C SD. Network bottleneck set to 5Mb/s.

Figure 4: The downward spiral effect is visible in all three services.

Service B: Unfortunately, Service B doesn't provide a "debug" facility to validate the video playback rate. However, one of the parameters in the client's HTTP request seems to indicate the requesting video playback rate. But we needed to verify this. While the segment size requested with parameter 3200 kb/s is about 3.1 times larger than the segments with 1000 kb/s, the same relationship does not hold between other video rates. Instead, we use an indirect method of verification. We set the link bandwidth to a value slightly higher than each of the parameter values and see if the client converges to requesting with that parameter value. The value of the parameter indeed closely follows the bandwidth setting. For example, when we set the available bandwidth to 3,350 kb/s, the HTTP parameter converges to 3200. Similarly, when we set the bandwidth to 1,650 kb/s, the HTTP parameter converges to 1500.

Service C: Like Service A, Service C also has a mapping between the requested filename and the video playback rate. Unfortunately, Service C does not directly tell us the current rate. However, because the TCP flow is limited by the receive window when the playback buffer is full, the average TCP throughput matches the video playback rate. We confirmed that the converged receiver-limited TCP throughput reflects the rate information embedded in the requested filename.

The video rates available from each of the three services are summarized in Table 2; some playback rates may not be available for some videos.

3.3 The Competing Flows

The competing flow is a TCP flow doing a long file download. However, to eliminate any unfairness due to variations in network path properties, we make sure that the competing flow is served by the same CDN, and usually by the same server. For Service A and Service C, the competing flow is generated by an open-ended byte range request to the file with the highest rate. Further, we use the DNS cache to make sure that the competing flow comes from same termination point

(the server or the load-balancer) as the video flow. For Service B, since the files are stored as small segments, an open-ended request only creates short-lived flows. Instead, we generate the competing flow by requesting the Flash version of the same video stored in the same CDN, using rtmpdump [19] over TCP.

4. THE DOWNWARD SPIRAL EFFECT

All three services suffer from what we call the "downward spiral effect" – a dramatic anomalous drop in the video playback rate in the presence of a competing TCP flow. The problem is starkly visible in Figure 4. In all four graphs, the video stream starts out alone, and then competes with another TCP flow. As soon as the competing flow starts up, the client mysteriously picks a video playback rate that is far below the available bandwidth. Our goal is to understand why this happens.

To give us a first inkling into what is going on, we calculate the upper bound of what the client might *believe* the instantaneous available bandwidth to be, by measuring the arrival rate of the last chunk. Specifically, we calculate the throughput upper bound by having the size of a received video segment, divided by the time it took to arrive (the time from when the first byte arrived until the last byte arrived), which excludes the initial server response time. In all of the graphs, the video playback rate chosen by the client is quite strongly correlated with the calculated throughput. As we will see, herein lies the problem: If the client is selecting the video rate based on some function of the throughput it perceived, and the throughput is so different from the actual available bandwidth, then it is not surprising the client does such a poor job. Let's now see what goes wrong for each service in turn.

4.1 Service A

Fig. 4(a) shows the playback rate of a Service A video session, along with the client's perceived throughput over time. Starting out, the video stream is the only flow and the client requests the highest video rate (1750kb/s). The competing flow begins after 400 seconds; the video rate steadily drops until it reaches the lowest rate (235kb/s), and it stays there most of the time until the competing flow stops. In theory, both flows should be able to stream at 2.5Mb/s (their fair share of the link) and the client should continue to stream at 1750kb/s.

We repeated the experiment 76 times over four days. In 67 cases (91%) the downward spiral happens, and the client picks either the lowest rate, or bounces between the two or three lowest rates. In just seven cases (9%) was the client able to maintain a playback rate above 1400kb/s. To ensure accuracy and eliminate problems introduced by competing flows with different characteristics (*e.g.* TCP flows with different RTTs), we make the competing flow request for the *same* video file (encoded at 1750kb/s) from the *same* CDN. Unlike the video flow, the competing flow is just a simple TCP file download and its download speed is only dictated by TCP congestion control algorithm and not capped by the video client.[3]

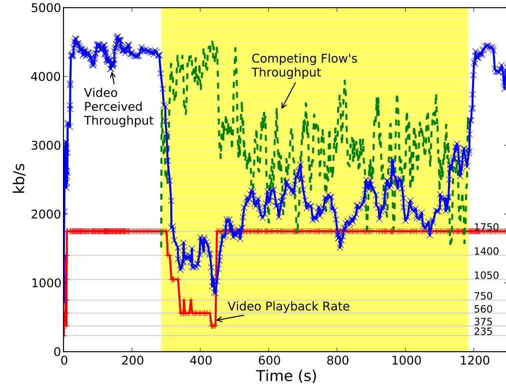


Figure 5: (Service A) The client manages to maintain the highest playback rate if we disable automatic rate selection.

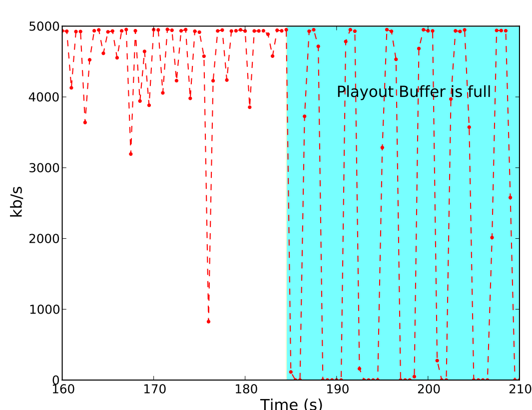
Why does throughput of the video flow drop so much below available fair-share bandwidth? Is it an inherent characteristic of streaming video over HTTP, or is the client simply picking the wrong video rate?

We first confirm that the available bandwidth really is available for streaming video. We do this using a feature provided by the Service A client that allows users to manually select a video rate and disable the client's automatic rate selection algorithm. We repeat the above experiment, but with a slight modification. As soon as the client picks a lower rate, we manually force the video to play at 1750kb/s. Figure 5 shows the results. Interestingly, not only can the client maintain the playback rate of 1750kb/s without causing rebuffering events, the throughput also increases. This suggests that the downward spiral effect is caused by underestimation of the available bandwidth in the client's rate selection algorithm. The bandwidth is available, but the client needs to go grab it.

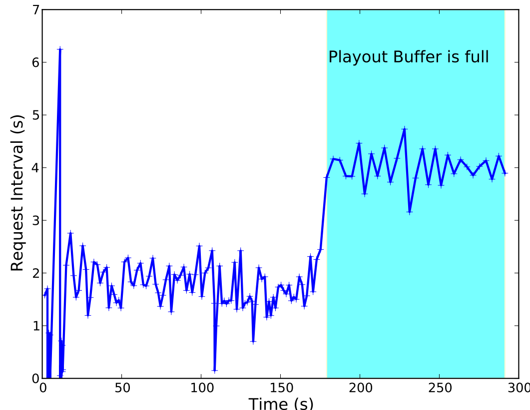
4.2 Service B

Figure 4(b) shows the same downward spiral effect in Service B. As before, the bottleneck bandwidth is 5Mb/s and the RTT is around 20 ms. We start a video streaming session first, allow it to settle at its highest rate (3200kb/s) and then start a competing flow after

[3]To eliminate variation caused by congestion at the server, we verified that the same problem occurs if we download the competing video file from a different server at the same CDN.



(a) TCP throughput before and after the buffer fills.



(b) Request interval before and after the buffer fills.

Figure 6: (Service A) Before and after the playback buffer fills at 185 seconds.

337 seconds, by reading the same video file from the same server.

The client should drop the video rate to 2500kb/s (its fair share of the available bandwidth). Instead, it steps all the way to the lowest rate offered by Service B, 650kb/s and occasionally to 1000kb/s. The throughput plummets too.

4.3 Service C

We observe the downward spiral effect in Service C as well. Since Service C does not automatically switch between its HD and SD bitrates, we do two separate experiments.

In the HD experiment, as shown in Fig. 4(c), we set the bottleneck bandwidth to 22Mb/s. To start with, the client picks the highest HD video rate (9Mb/s). When the client's playback buffer is full, the video flow is limited by the receive window, and the perceived throughput converges to the same value as the playback rate. We start the competing flow at 100 seconds downloading the same video file (9Mb/s video rate) from the same CDN.

Each flow has 11Mb/s available to it, plenty for the client to continue playing at 9Mb/s. But instead, the client resets the connection and switches to 4.5Mb/s and then 3Mb/s, before bouncing around several rates.

SD is similar. We set the bottleneck bandwidth to 5Mb/s, and the client correctly picks the highest rate (2000kb/s) to start with, as shown in Figure 4(d). When we start the competing flow, the video client drops down to 1000kb/s even though its share is 2.5Mb/s. Since Service C only offers three SD rates, we focus on its HD service in the rest of the paper.

5. WALKING THE DOWNWARD SPIRAL

To understand how the downward spiral happens, we examine each service in turn. Although each service enters the downward spiral for a slightly different reason, there is enough commonality for us to focus first on Service A (and Fig. 4(a)) and then describe how the other two services differ.

5.1 Initial Condition: No Competing Flow

In the absence of a competing flow (first 400 seconds), the Service A client correctly chooses the highest playback rate. Because the available network bandwidth (5Mb/s) is much higher than the playback rate (1750kb/s), the client busily fills up its playback buffer, and the bottleneck link is kept fully occupied. Eventually the playback buffer fills (after 185 seconds) and the client pauses to let it drain a little before issuing new requests. Figure 6(a) shows how the TCP throughput varies before and after the playback buffer fills up. After the buffer is full, the client enters a periodic ON-OFF sequence. As we will see shortly, the ON-OFF sequence is a part of the problem (but only one part). Before the buffer fills, the client requests a new 4 second chunk of video every 1.5 seconds on average (because it is filling the buffer). Figure 6(b) confirms that after the buffer is full, the client requests a new 4 second chunk every 4 seconds, on average. The problem is that during the 4 second OFF period, the TCP congestion window (*cwnd*) times out—due to inactivity longer than 200ms—and resets *cwnd* to its initial value of 10 packets [5, 6]. Even though the client is using an existing persistent TCP connection, the *cwnd* needs to ramp up from slow start for each new segment download.

It is natural to ask if the repeated dropping back to slow-start reduces the client's perceived throughput, causing it to switch to a lower rate. With no competing flow, it appears the answer is 'no'. We verify this by

measuring the throughput for many requests. We set the bottleneck link rate to be 2.5Mb/s, and use traces collected from actual sessions to replay the requests over a persistent connection to the same server, and pause the requests at the same interval as the pauses in the trace. Figure 7(a) shows the CDF of throughput the client perceived for requests corresponding to various playback rates. The perceived throughput is pretty accurate. Except for some minor variation, the perceived throughput accurately reflects the available bandwidth, and explains why the client picks the correct rate.

5.2 The Trigger: With a Competing Flow

Things go wrong when the competing flows starts (after 400 seconds). Figure 7(b) shows the throughput client perceived are mostly much too low, when there is a competing flow.[4] If we look at the progression of *cwnd* for the video flow after it resumes from a pause, we can tell how the server opens up the window differently when there is a competing flow. Because we don't control the server (it belongs to the CDN) we instead use our local proxy to serve both the video traffic and the competing flow, and use the `tcp_probe` kernel module to log the *cwnd* values. The video traffic here is generated by requesting a 235kbps video segment. Figure 8(a) shows how *cwnd* evolves, starting from the initial value of 10 at 1.5 seconds, then *repeatedly being beaten down by the competing* `wget` *flow*. The competing `wget` flow has already filled the buffer during the OFF period, and so the video flow sees very high packet loss. Worse still, the chunk is finished before *cwnd* climbs up again, and we re-enter the OFF period. The process will repeat for every ON-OFF period, and the throughput is held artificially low.

For comparison, and to understand the problem better, Figure 8(b) shows the result of the same experiment with a chunk size five times larger. With a larger chunk size, the *cwnd* has longer to climb up from the initial value; and has a much greater likelihood of reaching the correct steady state value.

Now that we know the throughput client perceived is very low (because of TCP), we would like to better understand how the client *reacts* to the low throughputs. We can track the client's behavior as we steadily reduce the available bandwidth, as shown in Figure 9. We start with a bottleneck link rate of 5Mb/s (and no competing flow), and then drop it 2.5Mb/s (to mimic a competing flow), then keep dropping it by 100kb/s every 3 minutes. The dashed line shows the available bandwidth, while the solid line shows the video rate picked by the client. Clearly, the client chooses the video rate conservatively. When available bandwidth drops from from 5Mb/s to 2.5Mb/s, the video rate goes down to 1400kb/s, and so on.

We can now put the two pieces together. Figure 7(b) tells us that in the presence of a competing flow, a client streaming at a playback rate of 1750kb/s perceives a throughput of less than 2Mb/s for 60% of the requests. Figure 9 tells us that with a perceived throughput of 2Mb/s, the client reduces its playback rate to 1050kb/s. Thus, 60% of the time the playback rate goes down to 1050kb/s once the competing flow starts.

It is interesting to observe that the Service A client is behaving quite rationally given the throughput it receives. The problem is that because it observes the throughput *above* TCP, it is not aware that TCP itself is having trouble reaching its fair share of the bandwidth. Coupled with a (natural) tendency to pick rates conservatively, the rate drops down.

5.3 The Spiral: Low Playback Rate

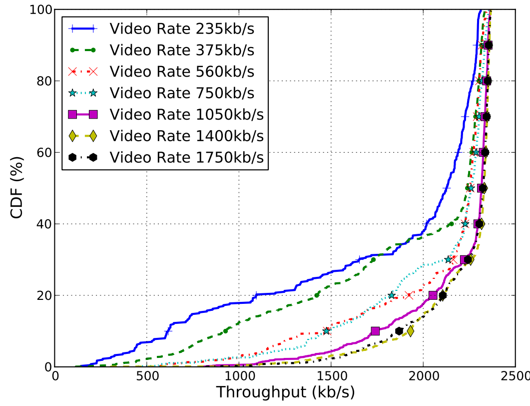
Finally, there is one more phenomenon driving the video rate to its lowest rate. Recall that each HTTP request is for four seconds of video. When the video rate is lower, the four-second segment is smaller, as shown in Figure 10. With a smaller segment, the video flow becomes more susceptible to perceive lower throughput, as shown in Figure 7(b). With a lower throughput, the client reduces the playback rate, creating a vicious cycle as shown in Figure 11. The feedback loop will continue until it reaches a steady state where the perceived available bandwidth is large enough to keep the rate selection algorithm at the chosen rate. In the worst case, the feedback loop creates a "death spiral" and brings the playback rate all the way down to its lowest value.

To summarize, when the playout buffer is full, the client enters a periodic ON-OFF sequence. The OFF period makes the TCP connection idle for too long and reset its congestion window. Also during the OFF period, the competing tcp flow fills the router buffer, so the video flow sees high packet loss once it is back to the ON period. Worse still, the video could finish downloading its video chunk before its *cwnd* climbs up to its fair share and thus re-enters the OFF period. This process repeats for every ON-OFF period; as the consequence, the video flow underestimates the available bandwidth and switches to a lower video rate. When switching to a lower rate, the client requests for a smaller video chunk, which makes the video flow further underestimate the available bandwidth, forming a vicious cycle.

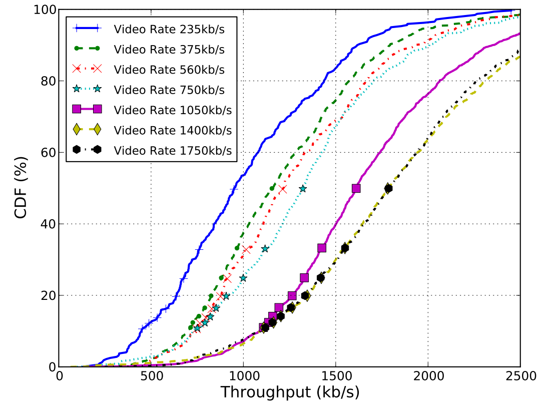
5.4 With Different Network Conditions

Our experiments so far were all for the same CDN. Fig. 12 shows different behavior for different CDN. The

[4]In Figure 7(b), the bottleneck bandwidth is set to 5Mb/s so that the available fair-share of bandwidth (2.5Mb/s) is the same as in Figure 7(a). Note that some chunk downloads are able to get more than its fair share, this is because the competing flow experiences losses and has not ramped up to its fair share yet. This is the reason why some of the CDF curves does not end with 100% at 2.5Mb/s in Figure 7(b).

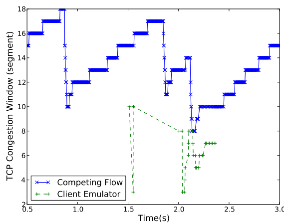


(a) Service A with no competing flow.

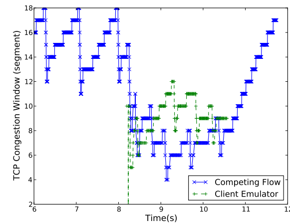


(b) Service A with one competing flow.

Figure 7: (Service A) Throughput at HTTP layer with and without a competing flow.



(a) A 235kbps Segment.



(b) Concate five contiguous 235kbps segments into one.

Figure 8: (Service A) The evolution of *cwnd* for different segment sizes.

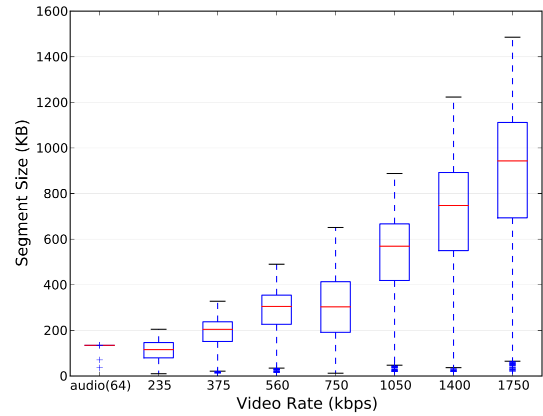


Figure 10: (Service A) The segment size for different video rates.

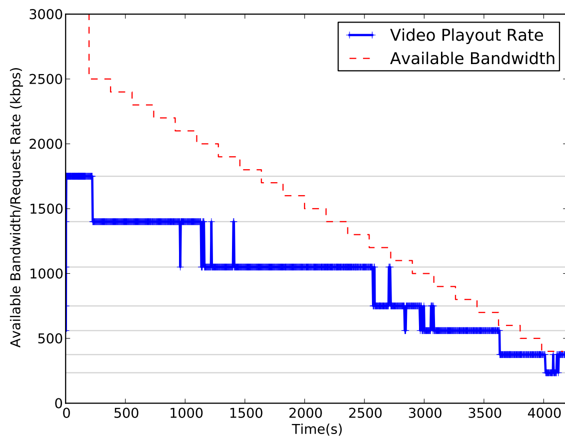


Figure 9: (Service A) The client picks a video rate depending on the available bandwidth. The horizontal gray lines are the available rates.

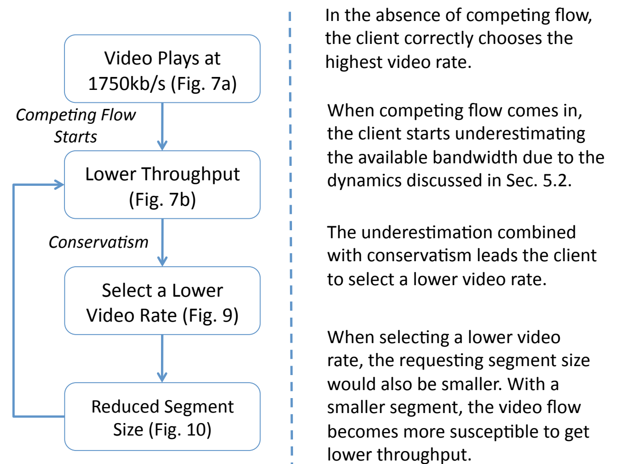


Figure 11: (Service A) The feedback loop.

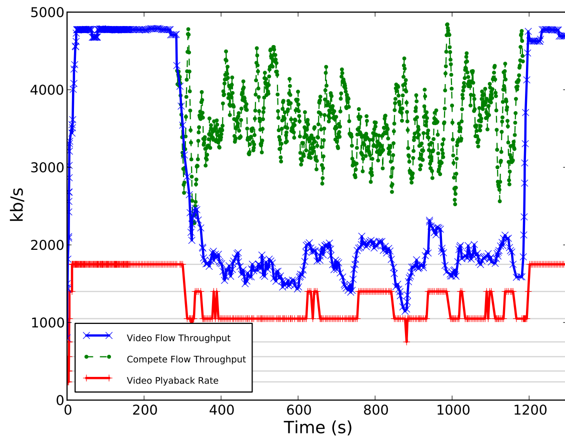


Figure 12: (Service A) When streaming from another CDN, the playback rate stabilizes at 1050kb/s.

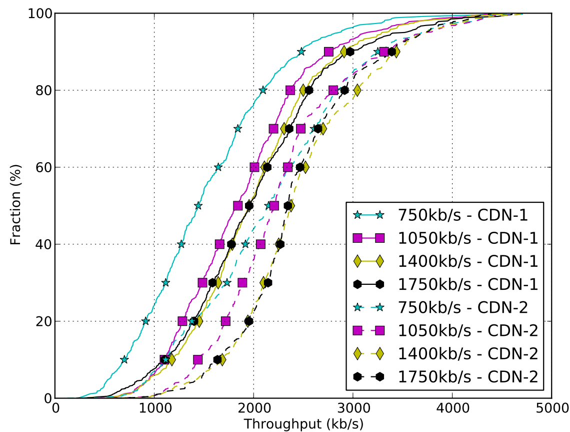


Figure 13: (Service A) Different network path properties (CDNs) lead to different perceived throughput.

path properties are different, and Fig. 13 shows that the perceived throughput is higher — hence the Service A client picks a higher rate (1050kb/s).

For comparison, we asked 10 volunteers to rerun this experiment with Service A in their home network connected to different ISPs, such as AT&T DSL, Comcast, Verizon and university residences. Even though there was sufficient available bandwidth for the highest video rate, even with the competing flow, seven people reported a rate of only 235kb/s-560kb/s.

5.5 Service B

Service B also exhibits ON-OFF behavior, but at the TCP level and not the HTTP level, i.e., the pause could happen while downloading a video segment. When its video playback buffer is full, the client would stop taking data from the TCP socket buffer. Eventually, the

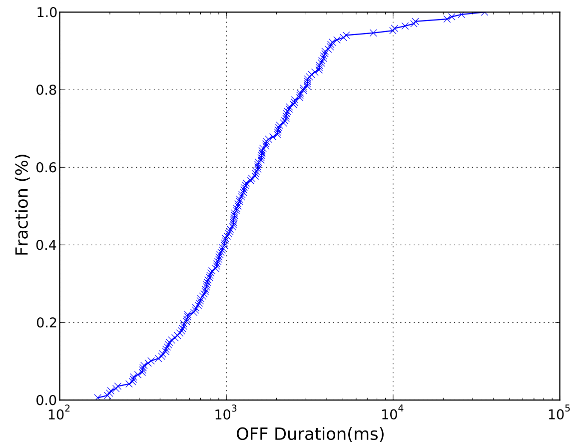


Figure 14: (Service B) Almost all the OFF periods in a single video session are greater than RTO (200ms).

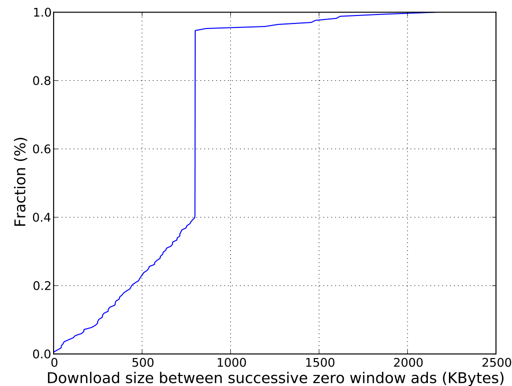
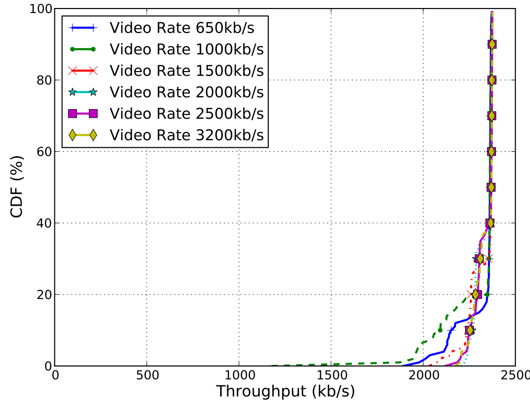


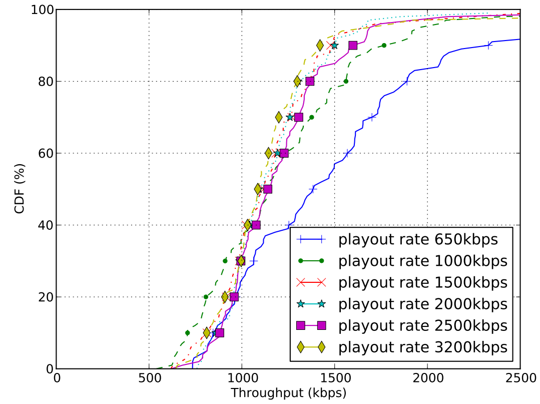
Figure 15: (Service B) When the video stream is receiver-limited, the effective segment size is small.

TCP socket buffer would also be full and triggers the TCP flow control to pause the server by sending a *zero window advertisement*. In Fig. 4(b), each *zero window advertisement* is marked by a hexagon. The client starts issuing *zero window advertisements* at around 100s, and continues to do so until a few seconds after the competing flow starts. Fig. 14 shows the CDF of the duration of the OFF periods. Almost all the pauses are longer than 200ms, and so *cwnd* is reset to its initial value. Thus, Service B effectively exhibits an ON-OFF behavior similar to that of Service A.

Worse still, during an ON period Service B does not request many bytes; Fig. 15 shows that over half of the time, it reads only 800kbytes, which is not enough for the *cwnd* to climb up to its steady state before the next OFF period. As the result, Fig. 4(b) and Fig. 16(b) show that the TCP throughput is only around 1Mbps to 1.5Mbps, causing Service B to pick a video rate of



(a) Service B with no competing flow.



(b) Service B with one competing flow.

Figure 16: (Service B) The TCP throughput changes in the presence of a competing flow.

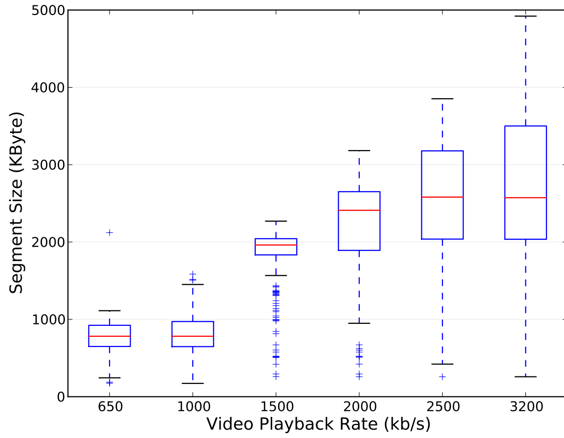


Figure 17: (Service B) Segment size varies with video playback rate.

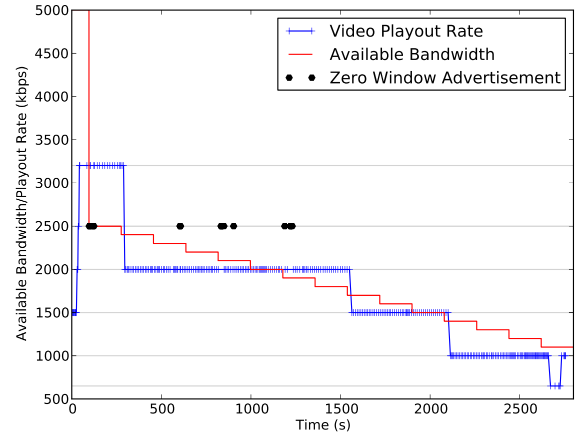


Figure 18: (Service B) While not too conservative, the client is limited by its TCP throughput.

1000kb/s, or even 650kb/s. As we saw earlier, when competing with another flow, the smaller the request, the higher the likelihood of perceiving a lower throughput. The problem would be even worse if it was coupled with a conservative rate selection algorithm. Fortunately, Service B is not nearly as conservative as the other two services, as shown in Fig. 18.

Similar to Service A, the segment size gets smaller with decreasing playback rate – from 3MByte to 1MByte as shown in Fig. 17. However, segment size does not play a significant role in the downward spiral effect of Service B, since the ON-OFF behavior is at the TCP level and the number of bytes for each ON period is not determined by segment size.

5.6 Service C

Service C performs an open-ended download, instead of segment-by-segment. Thus, it is not constrained by segment size, and does not pause between HTTP requests. It only slows down when the receive window fills, and so reduces the number of bytes in flight. However, the OFF period for Service C is less than an RTO and therefore does not trigger TCP to reset its window size. The problem with Service C is mainly caused by its conservativeness and being sensitive to temporal behaviors in TCP. Figure 19 shows the conservativeness of Service C, it steps down to 3Mb/s even when the available bandwidth is more than 9Mb/s. As shown in Figure 4(c), Service C regulates itself at 9Mb/s after its playout buffer is full. Thus, the competing flow would take over the rest of the 13Mb/s available bandwidth when it starts and make Service C to perceive less than 9Mb/s of available bandwidth. As the consequence, the video flow steps down to the video rate of 3Mb/s. Since Service C does open-ended download, it has to reset the current TCP connection to stop downloading the current video file and starts a new connection for the file with the newly selected rate. This would make the video

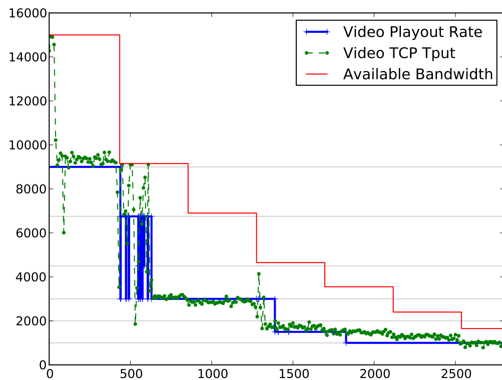


Figure 19: (Service C) The client is conservative in its rate selection.

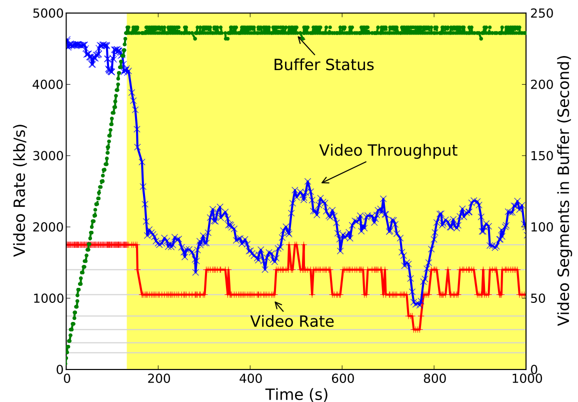


Figure 20: Custom client, similar to Service A – equally conservative, with a 10-sample moving average filter – displays the downward spiral.

flow perceives less bandwidth during the transition, as TCP needs to go through the phase of connection establishment and slow start. Thus, we can see that even though Service C is able to perceive a higher throughput and switches to a higher video rate after TCP ramps up, the overhead of transition makes it easily to switch down again after a short period.

6. ACTIVELY INTERVENING

We can verify our explanation in Section 5 by modifying the system. For example, a modified server might tell the client the highest rate achieved during a chunk download, or an augmented HTTP interface might report the recent bandwidth history to the client. Unfortunately, this is not an option for our three services, because they use unmodified HTTP CDN services. The experiments we describe next therefore leave the ecosystem unchanged, and only require client changes. Our intent is not to propose a new client design, but rather to confirm our understanding of the root causes of the downward spiral by trying small changes to the algorithm.

6.1 The Custom Client

For brevity, we focus on Service A. Our approach is to replay a movie, using the same CDN and chunks as Service A, but using our own client algorithm. The three services keep their algorithms secret, but our measurements provide a reasonable baseline. For Service A, Figure 9 indicates the bandwidth below which the client picks a lower video rate. Assume that Service A estimates bandwidth by simply dividing the download size by the download time and passing it through a fixed-size moving-average filter. We can estimate the size of the filter by measuring how long it takes from when the bandwidth drops until the client picks a new rate. A number of traces from Service A suggest a filter with 10 samples, though the true algorithm is probably more nuanced.

To closely mimic the Service A client, our custom client requests the movie chunks from the same locations in the CDN: We capture the chunk map given to the client after authentication, which locates the movie chunks for each supported playback rate. This means our client will experience the same chunk-size variation over the course of the movie, and when it shifts playback rate, the chunk size will change as well. Since our custom client uses tokens from an earlier playback, the CDN cannot tell the difference between our custom client and the real Service A client. To further match Service A, the playback buffer is set to 240 seconds, the client uses a single persistent connection to the server, and it pauses when the buffer is full. We first validate the client, then consider three changes: (1) being less conservative, (2) changing the filtering method, and (3) aggregating chunks.

6.2 Validating our Custom Client

Figure 20 shows the custom client in action. After downloading each segment, the custom client selects the playback rate based on Service A's conservative rate selection algorithm, observed in Figure 9. Once the playback buffer is full we introduce a competing flow. Like the real client, the playback rate drops suddenly when the competing flow starts, then fluctuates over the course of the movie. The downward spiral does not bottom out, which we suspect is due to some suttble differences between Service A's algorithm and ours.

6.3 Less Conservative

Bandwidth estimates based on download sizes and durations tend to under-report the available bandwidth, especially in the presence of a competing flow. If the algorithm is conservative, it exacerbates the problem. We try a less conservative algorithm, with a conservatism of 10% instead of 40%. Conservatism of 40% means the

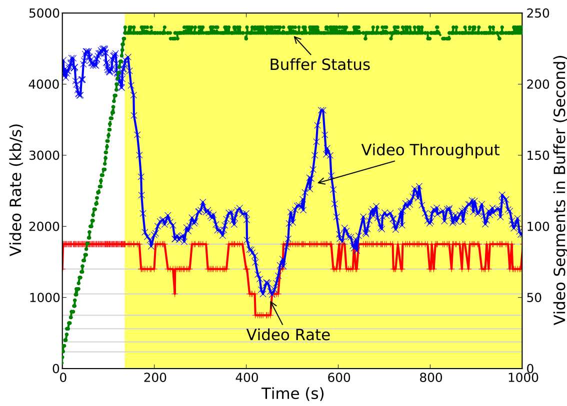


Figure 21: Custom client – without the conservatism, but with a 10-sample moving average filter.

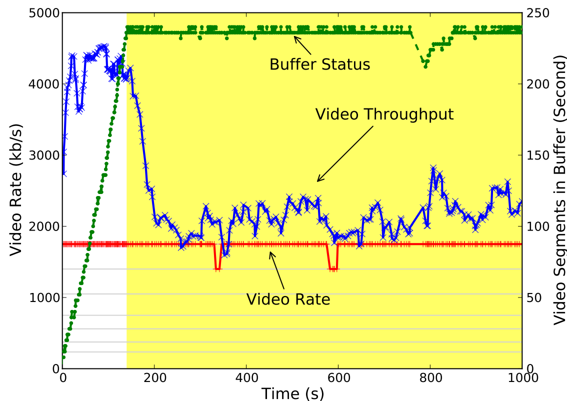


Figure 22: Custom client – without the conservatism, and with 80th percentile filter.

client requests for a video rate at highest of 1.2Mb/s when perceived 2.0Mb/s; while 10% means it would request at highest of 1.8Mb/s when perceived 2.0Mb/s. According to Figure 9, Service A requests video rate roughly at conservatism of 40%. Figure 21 shows that the video rate is higher, even though the playback buffer stays nice and full. The result is higher quality video, a high playback buffer occupancy (*i.e.* resilience against rebuffering) and four minutes of buffering to respond to changes in bandwidth. Note that even though the algorithm is less conservative, the underlying TCP will make sure the algorithm be a "good citizen" and only gets its fair share of available bandwidth.

6.4 Better Filtering

Averaging filters provide a more stable estimate of bandwidth, but a single outlier can confuse the algorithm. For example, a few seconds of low-information movie credits reduces the chunk size and the algorithm might drop the rate. So instead, we use medians and quantiles to reduce the vulnerability to outliers. Figure 22 shows what happens if we use only the 80% quantile of the measured rates of the past ten segment download. Variation is greatly reduced, and the majority of the movie plays at the highest-available rate. The playback buffer has small fluctuations, but is still far from a rebuffer event.

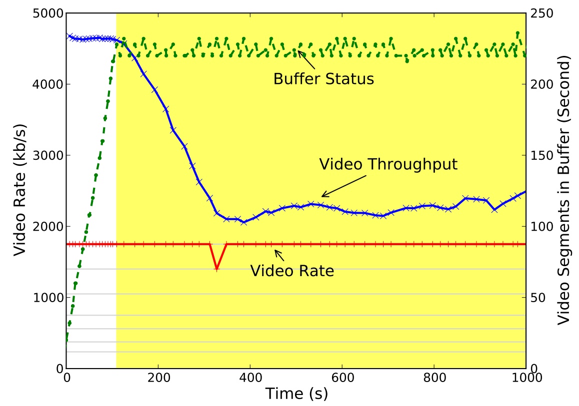


Figure 23: Custom client with increased segment size (5x).

6.5 Bigger Chunks

As noted earlier, bigger chunks give us better estimates of the available bandwidth, allowing TCP to escape slow-start. Figure 23 shows what happens if our client aggregates five requests into one. With the larger chunk size, the perceived throughput is more stable, and both the playback rate and buffer size are more stable.

In summary, larger chunks let TCP get its fair share and improve the perceived throughput. Picking higher rates less conservatively and filtering measurements more carefully can improve video quality. But we should note that these improvements are for one movie on one service. Given the prevalence of the downward spiral effect, these should not be interpreted as hard recommendations; merely as added detail to our understanding of the problem.

7. RELATED WORK

The related work largely considers three overlapping areas: systems for video streaming; measurements to understand their performance, and the design and analysis of rate selection algorithms.

Video Streaming Services. The first category covers video streaming approaches using HTTP, such as the commercial ones from Adobe, Apple, and Microsoft described in [21], which differ in their alignment of video switching rates, whether A/V streams are combined, and whether requests are issued as byte ranges or for pre-specified chunks. A more recent technique is MPEG DASH (Dynamic Adaptive Streaming over HTTP) [7]

which standardizes the formatting of video content and leaves open the specific client player algorithm. These techniques underpin the major commercial services like YouTube, Netflix, and Hulu.

Video Streaming Measurement. The second category measures the performance of individual video streaming clients experiencing local traffic conditions ("in the lab"), all the way to distributed measurement systems that compare the performance of thousands of clients ("in the wild").

The work most similar to ours is [3], where the authors also parse HTTP messages to determine playback rates and use a bandwidth limiter to test clients under varying network conditions. However, [3] focuses on the unfairness problem among two video players, while in this work we focus on the unfairness problem between a video player and a long-lived TCP flow. The paper considers a significantly different scenario: it focuses on a video client competing against *another video client*. In this context, they observe similar pathologies: poor bandwidth estimation, leading to instability. However, they explain their observations entirely in terms of the application-layer ON-OFF behavior of video clients; even if one video client perfectly obtained its fair share when ON, it can fail to correctly estimate available bandwidth (depending on the amount of overlap with the ON periods of the other client). By contrast, our paper demonstrates that this is only a symptom of a more general problem: inaccurate bandwidth estimation occurs *even* when the competing flow does not exhibit ON-OFF behavior. As we show in this paper, the problem arises because it is hard to estimate bandwidth above TCP. Others have identified the same problem but not explained its causes or validated potential fixes [15, 4].

Measuring the CDN servers rather than clients provides different insights. In [1], the authors examine the CDN selection strategy of Hulu, while in [2], the authors look at Netflix. Both papers find a predisposition for clients to stay with the original CDN, despite variation between CDNs and over time. In [9], the authors describe lessons learned from a distributed commercial measurement system to understand the effects of Quality-of-Experience (QoE) metrics on viewer engagement and retention. Rebuffer rates and average video quality are QoE metrics with measurable impacts on viewer engagement, which underscores the importance of getting rate measurement and selection right in the presence of competing flows.

Other work looks at network characteristics of video streaming traffic, rather than focusing on client or viewer experience [18, 11, 23]. In particular, the authors in [18] show ON-OFF cycle behavior for YouTube and Netflix and use a model to study aggregates of video client and their effects on the network. Both CDN and network traffic papers do not consider local effects on measured bandwidth or their effects on rate stability.

Rate Selection Algorithms. The third category is work on rate selection algorithms. This work complements ours, as a control system always benefits from more accurate measurements. In [8], the authors propose an algorithm to maintain the playout buffer at a target level. In [16], the authors implement a different buffer-aware rate selection algorithm and experimentally measure user preferences for gradual and infrequent playback rate changes. In [22], the authors model the rate selection problem as a Markov Decision Process and use a dynamic programming technique to choose a streaming strategy that improves QoE. In [13], the authors use simulations to show how parallel HTTP sessions can improve playback quality. Server-side pacing is another approach to selecting rate used by YouTube, as described in [10, 12].

8. CONCLUSION

Despite some differences in specific service implementations, all three services we study display degraded performance in the presence of competing traffic, well below the video quality possible if the client used its fair share of bandwidth. At a high level, our measurement analysis and experiments suggest that the root cause of this failure is a lack of information. In many circumstances, the HTTP layer is simply not privy to continuous high-fidelity feedback about the fair share at the bottleneck link.

There are two ways to interpret our observations. On one hand, we observe that determining the fair share of bandwidth available at the bottleneck is *precisely the role of TCP*. Thus, one path forward might be to suggest that we should design the client to improve information flow from TCP to the HTTP layer. In particular, we should ensure that TCP has a chance to reach its steady-state fair share; for example, increasing the chunk size enables this effect.

However, we believe there may be a more radical solution: *do not attempt to estimate bandwidth at all!* The video streaming client has two competing goals: attain the highest bitrate possible while avoiding buffer underruns. Thus the objective is *not* to ensure the buffer stays full; the objective is to ensure the buffer does not go empty. Since the buffer holds several minutes of video, this shift in perspective suggests that if the buffer is full then the client has picked a rate that is *too low*. Rather, the client should *increase* the bitrate when the buffer is high and *decrease* it when the buffer falls low. Though this sounds aggressive, note that it is exactly the correct layer separation: it hands off to TCP the objective of obtaining the fair share of bandwidth, and tries to always ensure the client picks the highest rate possible. This suggests an intriguing path forward for future re-

search: design video-streaming clients that deliver high performance by *eliminating* bandwidth estimation all together.

Acknowledgment

The authors would like to thank Kok-Kiong Yap, Masayoshi Kobayashi, Vimalkumar Jeyakumar and Yiannis Yiakoumis for helpful discussions that shaped the paper.

9. REFERENCES

[1] V. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z. Zhang. A tale of three cdns: An active measurement study of hulu and its cdns.

[2] V. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling netix: Understanding and improving multi-cdn movie delivery. In *IEEE INFOCOM 2012*.

[3] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. Begen. What happens when http adaptive streaming players compete for bandwidth? 2012.

[4] S. Akhshabi, C. Dovrolis, and A. Begen. An experimental evaluation of rate adaptation algorithms in adaptive streaming over http. In *ACM MMSYS 2011*, San Jose, CA, USA, Feburary 2011.

[5] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), Sept. 2009.

[6] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), Apr. 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[7] MPEG DASH specication (ISO/IEC DIS 23009-1.2), 2011.

[8] L. De Cicco, S. Mascolo, and V. Palmisano. Feedback control for adaptive live video streaming. *Proc. ACM MMSys*, 2011.

[9] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang. Understanding the impact of video quality on user engagement. In *ACM SIGCOMM 2011*, Toronto, Canada, August 2011.

[10] M. Ghobadi, Y. Cheng, A. Jain, and M. Mathis. Trickle: Rate limiting youtube video streaming. In *Annual Technical Conference*. USENIX, 2012.

[11] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pages 15–28. ACM, 2007.

[12] L. Kontothanassis. Content delivery considerations for different types of internet video. In *ACM MMSYS 2012 (Keynote)*, Chapel Hill, NC, USA, Febrary 2012.

[13] C. Liu, I. Bouazizi, and M. Gabbouj. Parallel adaptive http media streaming. In *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*, pages 1–6. IEEE, 2011.

[14] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo. Netfpga–an open platform for gigabit-rate network switching and routing. In *MSE '07: Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, pages 160–161, 2007.

[15] K. Miller, E. Quacchio, G. Gennari, and A. Wolisz. Adaptation algorithm for adaptive streaming over http.

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

[17] Sandvine: Global Internet Phenomena Report. `http://www.sandvine.com/news/pr_detail.asp?ID=312`.

[18] A. Rao, A. Legout, Y. Lim, D. Towsley, C. Barakat, and W. Dabbous. Network characteristics of video streaming traffic. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 25. ACM, 2011.

[19] RTMPDump. `http://rtmpdump.mplayerhq.hu/`.

[20] Consumer Report: Streaming Video Services Rating. `http://www.consumerreports.org/cro/magazine/2012/09/best-streaming-video-services/index.htm`.

[21] M. Watson. HTTP Adaptive Streaming in Practice. `http://web.cs.wpi.edu/~claypool/mmsys-2011/Keynote02.pdf`.

[22] S. Xiang, L. Cai, and J. Pan. Adaptive scalable video streaming in wireless networks. In *Proceedings of the 3rd Multimedia Systems Conference*, pages 167–172. ACM, 2012.

[23] M. Zink, K. Suh, Y. Gu, and J. Kurose. Characteristics of youtube network traffic at a campus network measurements, models, and implications. In *Computer Networks, Volume 53, Issue 4*, pages 501–514. Elsevier, 2009.