A BUFFER-BASED APPROACH TO VIDEO RATE ADAPTATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Te-Yuan Huang
June 2014

This dissertation is online at: http://purl.stanford.edu/wp360ym0003

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Nick McKeown, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Ramesh Johari

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Sachin Katti

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

iv

Abstract

During peak viewing time, well over 50% of US Internet traffic is streamed video from Netflix and YouTube. To provide a better streaming experience, these services adapt their video rates by observing and estimating the available capacity. However, accurate capacity estimation is difficult due to highly variable throughput and complex interactions between layers. As a result, existing rate adaptation algorithms often lead to suboptimal video quality and unnecessary rebuffers.

This thesis proposes an alternative buffer-based approach to adapt video rate. Rather than presuming that capacity estimation is always required, this approach starts the design by *only* using the playback buffer occupancy, and then ask *when* capacity estimation can be helpful. This design process leads to two separate phases of operation: during the steady-state phase, when the buffer encodes adequate information, we choose the video rate based only on the playback buffer; during the startup phase, when the buffer contains little information, we augment the buffer-based design with capacity estimation. This approach is tested with a series of field experiments spanning millions of Netflix users from May to September, 2013. The results demonstrate that although a simple capacity estimation is important during the startup phase, it is unnecessary in the steady state. The buffer-based approach allows us to reduce the rebuffer rate by 10–20% compared to a commercial algorithm used in Netflix, while delivering a similar overall average video rate and a higher video rate in steady state.

Acknowledgements

Pursuing a PhD is an adventure, and this journey would not have been as fulfilling and rewarding without the guidance and the support of many people.

First and foremost, I would like to thank my advisor, Prof. Nick McKeown. Nick is an amazing teacher and has had great influence on me. Nick always encourages me to challenge the status quo, to think outside of the box, and to dare to attempt the impossible. Over the years, Nick has taught me to appreciate the beauty of simple designs and the intuition behind the details. Nick has a magic power in that he always can put a positive twist on seemingly negative results, and he is able to keep me motivated through the difficult times. Nick is not only a brilliant researcher, but he is also a true believer on bridging the gap between academic ideas and industrial practice. Thanks to Nick, I get to seek feedback from the video streaming industry and have the opportunity to work with Netflix on this thesis. I am privileged to be his student.

I have also been very fortunate to work with Prof. Ramesh Johari. Ramesh is always encouraging and positive. Many times, when I have been lost in details, Ramesh would remind me to take a step back and approach the problem again with principles in mind. This thesis has benefited greatly from this thinking process.

I would also like to thank my dissertation committee: Prof. Sachin Katti, Prof. Hui Zhang, Prof. Fouad Tobagi and Prof. Ashish Goel. Their feedback and thought-provoking comments have better shaped this thesis. I have also benefited significantly from my interactions with Prof. Monica Lam, who kindly gave me plenty of guidance during my studies at Stanford.

I would also like to give special thanks to Netflix, who has given me the opportunity to experiment with the buffer-based approach in their service and has allowed me to improve the thesis based on the lessons that I learned there. I am grateful to Matt Trunnell, Mark Watson, Greg Wallace-Freedman, Kevin Morris, Wei Wei, Siqi Chen, Daniel Ellis, Alex

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Video streaming is a huge and growing fraction of Internet traffic. During the evening peak hours (8pm - 1am EDT), well over 50% of US Internet traffic is video streamed from Netflix and YouTube [28, 29]. Unlike traditional video downloads that must complete fully before playback can begin, streaming video starts playing within seconds. Each video is encoded at a number of different rates (typically 235kb/s standard definition to 5Mb/s high definition) and stored on servers as separate files. The video client—running on a home TV, game console, web browser, DVD player, etc.—chooses which video rate to stream by monitoring network conditions and estimating the available network capacity. This process is referred to as *adaptive bitrate selection* or ABR.

In this thesis, we propose an alternative approach to designing ABR algorithms. Rather than presuming that capacity estimation is always required, we propose to start the design by *only* using the playback buffer occupancy, and then ask *when* capacity estimation can be helpful. Before developing the motivation for this buffer-based approach in Chapter 2 and 3, we will first explain how rate selection works today and explain the challenges faced by the current practice.

1.1 Common Architecture for Internet Video

Most commercial video streaming services currently 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
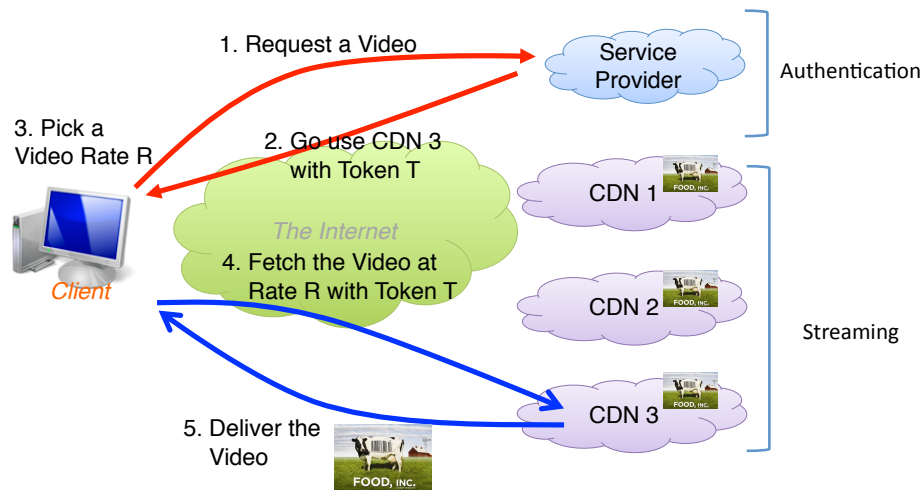
Figure 1.1: Common architecture of video streaming services over HTTP.

universally accessible (CDNs have already made sure their service can reach through NATs to end-hosts), and it is 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 the viewers' delight.

Currently (2014), the architecture of most commercial video streaming services can be summarized as in Figure 1.1. Video content is hosted at multiple CDN providers and is streamed over HTTP to the clients. A video service generally supports several different platforms, e.g., web browser plugin, game console and TV. A video session has two phases: authentication and streaming. When a client requests a video, the service provider authenticates the user account and directs the client to a CDN hosting the video. The video service provider tells the client which video streaming rates are available and issues a token for each rate. The client then picks a video rate and requests the video at the selected rate by presenting a token as a credential to the designated CDN.

Figure 1.2 provides details in the streaming phase and further illustrates how rate selection works over HTTP. A streaming service first encodes each video at a number of different rates and deploys them on CDNs as separate files. The set of video rates typically ranges from 235kb/s standard definition to 5Mb/s high definition. The client begins with a pre-configured starting video rate and monitors the arriving traffic to pick the next video rate. Because the service runs over "vanilla HTTP," the client makes the decision and the CDN
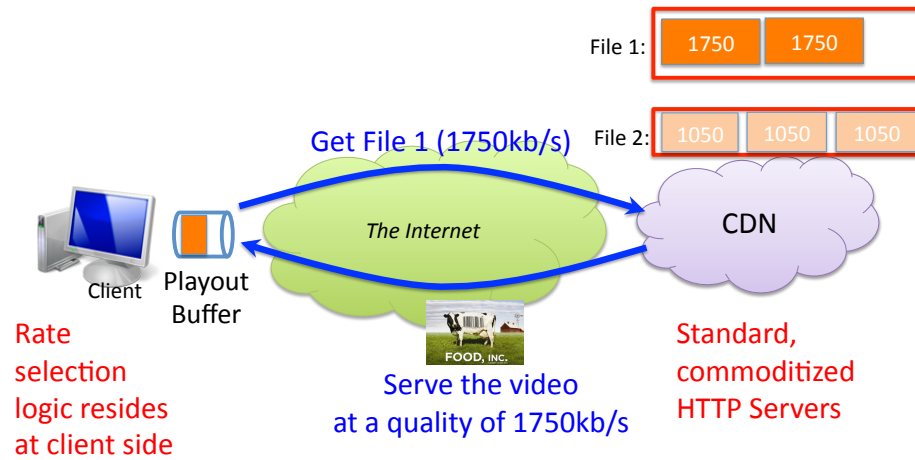
Figure 1.2: Overview of video rate selection over HTTP.

server is not involved in picking the video rate.[1] Once the ABR algorithm chooses which video rate to stream, the client requests *chunks* of the corresponding file from the server. Each chunk contains a fixed duration of video, typically 2 to 10 video seconds. The higher the video rate, the larger the chunk (in bytes).

The goal of ABR algorithms is to deliver a good streaming experience to the users. Currently there is still an on-going effort to determine the metrics which contribute to quality of experience (QoE), and it is not completely clear how user engagement depends on the QoE metrics [7, 23]. However, user engagement is heavily affected by viewing interruptions and video rate [11, 19, 23]. The delay before playing and how often the video rate changes can also have an effect on user engagement [11, 34]. This thesis focuses on reducing interruptions and improving video rate with some consideration for video rate changes.

1.2 Impact of Rate Selection on Streaming Experience

The choice of video rate affects the playback buffer occupancy. Figure 1.3 shows the relationship between available capacity and video rate in a video playback buffer. The buffer occupancy is generally tracked in *seconds of video*. Every second, one second of video is removed from the buffer and played to the user. The buffer drains at unit rate, since one second is played back every second of real time. The buffer receives video at the rate that

---

[1]Services that control both the server and the client (*e.g.,* YouTube) can involve the server and the client when picking a rate.
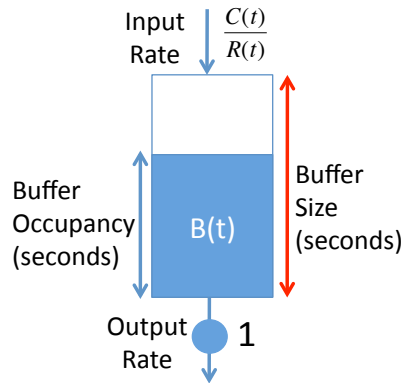
Figure 1.3: The video playback buffer, tracked in seconds. Every second, the buffer outputs one second of video to play to the user. The buffer also receives video at the rate that can be represented as a ratio between the system capacity and the selected video rate, $C(t)/R(t)$.

can be represented as a ratio between the system capacity, $C(t)$, and the selected video rate, $R(t)$. For example, if $C(t)$ is 9Mb/s and $R(t)$ is 3Mb/s, for every second, 3 seconds worth of video is inserted into the buffer.

If the ABR algorithm picks a video rate that is lower than the system capacity, the input rate of the buffer is larger than the drain rate ($C(t)/R(t) > 1$) and the buffer increases. However, this higher input rate means that the algorithm does not maximize the video rate and does not fully utilize the capacity. On the other hand, if the ABR algorithm picks a video rate that is greater than the system capacity, then new data is put into the buffer at rate $C(t)/R(t) < 1$ and the buffer decreases. Put another way, if more than one chunk is played before the next chunk arrives, then the buffer is depleted. If the ABR algorithm keeps requesting chunks that are too big for the network to sustain (i.e., the video rate is too high), eventually the buffer will run dry, playback freezes, and the viewer see the familiar "Rebuffering..." message on the screen. These interruptions are often referred to as *rebuffers* or *rebuffering events*. Note that if rebuffers happen because the capacity is not able to sustain even the minimum video rate R_{min}, there is nothing an ABR algorithm can do to avoid them. However, if rebuffers happen because the ABR algorithm chooses a non-sustainable rate, they are avoidable with a better algorithm design and we call them *unnecessary rebuffers*.
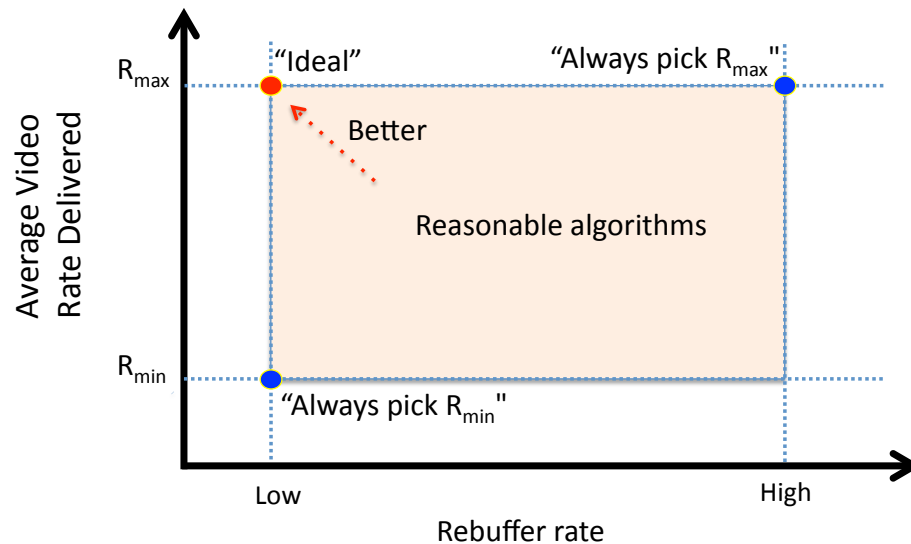
Figure 1.4: The goal of ABR algorithms: maximize the video quality and minimize rebuffering events.

1.3 Challenges of Existing Rate Selection Algorithms

ABR algorithms need to balance two overarching goals. On one hand, they try to maximize the video quality by picking the highest video rate the network can support. On the other hand, they try to minimize rebuffering events. It is easy for a streaming service to meet either one of these two objectives on its own. To maximize video quality, a service could just stream at the maximum video rate R_{max} all the time. Of course, this would risk extensive rebuffering. To minimize rebuffering, the service could just stream at the minimum video rate R_{min} all the time—but this extreme would lead to low video quality. The design goal of an ABR algorithm is to *simultaneously* obtain high performance on both metrics in order to give users a good viewing experience, as shown in Figure 1.4.

Existing ABR algorithms approach this design goal by picking a video rate based on capacity estimation. However, accurate estimation is challenging, especially in an environment with highly variable throughput. Figure 1.5 is a sample trace reported by a Netflix video player, showing how the measured throughput varies wildly from 17Mb/s to 500kb/s. The throughput is measured at the video player, and each point represents the average throughput over each video chunk download. This variation has a significant impact on
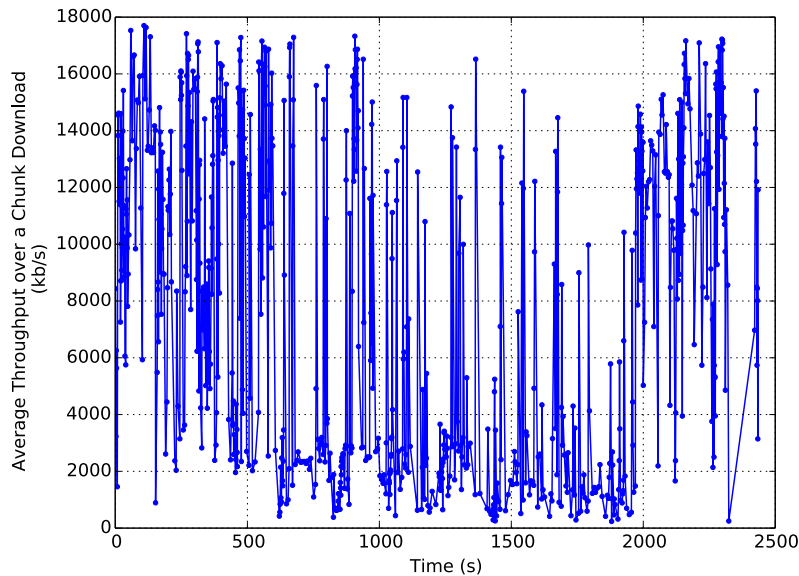
Figure 1.5: A sample throughput trace of a Netflix video session. This session is originated from a major US ISP and the throughput is reported by Netflix's browser-based player in June, 2013. Each point represents an average throughput over a video chunk download. End-to-end throughput is often highly variable within a session.

customers: approximately 10% of sessions in this service experience at least this much variation, and 22% of sessions experience at least half as much variation.[2] Variation can be caused by many factors, such as WiFi interference, congestion in the network, congestion in the client (e.g., anti-virus software scanning incoming HTTP traffic), or an overloaded video server. Whatever the cause, with capacity varying unpredictably, it is challenging to accurately estimate future capacity from past and present observations.

Other factors make it hard to even measure *current* throughput, let alone predict future throughput. For example, ABR algorithms often operate on top of a third party framework (e.g., Flash, Silverlight, HTML5), which hides accurate per-packet timing information from the ABR algorithm by delivering video to the browser in video chunks a few seconds long. Inaccurate throughput estimates lead to poor behaviors by video clients, such as rebuffers [9] and "unfair" link sharing between multiple video players [16, 21].

Many techniques have been proposed to work with inaccurate estimates, by incorporating information about the playback buffer. Some leverage control theory to adjust

---

[2]The variation is designed to be the ratio of 75th to 25th percentile throughput; which is 5.6 for this trace.
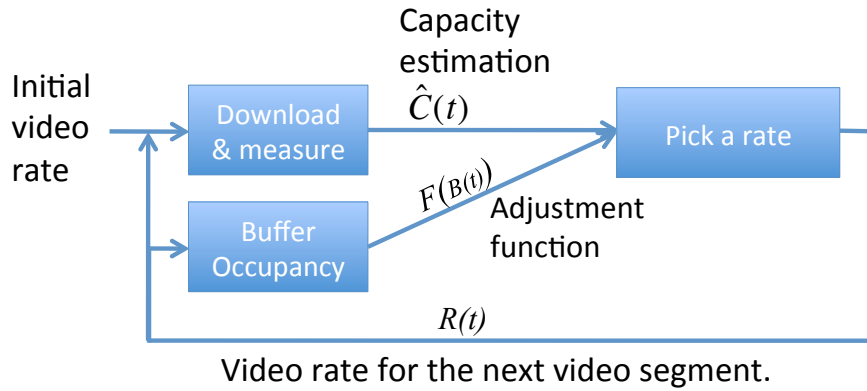
Figure 1.6: The control flow in current ABR algorithms. They discount the capacity estimation by applying a safety margin based on the buffer occupancy.

the capacity estimation based on the buffer occupancy [8, 35], some smooth the quality degradation according to the buffer occupancy [26], and some randomize chunk scheduling depending on the buffer occupancy to have better samples of the channel [16].

At a high level, we can capture existing approaches using the abstract design flow in Figure 1.6. The client measures how fast chunks arrive to estimate capacity, $\hat{C}(t)$. The estimate is optionally supplemented with knowledge of the buffer occupancy, which we represent by an *adjustment* factor $F(B(t))$, a function of the playback buffer occupancy. The selected video rate is $R(t) = F(B(t))\hat{C}(t)$; different designs use different adjustment functions $F(\cdot)$.

When the buffer contains many chunks, $R(t)$ can safely deviate from $C(t)$ without triggering a rebuffer. The client can "aggressively" try to maximize the video quality by picking $R(t) = \hat{C}(t)$.

When the buffer is low, on the other hand, the client should be more "conservative", deliberately underestimating capacity so as to pick a lower video rate and quickly replenish the buffer. In this case, designing the adjustment function is much harder, as the following analysis shows. Consider the case when there is only one chunk in the buffer. The requested chunk (V seconds) must arrive before the current chunk plays, else the buffer will run dry. In other words, we require $VR(t)/C(t) < B(t)$, where $VR(t)$ is the chunk size in bytes.

Thus, the selected video rate $R(t)$ needs to satisfy:

$$R(t) < \left(\frac{B(t)}{V}\right) C(t)$$

to prevent rebuffers. Replacing the selected video rate $R(t)$ with $F(B(t))\hat{C}(t)$ in the above inequality, we get the following requirement on $F(B(t))$ to avoid rebuffers:

$$F(B(t)) < \left(\frac{B(t)}{V}\right) \left(\frac{C(t)}{\hat{C}(t)}\right) \text{ for all } t. \tag{1.1}$$

This tells us we must pick $F(V)$ to be smaller than the *worst case ratio* of $C(t)$ to $\hat{C}(t)$. Unfortunately, $C(t)/\hat{C}(t)$ is tiny if the throughput is varying wildly; and since we have to choose F without knowing the actual capacity that will be observed, it leads to a very conservative algorithm. For example, in Figure 1.5, the ratio $C(t)/\hat{C}(t)$ can be as small as 0.03 (500 kb/s $< C(t) <$ 17 Mb/s). In other words, for this session, we need to pick $F(V) \leq 0.03$ to prevent rebuffers, and the video rate will be just 3% of the rate we could pick with an accurate estimate. Worse, if $F(.)$ makes us pick a rate lower than the minimum video rate available, the constraint becomes impossible to meet. In practice, large throughput variation within a session is not uncommon. A random sample of 300,000 Netflix sessions shows that roughly 10% of sessions experience a median throughput *less than half of the 95th percentile throughput.* When designing an ABR algorithm, the service provider needs to choose a $F(\cdot)$ that works well for *all* customers, with both stable and variable throughput.

However, the notion of buffer-based adjustment used in current schemes is quite suggestive: note that the occupancy of the playback buffer is the primary state variable we are trying to manage. This inspires the following question: namely, can we take the design to its logical extreme, and choose the video rate based *only* on the playback buffer occupancy?

In this thesis, we propose an alternative buffer-based approach: we start the design by *only* using the playback buffer occupancy, and then ask *when* capacity estimation can be helpful. This design process leads to two separate phases of operation: during the steady-state phase, when the buffer encodes adequate information, we choose the video rate based only on the playback buffer; during the startup phase, when the buffer contains little information, we augment the buffer-based design with capacity estimation. We will show that the buffer occupancy is in fact the primary state variable that an ABR algorithm

should control at the steady state. By focusing on the buffer, we can not only provide performance guarantees, but also avoid the difficulties of handling estimation errors.

1.4 Contributions

The contributions of this thesis is as follows. First, we identify the problems in the current practice and understand the interactions between network layers. This part of the thesis was published in ACM Internet Measurement Conference in 2012 and received the IRTF applied network research prize in 2013. Second, we propose an alternative buffer-based approach, which picks a video rate mainly based on the playback buffer occupancy. Through formal analysis, we show that this approach is able to provide performance guarantee at the steady state. This part of the thesis was published in ACM SIGCOMM Future Human-Centric Multimedia Networking (FhMN) workshop in 2013. Finally, we verify the effectiveness of this approach through a real-world deployment in Netflix. We deploy buffer-based ABR algorithms in Netflix's browser-based player and conduct a series of experiments with millions of real users during May–September 2013. Our own investigation reveals that capacity estimation is unnecessary in steady state; however using simple capacity estimation (based on immediate past throughput) is important during the startup phase, when the buffer itself is growing from empty. We find that buffer-based algorithms can reduce the rebuffer rate by 10–20% compared to the production algorithm in Netflix, while improving the steady-state video rate. This part of the thesis will be published in ACM SIGCOMM Conference in 2014. Since 2012, Netflix has incorporated some designs from the buffer-based approach in their browser-based video player, serving over 40 millions Netflix subscribers. We currently continue to work with Netflix to productize the rest of the designs.

1.5 Thesis Outline

The remaining thesis is organized as follows. In Chapter 2 and 3, we present case studies and identify the challenges in the current practice. In Chapter 4, based on the observations made in the previous chapters, we will motivate and propose the pure buffer-based approach. In Chapter 5, we will describe our deployment of the buffer-based ABR algorithm in Netflix and discuss the experimental results. Based on the results, we will also discuss design principles for future ABR algorithm designers. In Chapter 6, we will discuss related works

and other efforts to better support Internet video. Finally, we will conclude and discuss a future roadmap in Chapter 7.

Chapter 2

Case Study: Suboptimal Video Quality

In Chapter 1, we showed that capacity estimation is often inaccurate in environments with highly variable capacity. Hence, existing algorithms often need to apply a safety margin to the noisy estimates. However, it is hard to determine a safety margin that can simultaneously maximize video quality and minimize the rate of rebuffer events. A given safety margin can be too conservative for some customers and too aggressive for others. Being conservative leads to suboptimal video rate, while being aggressive leads to unnecessary rebuffers. In the following two chapters, we will present two case studies and investigate the consequences of being either conservative or aggressive.

In this chapter, we show that being conservative can trigger a feedback loop between network layers, leading to undesirably variable and low-quality video. We call this phenomenon the *downward spiral effect*, and we observed the effect in all three video streaming services we studied: Hulu, Netflix, and Vudu. We first describe how the three video services work, then we demonstrate how they all experience the downward spiral effect. Next, we explain how this effect is triggered by the interaction between ABR algorithms at the application layer and TCP logics at the transport layer. Finally, we provide recommendations to avoid the downward spiral. These recommendations are the precursor of the buffer-based approach. They are now incorporated in the ABR algorithm in Netflix and have demonstrated quality improvements in the real world.

11

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

Table 2.1: 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	SD: 235, 375, 560, 750, 1050, 1400, 1750 HD: 2350, 3600
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 2.2: Summary of the available playback rates for each of the three services.

2.1   The Three Streaming Services

The measurement study was done on three popular HTTP-based video streaming services, Hulu, Netflix, and Vudu, between March and May, 2012. At the time of study, both Hulu and Netflix offered monthly subscription services, while Vudu offered pay-per-view service. In 2012, Hulu had 3 million paid subscribers [15], Netflix had 25 million paid subscribers [27], and Vudu had one of the largest libraries of streamed movies and was ranked as one of the best streaming services by Consumer Reports [33]. Throughout this chapter we will refer to the services by the names A, B, and C. All of the 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 this work is not a comparison of the services; rather, we wish to show that existing algorithms must wrestle with the problem of noisy observations above HTTP when picking a video rate based on capacity estimation.

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

Service A: The measurements for Service A are based on a web-browser client. The Service A client sends HTTP byte range requests to the CDN, requesting 4-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 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 transport protocol; thus, our measurements for Service B are based on its client on the Sony PlayStation 3 (PS3), which operates over HTTP. Service B's PS3 client also requests one chunk at a time, but each chunk is stored as a separate file and each request for a new chunk 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; the player 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 compared to its desktop client. 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 clients request new chunks, 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 2.1 and Table 2.2 summarize the three services.

2.2 Measurement Setup

In this section we describe our experimental setup and measurement approach. In particular, we describe how to control bottleneck bandwidth and eliminate path variance through a local proxy, how to identify the video playback rate, and how to create the competing flow.

2.2.1 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, all of the video streams must pass through a rate limiter between the CDN and the client. Figure 2.1 shows how we place a NetFPGA machine [24] in-line to limit bandwidth. In most cases, we limit the bandwidth to 5Mb/s and the buffer size to 120kbit, which is sufficient to sustain 100% throughput with a 4–20 ms RTT. We use these configuration parameters throughout the chapter, unless stated otherwise. The competing flow always passes through the NetFPGA too, and it shares the bottleneck link with the video stream.
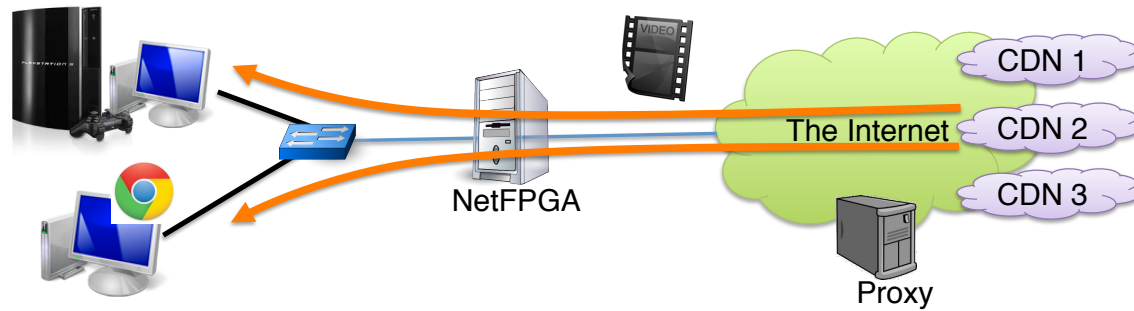
Figure 2.1: The experimental setup. We control the bandwidth with a NetFPGA bandwidth limiter. We also use a proxy server in experiments that require end-to-end control.

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 Internet paths.

2.2.2 Video Playback Rate Identification

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 these rates to validate our mapping.

Service B: Unfortunately, Service B does not 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 the parameter set to 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. Indeed, the value of the parameter 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 2.2; some playback rates may not be available for some videos.

2.2.3 The Competing Flow

The competing flow is a TCP flow doing a long file download. To eliminate any unfairness due to variations in network path properties, we ensure 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 the same termination point (the server or the load-balancer) as the video flow. For Service B, because the files are stored as small segments, an open-ended request creates only 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 [32] over TCP.

2.3 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 2.2. 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 gain 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 bitrate of the last video chunk. Specifically, we calculate the throughput upper bound as

(a) Service A. Network bottleneck set to 5Mb/s.

(b) Service B. Network bottleneck set to 5Mb/s.

(c) Service C HD. Network bottleneck set to 22Mb/s. (d) Service C SD. Network bottleneck set to 5Mb/s.
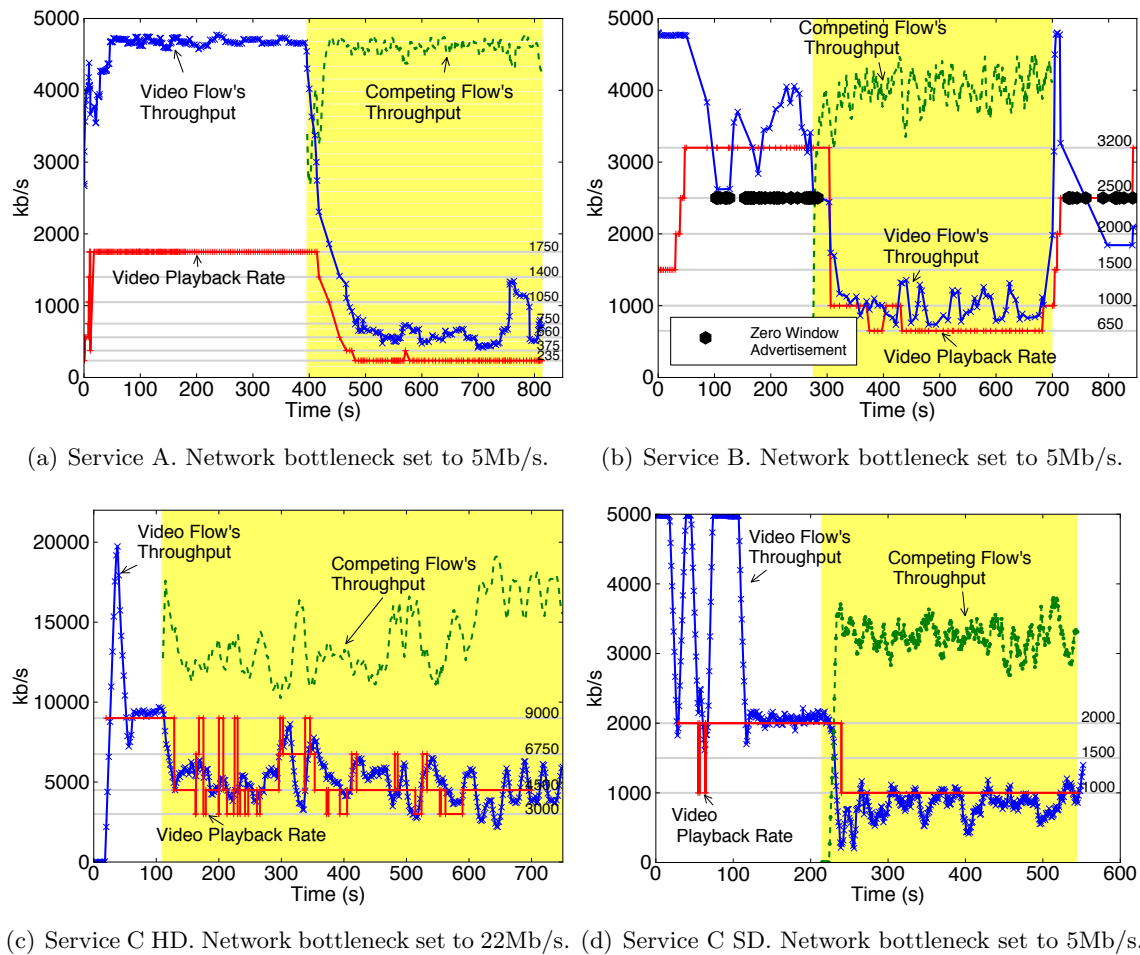
Figure 2.2: The downward spiral effect in the three services.

the size of a received video chunk divided by the time it took to arrive, i.e., the time from when the first byte arrived until the last byte arrived. Thus, the initial server response time is excluded from the throughput calculation. This throughput upper bound is plotted as the blue lines with cross markers in the graphs. 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 that the client does such a poor job. Let's now see what goes wrong for each service in turn. For ease of discussion, we will use *video throughput* to refer to the throughput a client perceived by downloading a video chunk.

2.3.1 Service A

Figure 2.2(a) shows the playback rate of a Service A video session along with the client's video 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%), the client was 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 the *same* video file 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 the TCP congestion control algorithm and not capped by the video client.[1]

Why does the 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?

---

[1]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.
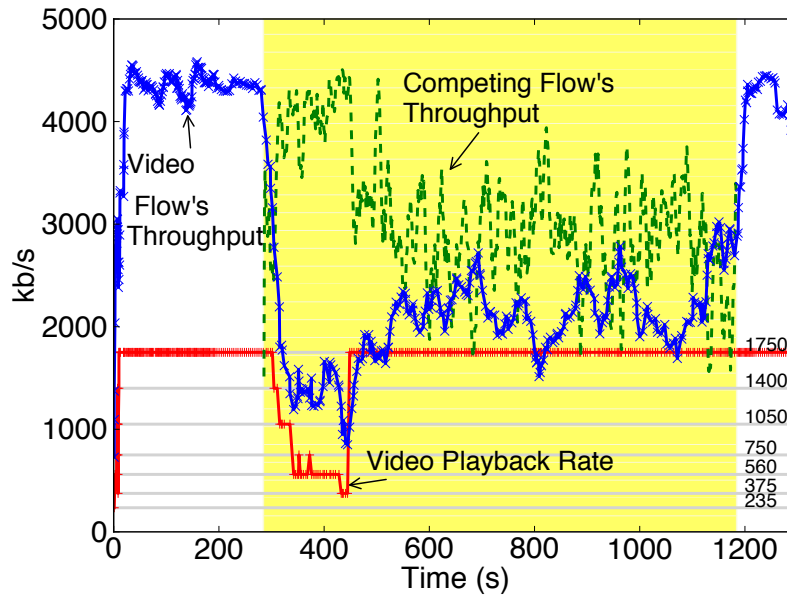
Figure 2.3: (Service A) Confirmation of the available bandwidth. When the ABR algorithm is disabled, the client is able to to sustain the highest video rate. This indicates that ABR algorithms might be the cause of the downward spiral effect.

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 2.3 shows the results. Interestingly, the client maintains a playback rate of 1750kb/s without causing rebuffers, and 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.

2.3.2   Service B

Figure 2.2(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 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.

2.3.3   Service C

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

In the HD experiment, as shown in Figure 2.2(c), we set the bottleneck bandwidth to 22Mb/s. The client starts by picking 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 throughput converges to the same value as the playback rate. We start the competing flow at 100 seconds, and it downloads 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 2.2(d). When we start the competing flow, the video client drops down to 1000kb/s even though its share is 2.5Mb/s. As Service C only offers three SD rates, we focus on its HD service in the rest of the chapter.

2.4   Walking the Downward Spiral

To understand how the downward spiral occurs, 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 Figure 2.2(a)) and then describe how the other two services differ.

2.4.1   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 2.4(a) shows

(a) TCP throughput.
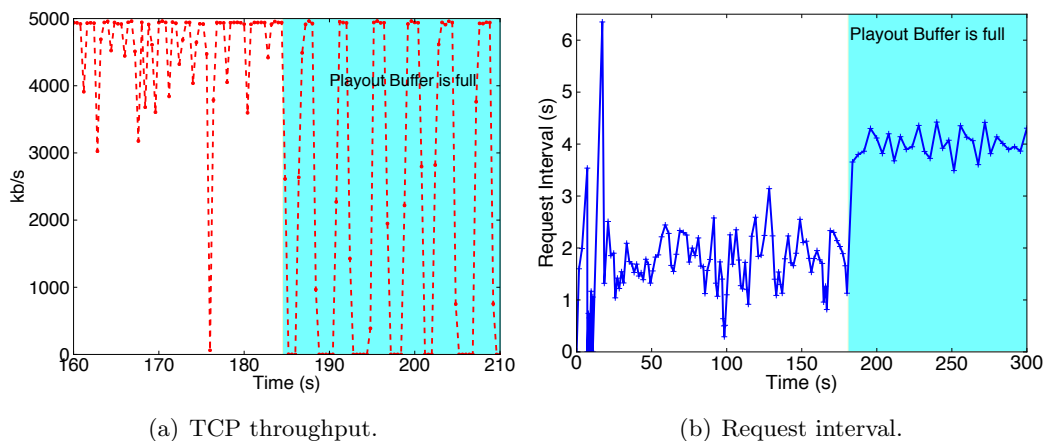
(b) Request interval.

Figure 2.4: (Service A) The TCP throughput and video request interval before and after the playback buffer fills at 185 seconds. After the buffer is full, the client pauses the next request until there is enough space freed up in the buffer. As a result, we observe an ON-OFF traffic pattern from the TCP throughput.

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, albeit 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 2.4(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 [4, 5]. Even though the client is using an existing persistent TCP connection, the *cwnd* needs to ramp up from slow start for each new chunk download.

It is natural to ask if the repeated dropping back to slow-start reduces the client's video 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 video throughput for many requests. We set the bottleneck link rate to 2.5Mb/s, 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 2.5(a) shows the CDF of the client's video throughput for requests corresponding to various playback rates. The video throughput is pretty accurate: except for some minor variation, the video throughput accurately reflects the available bandwidth, and explains why the client picks the correct rate.

(a) Service A with no competing flow.   (b) Service A with one competing flow.
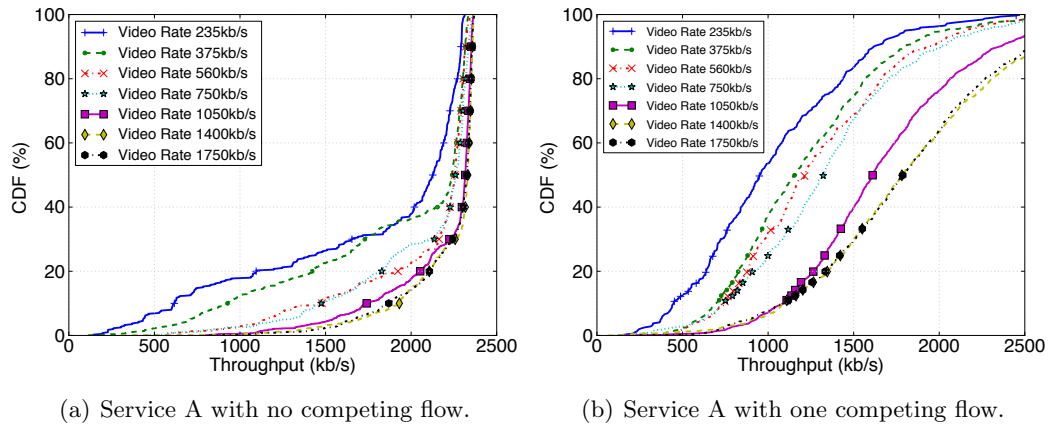
Figure 2.5: (Service A) The throughput at HTTP layer, with and without a competing flow. In both figures the available fair share of bandwidth for the video traffic is 2.5Mb/s. When there is no competing flow (Figure 2.5(a)), the bottleneck bandwidth is set to be 2.5Mb/s. When there is one competing flow (Figure 2.5(b)), the bottleneck bandwidth is set to 5Mb/s so that the fair share remains 2.5Mb/s. When competing with another flow, some chunk downloads are able to get more than their fair share; in these cases, 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 do not end with 100% at 2.5Mb/s in Figure 2.5(b).

2.4.2 The Trigger: With a Competing Flow

Things go wrong when the competing flows starts (after 400 seconds). Figure 2.5(b) shows that the client's video throughput is mostly lower than its fair share (2.5Mb/s) when there is a competing flow. 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 chunk. Figure 2.6(a) shows how *cwnd* evolves, starting from the initial value of 10 at 1.5 seconds, then *repeatedly being dropped 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.

(a) A 235kbps chunk.

(b) Five contiguous 235kbps chunks concatenated into one.
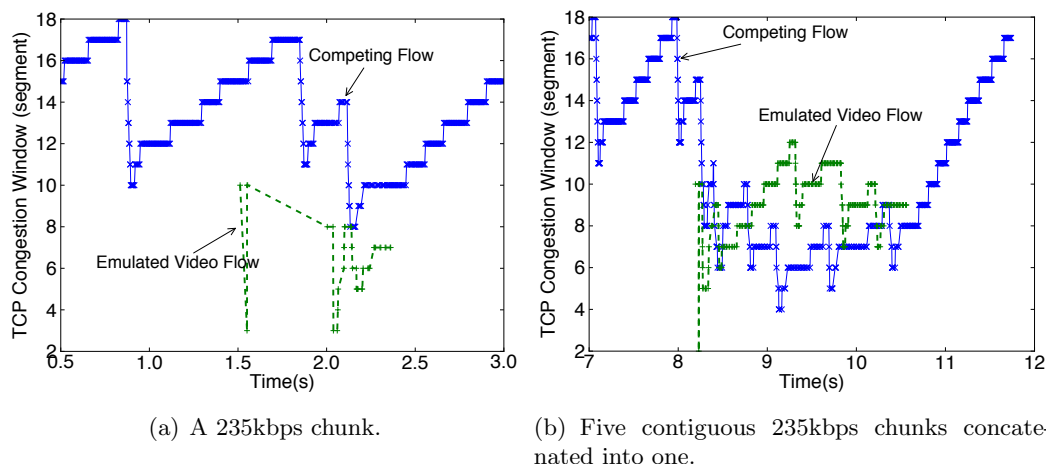
Figure 2.6: (Service A) The evolution of *cwnd* for different chunk sizes. With a larger chunk size, the *cwnd* has a greater chance to reach the correct steady state value.

For comparison, and to understand the problem better, Figure 2.6(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 it also has a much greater likelihood of reaching the correct steady state value.

Now that we know the video throughput tends to be low (because of TCP), we would like to better understand how the client *reacts* to the low throughput. We can track the client's behavior as we steadily reduce the available bandwidth, as shown in Figure 2.7. We start with a bottleneck link rate of 5Mb/s (and no competing flow), drop it to 2.5Mb/s (to mimic a competing flow), and 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. 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. Consider a client streaming at a playback rate of 1750kb/s; the median video throughput it perceives is 1787kb/s, as shown in Figure 2.5(b). According to Figure 2.7, with an available bandwidth of 1787kb/s, the client reduces its playback rate to 1050kb/s. Thus, we expect the video rate will go 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 perceives. The problem is that because Service A observes the throughput *above* TCP, it is not aware that TCP itself is having trouble reaching its fair share of the
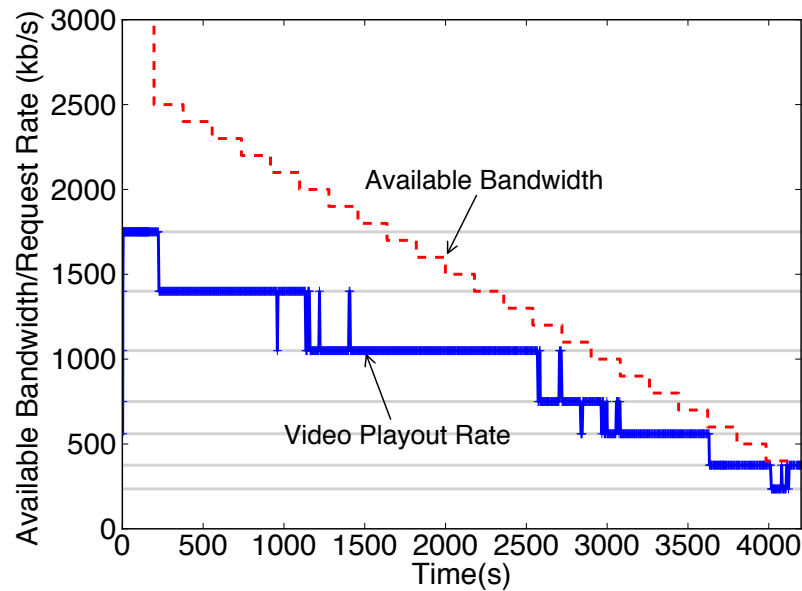
Figure 2.7: (Service A) The choice of video rate under different available bandwidth. The horizontal gray lines are the available video rates provided by the service.

bandwidth. Coupled with a (natural) tendency to pick rates conservatively, the rate drops down.

2.4.3 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 chunk is smaller (in bytes), as shown in Figure 2.8. With a smaller chunk, the video flow becomes more susceptible to perceiving lower throughput, as shown in Figure 2.5(b). With a lower throughput, the client reduces the playback rate, creating the vicious cycle shown in Figure 2.9. 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 playback 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
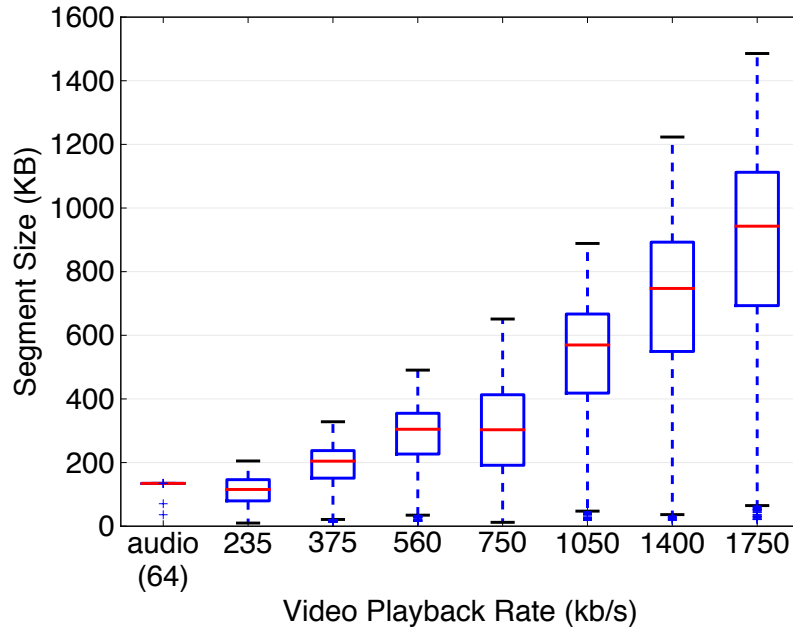
Figure 2.8: (Service A) The chunk size for different video rates. Each chunk contains four seconds of video. When the video rate is lower, the chunks are smaller in bytes.

buffer, so the video flow sees high packet loss when it returns 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 a consequence, the video flow underestimates the available bandwidth and switches to a lower video rate. When switching to a lower rate, the client requests a smaller video chunk, which makes the video flow further underestimate the available bandwidth, forming a vicious cycle.

2.4.4   With Different Network Conditions

Our experiments so far were all for the same CDN. Figure 2.10 shows a different behavior when using a different CDN. This is because different network path properties, such as shorter RTTs, allows the video traffic to get a higher throughput in the presence of a competing flow, as shown in Figure 2.11. As a result, the Service A client picks a higher rate (1050kb/s).

To understand how prevalent the downward spiral effect is in home networks, we asked 10 volunteers to rerun this experiment with Service A in their home networks connected
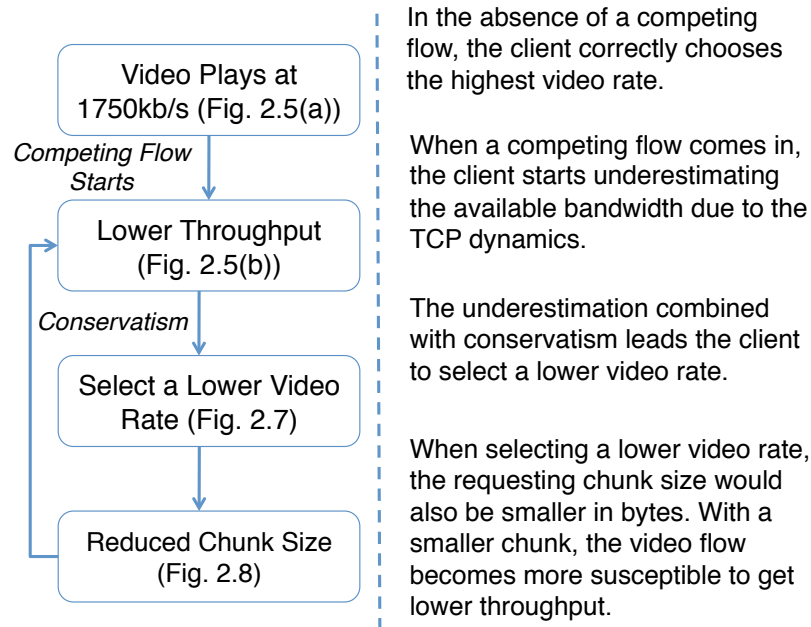
Figure 2.9: (Service A) The feedback loop.

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 in the presence of a competing flow, seven people reported a rate of only 235kb/s-560kb/s.

2.4.5 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 chunk. When its video playback buffer is full, the client stops taking data from the TCP socket buffer. Eventually, the TCP socket buffer also fills and triggers TCP flow control to pause the server by sending a *zero window advertisement*. In Figure 2.2(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. Figure 2.12(a) shows the CDF of the duration of the OFF periods. Almost all the pauses are longer than 200ms, 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; Figure 2.12(b) shows that over half of the time, it reads only 800kbytes, which is not enough for the *cwnd*
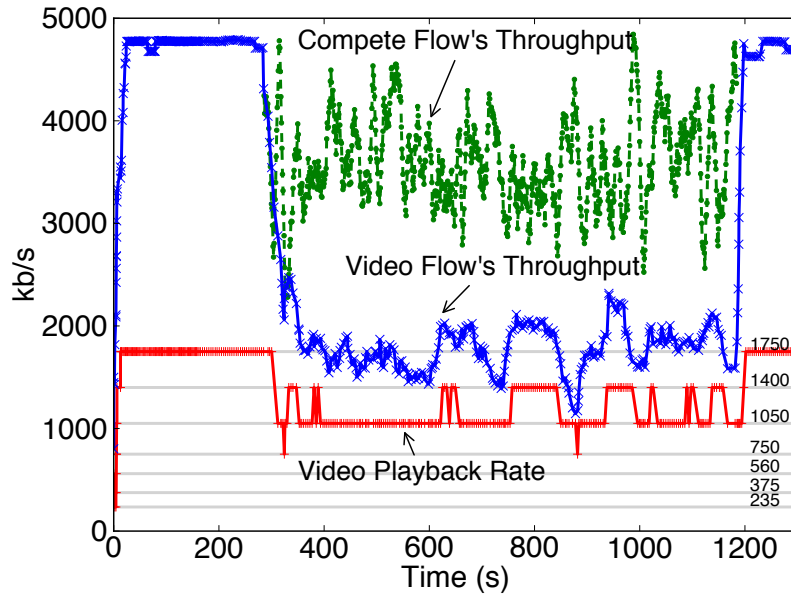
Figure 2.10: (Service A) The downward spiral effect under different network path properties. When streaming from another CDN, the playback rate stabilizes at 1050kb/s.
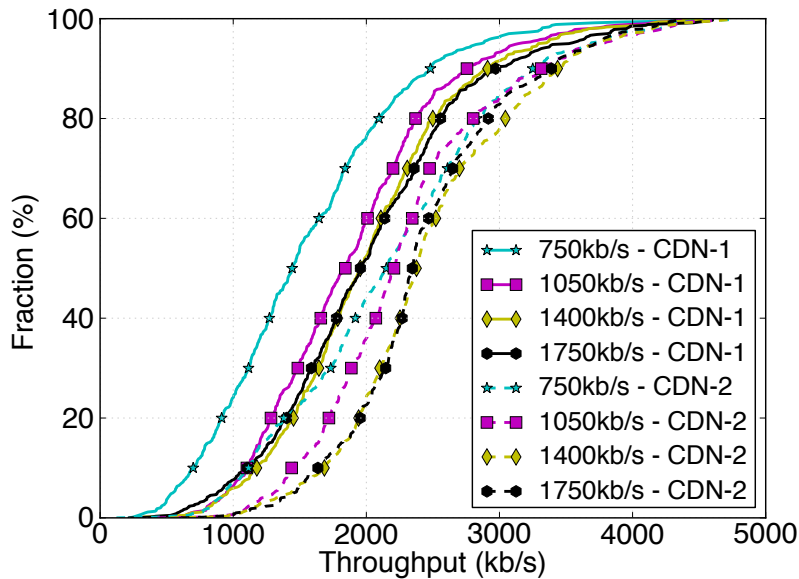
Figure 2.11: (Service A) The video throughput under different network path properties. Different network path properties, such as shorter RTTs, allows the video traffic to get a higher throughput in the presence of a competing flow.
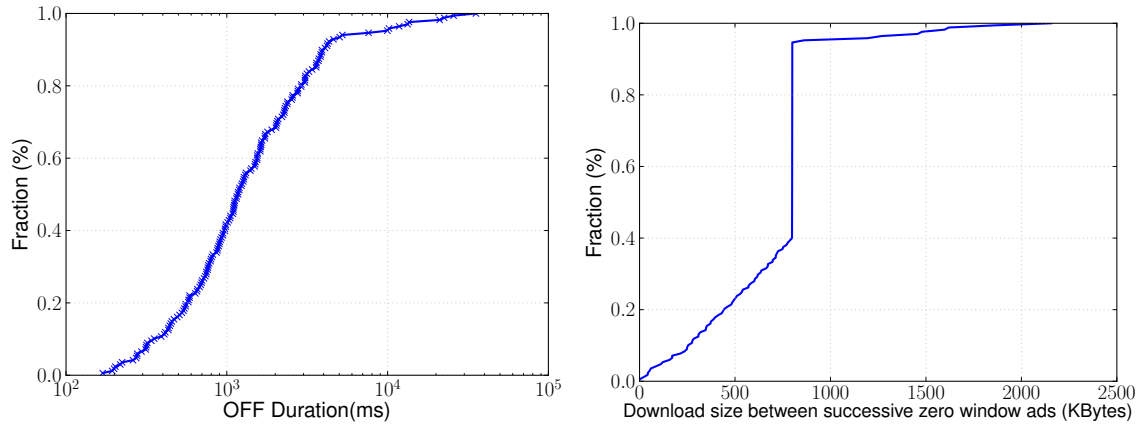
(a) (Service B) The duration of the OFF periods. Almost all the OFF periods in a video session are greater than RTO (200ms).

(b) (Service B) The bytes requested during the ON periods. When the video stream is receiver-limited, the client does not request many bytes during an ON period.

Figure 2.12: (Service B) The ON-OFF behavior at the TCP level.

to climb up to its steady state before the next OFF period. Figure 2.2(b) and Figure 2.13(b) show the result, that the TCP throughput is only around 1Mbps to 1.5Mbps, causing Service B to pick a video rate of 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 Figure 2.14.

Similar to Service A, the chunk size gets smaller with decreasing playback rate – from 3MByte to 1MByte, as shown in Figure 2.15. However, chunk size does not play a significant role in the downward spiral effect of Service B, as the ON-OFF behavior is at the TCP level and the number of bytes for each ON period is not determined by chunk size.

2.4.6 Service C

Service C performs an open-ended download, instead of going chunk-by-chunk. Thus, it is not constrained by chunk size and does not pause between HTTP requests. The download is slowed down only when the receive window fills and the client 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

(a) Service B with no competing flow.   (b) Service B with one competing flow.
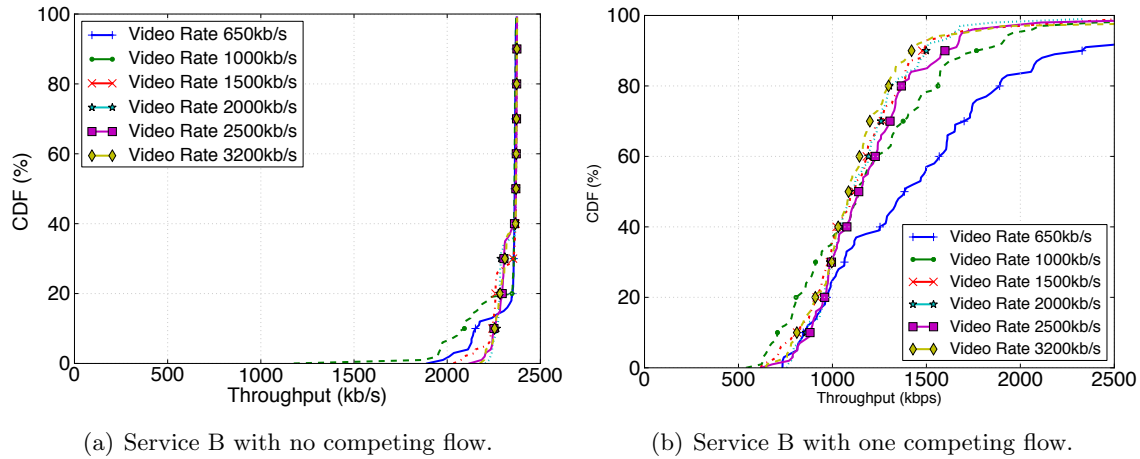
Figure 2.13: (Service B) The TCP throughput before and after the presence of a competing flow. The available fair share of bandwidth for the video traffic is 2.5Mb/s in both figures. In the presence of a competing flow, the video flow gets a much less throughput than its fair share.
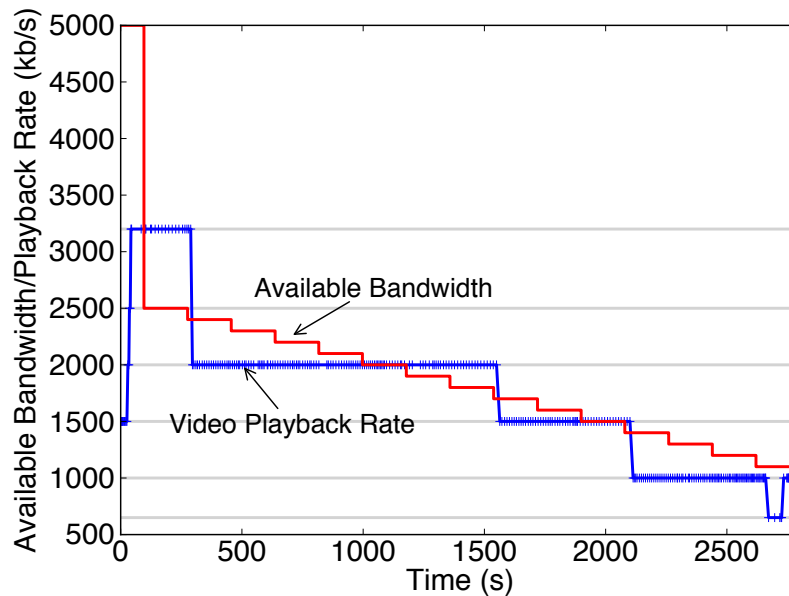
Figure 2.14: (Service B) The choice of video rate under different available bandwidth. While Service B is not conservative, the client is limited by its TCP throughput.
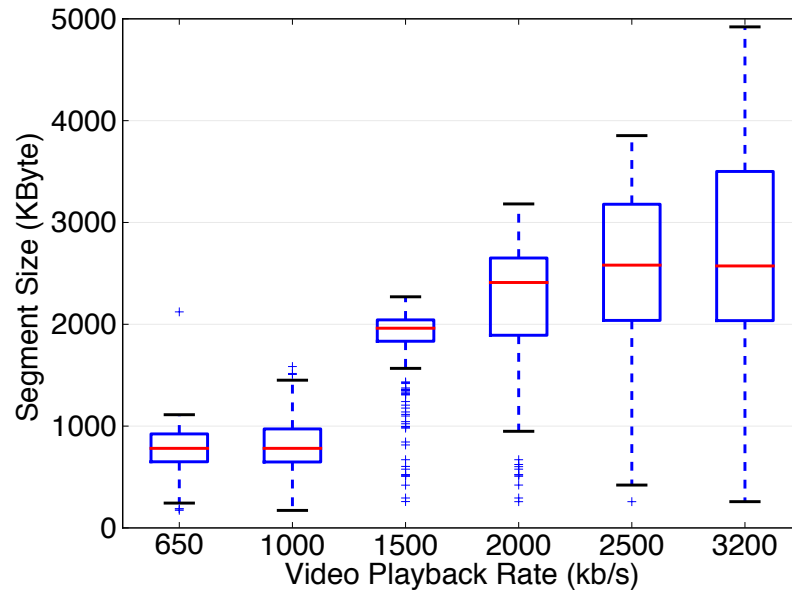
Figure 2.15: (Service B) The chunk sizes under different video playback rates.

conservativeness and being sensitive to temporal behaviors in TCP. Figure 2.16 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 2.2(c), Service C regulates itself at 9Mb/s after its playback 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 perceive less than 9Mb/s of available bandwidth. As a consequence, the video flow steps down to the video rate of 3Mb/s. Because Service C does an open-ended download, it has to reset the current TCP connection to stop downloading the current video file when switching to a different video rate. The client then starts a new connection for the file with the newly selected rate. This would make the video flow perceive less bandwidth during the transition, as TCP needs to go through the phase of connection establishment and slow start. As a result, whenever Service C switches to a higher video rate, this overhead of transition makes the client perceive less bandwidth and bounce back to a lower video rate.
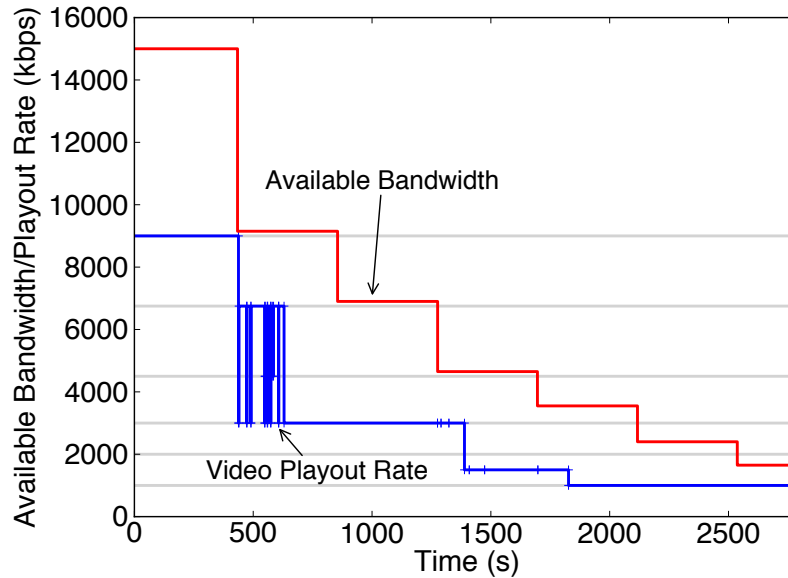
Figure 2.16: (Service C) The choice of video rate under different available bandwidth. The client is conservative in its rate selection.

2.5   Verifying the Explanation

We can verify the cause of the downward spiral by demonstrating that a small modification to the ABR algorithm enables a client to achieve its fair share of video bandwidth. We will discuss the recommendations in the next section.

2.5.1   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 with our own client algorithm. The three services keep their algorithms secret, but our measurements provide a reasonable baseline. For Service A, Figure 2.7 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.
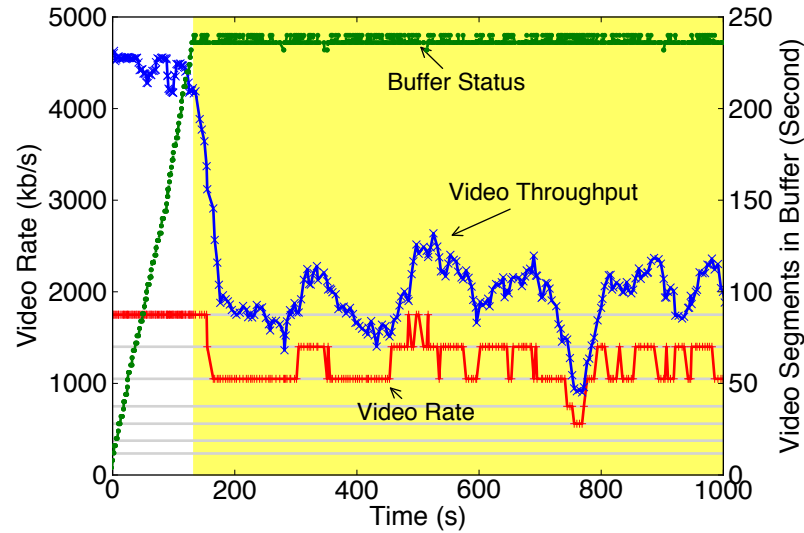
Figure 2.17: Replicating the downward spiral effect with our custom client. The custom client is modeled after the client of Service A. They are equally conservative, and our client estimates the future capacity with a 10-sample moving average filter.

To closely mimic the Service A client, our custom client requests video chunks with the same sizes from the same locations in the CDN: we capture the chunk map given to the client after authentication, which locates the video chunks for each supported playback rate. Hence, our custom 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. As 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 validate our understanding of downward spiral by considering three changes: (1) being less conservative, (2) changing the filtering method, and (3) aggregating chunks.

2.5.2 Validating our Custom Client

Figure 2.17 shows the custom client in action. After downloading each chunk, the custom client selects the playback rate based on Service A's conservative rate selection algorithm, observed in Figure 2.7. 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
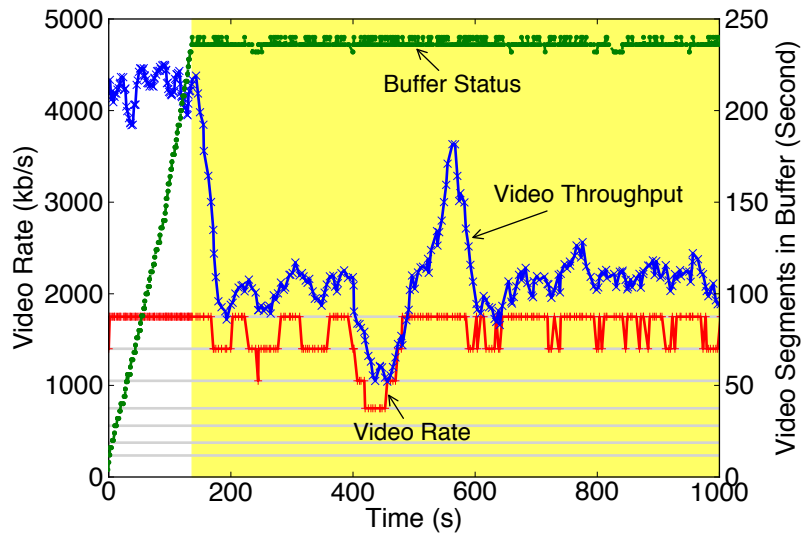
Figure 2.18: Breaking the spiral with a less conservative client. This custom client still estimates the future capacity with a 10-sample moving average filter, but it is much less conservative than the client of Service A.

fluctuates over the course of the movie. The downward spiral does not bottom out, which we suspect is due to some subtle differences between Service A's algorithm and ours.

2.5.3   Breaking the Spiral: 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 client requests a video rate of at most 1.2Mb/s when it perceives 2.0Mb/s, while 10% means it requests at most 1.8Mb/s when perceiving 2.0Mb/s. According to Figure 2.7, Service A requests video rate with a conservatism of approximately 40%. Figure 2.18 shows that by being less conservative, the video rate is higher and the playback buffer still stays full. Note that even though the algorithm is less conservative, the underlying TCP ensures that the algorithm stays a "good citizen" and only gets its fair share of available bandwidth. This result verifies that conservatism contributes to the downward spiral effect.
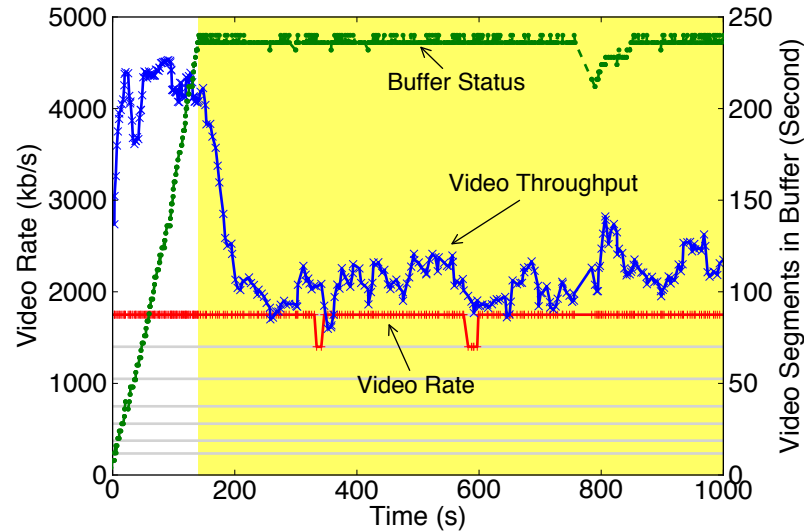
Figure 2.19: Breaking the spiral with a better filtering technique. This custom client is not only much less conservative, but also using a better filter. It estimates the future capacity with an 80th-percentile filter, instead of a 10-sample moving average filter.

2.5.4 Breaking the Spiral: 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. In place of averages, we consider medians and quantiles to reduce the vulnerability to outliers. Figure 2.19 shows what happens if we use the 80th-percentile of measured rate of the past ten chunk downloads. Variation is greatly reduced, and the majority of the movie plays at the highest-available rate. The playback buffer has small fluctuations, but it is still far from a rebuffer event. This result verifies that poor capacity estimation also contributes to the downward spiral effect.

2.5.5 Breaking the Spiral: Bigger Chunks

As noted earlier, bigger chunks provide better estimates of the available bandwidth, allowing TCP to escape slow-start. Figure 2.20 shows what happens if our client aggregates five requests into one. With the larger chunk size, the video throughput is more stable, and both the playback rate and buffer size are more stable. This result verifies that the combination of ON-OFF behavior and small chunk size also contributes to the downward spiral effect.
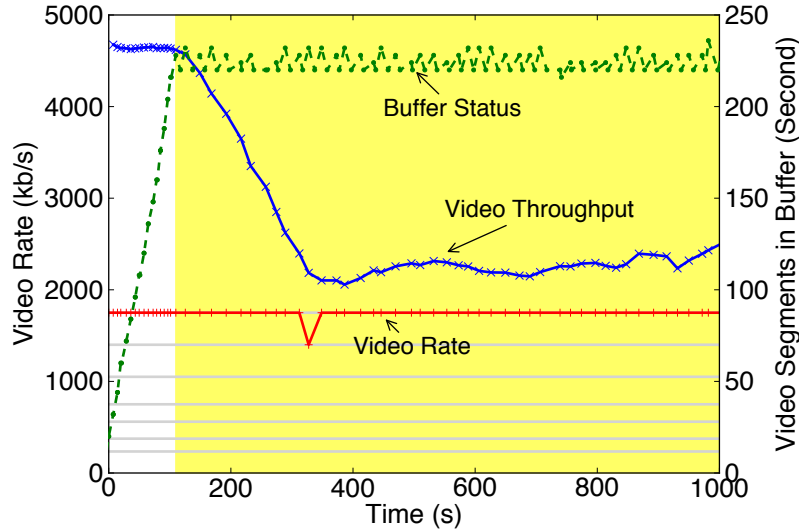
Figure 2.20: Breaking the spiral with bigger chunks. This custom client requests videos with an increased chunk size (5x).

In summary, the downward spiral effect happens because the video flow gets much less throughput than its fair share. Requesting larger chunks helps TCP reach its fair share after each OFF period and improves the video throughput. Picking higher rates less conservatively and filtering measurements more carefully can also help improve video quality. But more fundamentally, an ABR algorithm should ensure video flows get their fair share. Given TCP already has mechanisms to reach the fair share, ABR algorithms do not need to reinvent the wheel. Instead, they should focus on letting TCP do its job and avoid interfering with TCP.

2.6   Our Recommendation

The ON-OFF behavior is the root cause of why a video client cannot get its fair share. Thus, an algorithm should prevent the ON-OFF behavior unless the capacity is higher than R_{max} and a full buffer is inevitable. In other words, the playback buffer should only be full $(B(t) = B_{max})$ when there is excessive capacity $(C(t) > R_{max})$.

It turns out that we can easily ensure this property by focusing on the buffer occupancy. From the discussion in Section 1.2, we know the buffer increases when $C(t) > R(t)$ and the buffer decreases when $C(t) < R(t)$. If we let $R(t) = R_{max}$ when the buffer is full
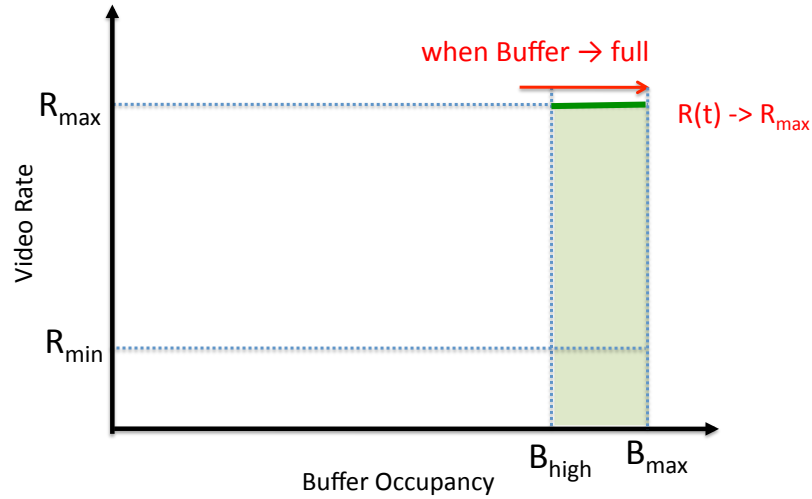
Figure 2.21: Our technique to avoid the ON-OFF traffic pattern: picking the highest video rate ($R(t) = R_{max}$) when the buffer is almost full ($B(t) > B_{high}$). By doing so, the buffer will only be full when there is more capacity than the highest video rate ($C(t) > R_{max}$). As the ON-OFF pattern only appears when the buffer is full, the ON-OFF pattern does not appear unless $C(t) > R_{max}$.

($B(t) = B_{max}$), the buffer only remains full ($B(t) = B_{max}$) when $C(t) \geq R_{max}$. Otherwise, the buffer decreases and $B(t) < B_{max}$ when $C(t) < R_{max}$.

This recommendation is counterintuitive and against the common wisdom. It is commonly believed that the buffer should be as full as possible, because higher buffer occupancy provides better rebuffer protection. However, by allowing the buffer to not reach full, and as a result, avoid ON-OFF behavior, TCP is able to get its fair share and improve video quality. We can still provide similar rebuffer protection by maintaining the buffer occupancy above a certain threshold. Figure 2.21 visualizes this recommendation in terms of the relationship between video rate and buffer occupancy.

We have shown that when competing with a long-lived TCP flow, the ON-OFF pattern can trigger the downward spiral. Other research also shows that when competing with other video players, the overlapping ON-OFF periods can confuse capacity estimation, leading to oscillating quality and unfair link share among players, as shown in Figure 2.22 [2, 16, 21]. In our recommendation, as the algorithm only requests R_{max} when the buffer approaches full, the ON-OFF traffic pattern appears only when the available capacity is higher than R_{max}. When competing with a long-lived TCP flow, our algorithm continues to request R_{max} when the ON-OFF pattern happens, avoiding the downward spiral. When competing
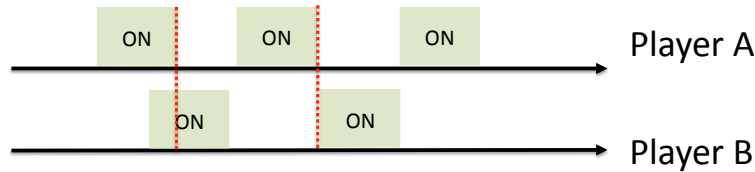
Figure 2.22: Another problem caused by the ON-OFF traffic pattern: the overlapping ON-OFF periods between competing video players can confuse ABR algorithms, leading to oscillating quality and unfair link share among players.
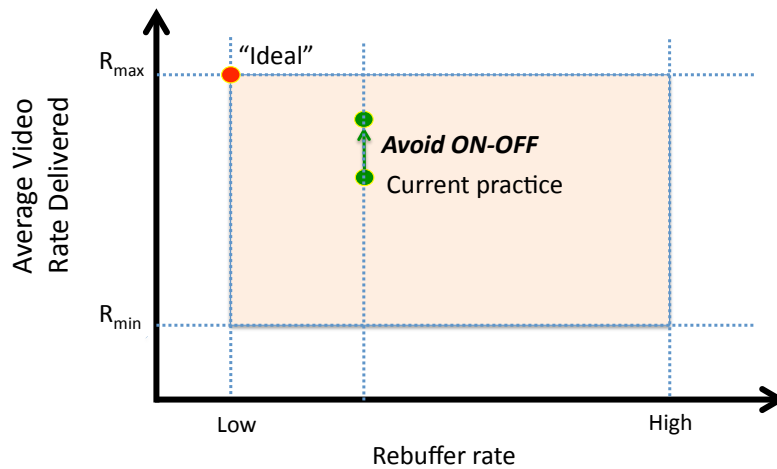
Figure 2.23: The real world impact of our recommendation. By avoiding the ON-OFF behavior, we improve the average video rate of the current practice, moving it closer to the ideal algorithm design.

with other video players, if the buffer is full, all players have reached R_{max} and so the algorithm is fair.

2.7   The Real World Impact and Netflix Deployment

The work described in this chapter was published in ACM Internet Measurement Conference in 2012 and awarded the IRTF Applied Networking Research Award in 2013. The work also drew much attention from the industry. When designing ABR algorithms, streaming service providers incorporate designs to avoid the downward spiral effect. Netflix tested a variant of our recommendation on 700,000 sessions in June, 2012. Their experimental results show that video rate is improved by 50 kb/s while maintaining a slightly lower rebuffer rate. Since

then, Netflix has incorporated this recommendation as part of its default algorithm in its browser-based player. This simple recommendation effectively moves the current practice towards a better algorithm design by improving the video rate, as shown in Figure 2.23.

In the next chapter, we will discuss how to further improve the current practice by reducing unnecessary rebuffers.

Chapter 3

Case Study: Unnecessary Rebuffers

In Chapter 2, we showed that a conservative safety margin often causes the downward spiral effect and leads to low-quality video. In this chapter, we will show that this same safety margin, while being conservative for some customers, can also be too aggressive for others and lead to unnecessary rebuffers.

Unnecessary rebuffers happen when an ABR algorithm chooses a video rate that is higher than the end-to-end system capacity can sustain. These rebuffers would be avoidable if the algorithm chose a lower video rate. In this chapter, we first investigate the frequency at which unnecessary rebuffers happen in a commercial service. With help from Netflix, we find that 20-30% of rebuffers seen in their streaming video service are unnecessary and can be eliminated through a better algorithm design. We further study the cause of unnecessary rebuffers through controlled experiments. Based on our observations, we then provide recommendations to avoid these rebuffers. Together with the recommendations in Chapter 2, these recommendations motivate the buffer-based approach for video rate adaptation.

3.1   The Prevalence of Unnecessary Rebuffers

To understand the prevalence of unnecessary rebuffers, we need to compare the rebuffer rate of the current practice with an algorithm that never unnecessarily rebuffers. Ideally, we could compare with a hypothetical optimal algorithm that never unnecessarily rebuffers,
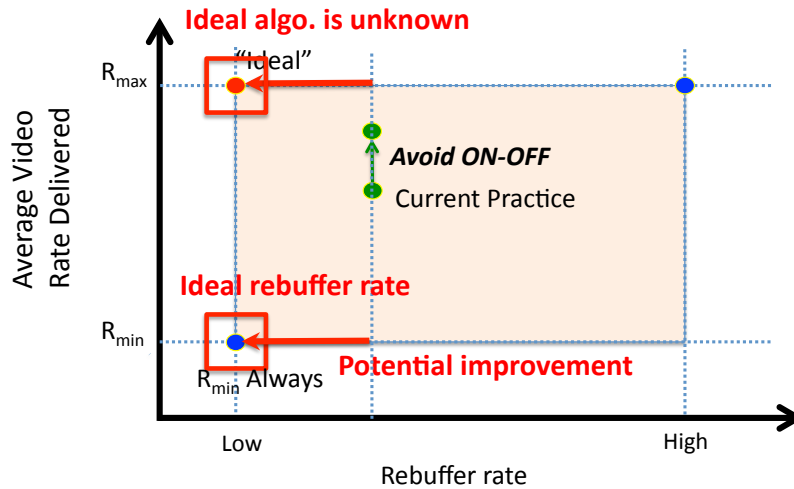
Figure 3.1: R_{min} *Always*: an alternative algorithm that never unnecessarily rebuffers.

yet always maximizes video rate. However, as the ideal algorithm is unknown, we need an alternative algorithm that never unnecessarily rebuffers, regardless the video rate it achieves. One option is to stream at the minimum video rate, $R_{\min}$, all the time. Although this choice leads to low video quality, it minimizes the chances of the buffer running dry and never unnecessarily rebuffers. This desirable property of minimal rebuffers occurs, because when streaming at $R_{\min}$, a rebuffer only happens when $C(t) < R_{min}$. We call this degenerate algorithm $R_{\min}$ *Always* and represent it in the design space in Figure 3.1.

With help from Netflix, we performed an A/B test using their browser-based player. This player has 240 seconds of playback buffer and downloads the ABR algorithm at the start of the video session. Although this player enjoys a bigger buffer than players on embedded devices, it does not have visibility into, or control of, the network layer. We randomly picked two groups of users around the world to take part in the experiments conducted between September 6th (Friday) and 9th (Monday), 2013.

Group 1 is our *Control* group, which uses Netflix's (then default) ABR algorithm.[1] The *Control* algorithm has steadily improved over the past five years to perform well under many conditions. The *Control* algorithm directly follows the design in Figure 1.6, and it is representative of how video streaming services work; other major services, such as Hulu [14] and YouTube [38], follow the same design. At the time of the experiment (2013), Netflix

---

[1]The ABR algorithm in commercial services keeps evolving and Netflix's current algorithm is now different.
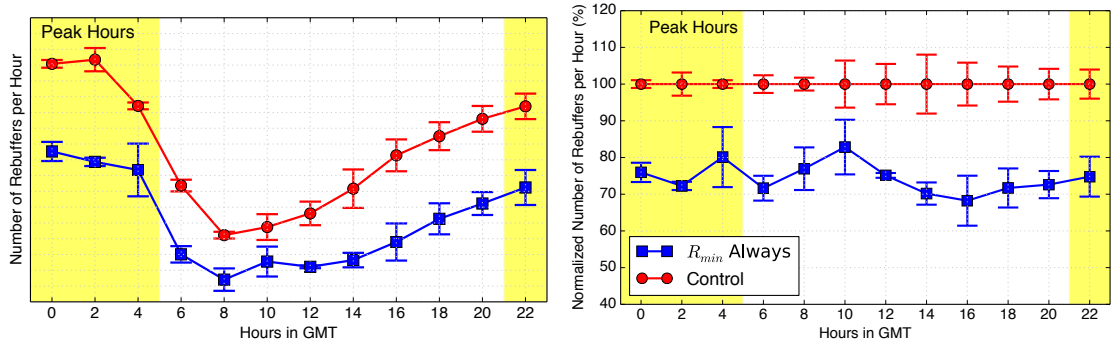
(a) Number of rebuffers per playhour in each two-hour period.

(b) Normalized number of rebuffers per playhour. In each two-hour period, we normalized the number to the average rebuffer rate of *Control*.

Figure 3.2: Number of rebuffers per playhour for the *Control* and $R_{\min}$ *Always*. The error bars represent the variance of rebuffer rates from different days in the same two-hour period.

traffic represented 35% of the US peak Internet traffic and Netflix served 40 million users world-wide. For these reasons, we believe that *Control* is a representative algorithm for our comparison.

Group 2 uses the $R_{\min}$ *Always* algorithm, providing an empirical lower bound on the rebuffer rate against which we can compare to *Control*. For most sessions $R_{\min} = 560kb/s$, but in some cases it is $235kb/s$.[2]

Both user groups are distributed similarly across ISPs, geographic locations, viewing behaviors and devices. The only difference between the two groups of clients is the ABR algorithm; they share the same code base for other mechanisms, such as CDN selection and error handling. We also keep the play delay the same across the groups; this is the delay between when the play button is clicked and when the first video frame is shown on the screen. During our experiments each group of users viewed roughly $120,000$ hours of video. By comparing the rebuffer rate of *Control* and $R_{\min}$ *Always*, we can understand how often unnecessary rebuffers happen in a real commercial system.

Figure 3.2(a) plots the number of rebuffers per play-hour throughout the day. To compare the performance quantitatively, Figure 3.2(b) normalizes the numbers to the average rebuffer rate of the *Control* group in each two-hour period. The first thing to notice is that

---

[2]In Netflix, R_{min} is normally $235kb/s$. However, most customers can sustain 560kb/s, especially in Europe. If a user historically sustained 560kb/s, we artificially set $R_{min} = 560kb/s$ to avoid degrading the video experience too much. The mechanism to pick $R_{\min}$ is the same across all test groups.
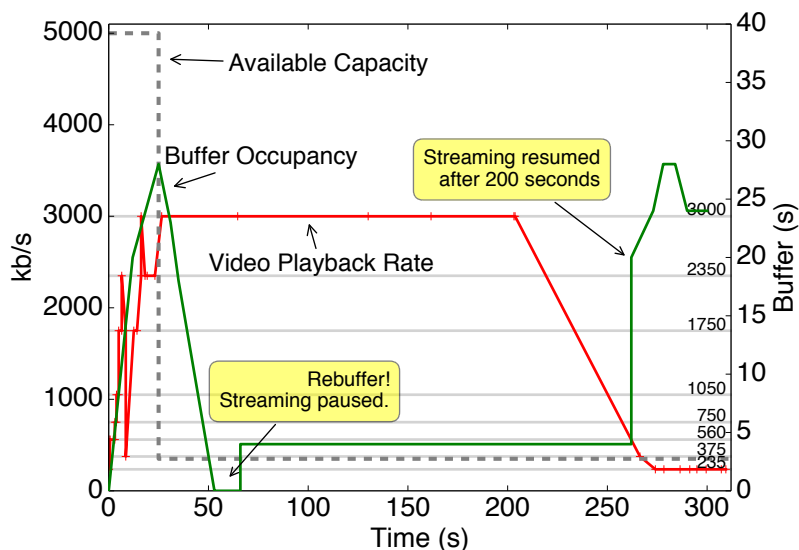
Figure 3.3: An Example of an existing algorithm being too aggressive: A video starts streaming at 3Mb/s over a 5Mb/s network. After 25s the available capacity drops to 350 kb/s. Instead of switching down to a lower video rate, e.g., 235kb/s, the client keeps playing at 3Mb/s. As a result, the client rebuffers and does not resume playing video for 200s. Note that the buffer occupancy was not updated during rebufferings.

$R_{\min}$ *Always* always has a lower rebuffer rate than the *Control* algorithm. The difference between the *Control* algorithm and the $R_{\min}$ *Always* algorithm suggests that 20-30% of the rebuffers might be caused by poor choice of video rate. In the next section, we will focus on an individual example to understand the causes of these unnecessary rebuffers.

3.2   Individual Cases

We reuse the setup described in Chapter 2 to control the bandwidth and analyze the result. Figure 3.3 shows an example from service A to illustrate the problem. The ABR algorithm in the figure overestimates the capacity and continues to request video at a unsustainable rate after the capacity has dropped. The client rebuffers and freezes playback for 200 seconds. Similar behavior is also observed in a different commercial service [9]. Notice that this rebuffer is entirely unnecessary because $C(t) > R_{\min}$ for the entire time series. In fact, if the network capacity is always greater than the lowest video rate $R_{\min}$, i.e. $C(t) > R_{\min}, \forall t > 0$, there *never* needs to be a rebuffer—the algorithm can simply pick $R(t) = R_{\min}$ so that $C(t)/R(t) > 1, \forall t > 0$ and the buffer keeps growing. The main reason
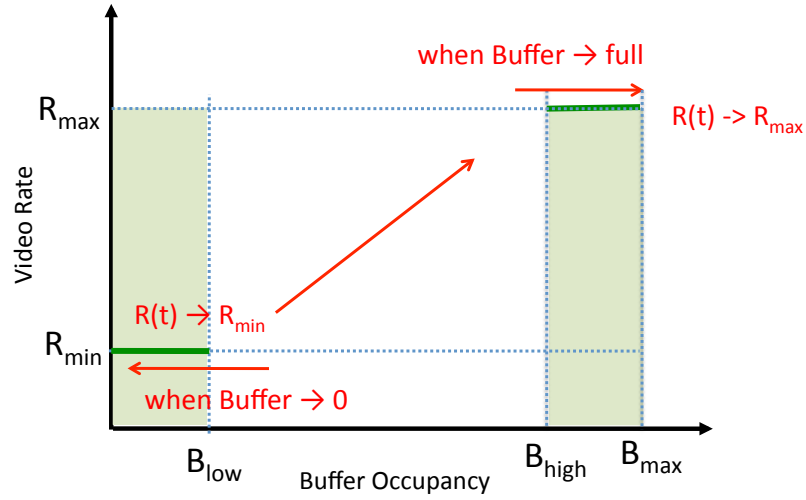
Figure 3.4: Our recommendations based on the two case studies. To avoid unnecessary rebuffers, request $R_{\min}$ when the buffer approaches empty. To avoid downward spiral effect, request $R_{\max}$ when the buffer approaches full. Together they motivate the buffer-based approach to video rate adaptation.

the client does not switch is that *it incorrectly estimates the current capacity to be sufficient to sustain a higher video rate.* As a result, despite the fact that capacity is sufficient to sustain $R_{\min}$, the client does not find its way to that video rate in time.

3.3 Towards the Buffer-Based Design

Our informal discussion suggests a path forward. The easiest way to ensure that an algorithm never unnecessarily rebuffers is to simply request rate $R_{\min}$ when the buffer approaches empty, allowing the buffer to grow as long as $C(t) > R_{\min}$. As the buffer grows, it is safe to increase $R(t)$ up to the maximum video rate as the buffer approaches full.

Together with the recommendation in Chapter 2, we now have the following two recommendations in total:

1. To avoid the downward spiral: an ABR algorithm should pick $R(t) = R_{max}$ when $B(t) \to B_{max}$. (Chapter 2)

2. To avoid unnecessary rebuffers: an ABR algorithm should pick $R(t) = R_{min}$ when $B(t) \to 0$, and increase $R(t)$ as $B(t)$ grows. (Chapter 3)

Figure 3.4 summarizes the two recommendations. Notice that both recommendations pick a video rate $R(t)$ *solely* based on the buffer occupancy $B(t)$. This shows that the occupancy of the playback buffer is the primary state variable we should control. To maximize video quality, we are in fact trying to prevent unnecessary buffer overruns (i.e., streaming below the highest possible quality level). On the other hand, to minimize unnecessary rebuffers, we are in fact trying to prevent unnecessary buffer underruns (i.e., unnecessary rebuffers).

If it is the playback buffer we are controlling, then why not measure and control its occupancy directly? The current buffer occupancy tells us how far we are from an underrun or overrun, and its rate of change captures the mismatch between the system capacity and the requested video rate. The buffer occupancy reflects the end-to-end system capacity, including current load conditions of the network, the CDN, and the video client. In brief, the buffer occupancy contains a considerable amount of information and is the variable we should directly control. This observation motivates our buffer-based design in Chapter 4.

Chapter 4

The Buffer-Based Approach: Theoretical Foundations

Inspired by the observations in Chapter 2 and 3, we propose a class of algorithms that select the video rate *solely* based on the playback buffer level. This class of algorithms does no capacity estimation at all; rather, they rely on buffer dynamics to implicitly capture the available capacity. We call this class of ABR algorithms *buffer-based algorithms (BBAs)*. In this chapter, we first define buffer-based algorithms and then formally analyze their properties. We show that buffer-based algorithms (1) will never unnecessarily rebuffer and (2) achieve an average video rate equal to the available capacity in steady state.

4.1 An HTTP Streaming Model

To set the stage, we extend the observation in the previous chapters and develop a formal model of HTTP-based video streaming.

Video rates and video chunks. In our formal model, we assume the video client can choose from a set of m discrete video rates, $\{R_1, \ldots, R_m\}$, where $R_1 < R_2 < \cdots < R_m$. We also refer to R_1 as $R_{\min}$ (the minimum video rate) and R_m as $R_{\max}$ (the maximum video rate). In addition, we assume that clients download video chunk by chunk. Regardless of the encoded video rate, each chunk contains V seconds of video. A client can only change its selected video rate on a chunk-by-chunk basis, since this is the granularity of requests.

The streaming buffer. In the model we consider, the streaming buffer in the video client is typically measured in seconds of playback time. At any time, the buffer may contain

45

chunks with many different video rates, and the output bitrate of the buffer will depend on the video rate of the chunk currently being played. As a result, there is no direct mapping between buffer occupancy in bytes and buffer occupancy in seconds. By measuring the buffer in the time domain, the client keeps a record of how many *video seconds* of playback video currently reside in the buffer without having to track the video rate associated with each video chunk. We let $B_{\max}$ denote the maximum buffer capacity in seconds.

Buffer dynamics. We index time by $t \geq 0$ and let $C(t)$ denote the system capacity at time t. Here, the system capacity represents the overall end-to-end capability of the system, including the capacity of the CDN servers, the video client, and the available bandwidth of the network in between. We let $B(t)$ be the playback buffer occupancy at time t (measured in seconds). Finally, we let $R(t)$ denote the video rate selected at time t. Note that if the buffer is full, then no chunk can be downloaded at time t; thus we adopt the convention that if $B(t) = B_{\max}$ then $C(t) = 0$.

Observe that the buffer drains at unit rate (since one second is played back every second of real time) and fills at rate $C(t)/R(t)$. Thus the buffer dynamics obey the following simple differential equation:

$$\frac{dB(t)}{dt} = \begin{cases} [C(t)/R(t) - 1]^+, & \text{if } B(t) = 0; \\ C(t)/R(t) - 1, & \text{if } 0 < B(t) < B_{\max}; \\ [C(t)/R(t) - 1]^-, & \text{if } B(t) = B_{\max}. \end{cases} \tag{4.1}$$

(Here the notation $[x]^+$ means the positive part of x, $\max\{x, 0\}$; and $[x]^-$ means the negative part of x, $\min\{x, 0\}$.)

Given the constraint that we can only select video rates on a chunk-by-chunk basis, it is useful to consider the buffer dynamics when they are observed *at instants in time when a chunk finishes*. Formally, let t_k be the completion time of the k-th chunk; by convention, let $t_0 = 0$. Let $r[k]$ be the video rate selected for the k-th chunk, so that $R(t) = r[k]$ for $t_{k-1} < t \leq t_k$. Similarly, let $c[k]$ be the average download capacity for the k-th chunk, so that:

$$c[k] = \frac{\int_{t_{k-1}}^{t_k} C(t)\, dt}{t_k - t_{k-1}}.$$

If each chunk contains V seconds of video, the k-th chunk is $Vr[k]$ bytes long. This assumes a constant bitrate (CBR) stream; we extend our results to a variable bitrate (VBR) stream in Chapter 5. Since the k-th chunk takes $Vr[k]/c[k]$ seconds to download, we have $t_k =$
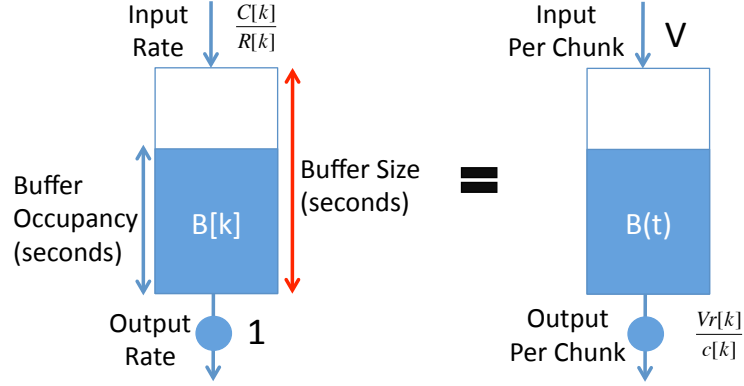
Figure 4.1: Two equivalent models of the streaming playback buffer.

$t_{k-1} + \frac{C(t)}{R(t)} V r[k]/c[k]$. On the other hand, between t_{k-1} and t_k, the buffer fills with V seconds of video. Therefore:

$$B(t_k) = \left[B(t_{k-1}) + V - \frac{V r[k]}{c[k]} \right]^+ . \tag{4.2}$$

Note that since $C(t) = 0$ whenever $B(t) = B_{\max}$, the buffer must be less than or equal to $B_{\max}$ when a chunk completes.

We summarize the two equivalent models of buffer dynamics in Figure 4.1.

Problem statement. We can now formally define the two objectives discussed in the previous chapters.

No unnecessary rebuffers. An *unnecessary* rebuffer occurs when the buffer underruns despite the fact that sufficient capacity was available. Formally, we require the following property: *If $C(t) > R_{\min}$ for all $t \geq 0$, then $B(t) > 0$ for all $t \geq 0$.*

Average video rate optimization. The playback quality (as perceived by the viewer) is measured by averaging the video rate over chunks; thus the long-run average video rate is $\bar{R} = \lim_{K \to \infty} \frac{1}{K} \sum_{k=0}^{K} r[k]$. Our goal is to maximize $\bar{R}$. At the same time, this rate must be less than or equal to the long-run average capacity, $\bar{C} \leq \lim_{T \to \infty} \frac{1}{T} \int_0^T C(t) \, dt$. Note that even though our goal is to maximize capacity utilization, the underlying TCP ensures the algorithm stays a "good citizen" and only gets its fair share of available capacity.

4.2 Rate Maps and Buffer-Based Algorithms

In this section we define the class of buffer-based algorithms. Since we select a video rate based solely on buffer occupancy, the design space for our algorithms can be expressed
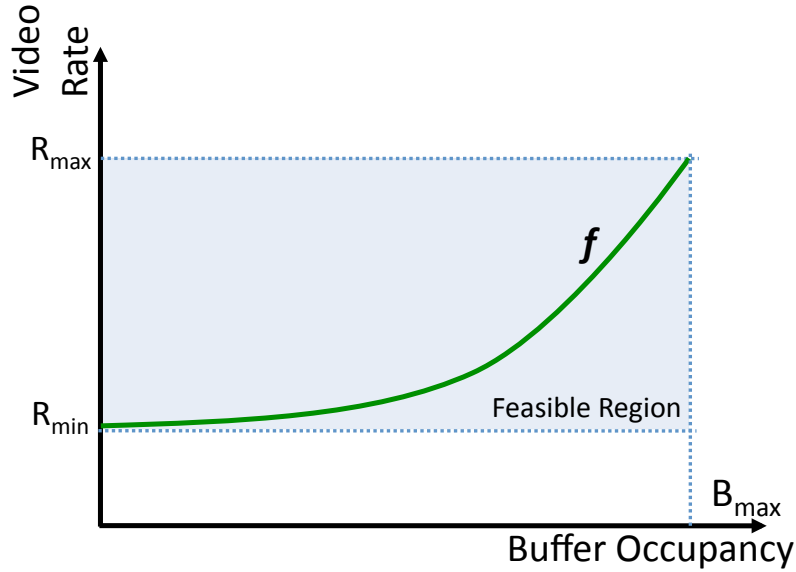
Figure 4.2: The design space of rate maps.

as the buffer-rate plane shown in Figure 4.2. The shaded region between $[0, B_{\max}]$ on the buffer-axis and $[R_{\min}, R_{\max}]$ on the rate-axis defines the feasible region. Any curve $f(B)$ on the plane within the feasible region defines a *rate map*, which maps the current buffer occupancy to a video rate between $R_{\min}$ and $R_{\max}$. We focus on rate maps that are *continuous* functions of the buffer occupancy B and that are *strictly increasing* in the region $\{B : R_{\min} \leq f(B) \leq R_{\max}\}$.

Note that a rate map by itself does not define an algorithm. Since the rate map is continuous, it may not directly correspond to an available discrete video rate. Plus, continuously changing the rate may cause the video rate to oscillate. Instead, we desire an implementation where the video rate is a little "sticky".

We therefore use the rate adaptation algorithm described in Algorithm 1. The algorithm follows a simple principle: it stays at the current video rate as long as the rate suggested by the rate map does not pass either the next-highest available video rate (Rate_+) or the next-lowest available video rate (Rate_-). If either of these "barriers" are hit, the rate is switched up or down (respectively) to a new discrete value suggested by the rate map. In other words, if the rate suggested by the rate map ($f(B)$) is higher than Rate_+, the rate is switched up to the highest rate that is lower than $f(B)$. Similarly, if $f(B)$ is lower than Rate_-, the rate is switched down to the lowest rate that is higher than $f(B)$. In this way,

the buffer distance between the adjacent video rates provides a natural cushion to absorb rate oscillations.

Algorithm 1: Video Rate Adaptation Algorithm

Input: $\text{Rate}_{\text{prev}}$: The previously used video rate

Buf_{now}: The current buffer occupancy

Output: $\text{Rate}_{\text{next}}$: The next video rate

if $\text{Rate}_{\text{prev}} = R_{\max}$ **then**
| $\text{Rate}_+ = R_{\max}$
else
| $\text{Rate}_+ = \min\{R_i : R_i > \text{Rate}_{\text{prev}}\}$

if $\text{Rate}_{\text{prev}} = R_{\min}$ **then**
| $\text{Rate}_- = R_{\min}$
else
| $\text{Rate}_- = \max\{R_i : R_i < \text{Rate}_{\text{prev}}\}$

if $f(\text{Buf}_{\text{now}}) = \text{Rate}_{max}$ **then**
| $\text{Rate}_{\text{next}} = \text{Rate}_{max}$;
else if $f(\text{Buf}_{\text{now}}) = \text{Rate}_{min}$ **then**
| $\text{Rate}_{\text{next}} = \text{Rate}_{min}$;
else if $f(\text{Buf}_{\text{now}}) \geq \text{Rate}_+$ **then**
| $\text{Rate}_{\text{next}} = \max\{R_i : R_i < f(\text{Buf}_{\text{now}})\}$;
else if $f(\text{Buf}_{\text{now}}) \leq \text{Rate}_-$ **then**
| $\text{Rate}_{\text{next}} = \min\{R_i : R_i > f(\text{Buf}_{\text{now}})\}$;
else
| $\text{Rate}_{\text{next}} = \text{Rate}_{\text{prev}}$;

return $\text{Rate}_{\text{next}}$;

4.3   An Idealized Setting

Given the algorithm described in Section 4.2, our goal is to find a class of mapping functions that can achieve our two objectives: (1) to avoid unnecessary rebuffers, and (2) to maximize average video rate. In this section we use an idealized model to gain some intuition before relaxing our assumptions in Section 4.4. To start with, we make the following simplifying assumptions:

1. The chunk size is infinitesimal, so that we can change the video rate continuously.
2. *Any* video rate between $R_{\min}$ and $R_{\max}$ is available.

3. Videos are encoded at a constant bitrate (CBR).

4. Videos are infinitely long.

With these assumptions, the buffer-based algorithm becomes quite simple: at every time t, we *instantaneously* map the buffer level $B(t)$ to the video rate $f(B(t))$. The resulting buffer dynamics are:

$$\frac{dB(t)}{dt} = \begin{cases} [C(t)/f(0) - 1]^+, & \text{if } B(t) = 0; \\ C(t)/f(B(t)) - 1, & \text{if } 0 < B(t) < B_{\max}; \\ [C(t)/f(B_{\max}) - 1]^-, & \text{if } B(t) = B_{\max}. \end{cases} \tag{4.3}$$

In what follows, we consider rate maps f (cf. Section 4.2) that are pinned at both ends: $f(0) = R_{\min}$ and $f(B_{\max}) = R_{\max}$. In other words, the rate map moves from the lowest to highest video rate as the buffer moves from empty to full. As we will see, any such rate map will automatically give us the desired properties in this idealized setting.

No unnecessary rebuffers. Since $f(B) \to R_{\min}$ as $B \to 0$, the derivative of $B(t)$ will become positive before the buffer hits zero. Thus we have the following result.

Theorem 1. [No unnecessary rebuffers] As long as $C(t) \geq R_{\min}$ for all t and we adapt $f(B) \to R_{\min}$ as $B \to 0$, we will never unnecessarily rebuffer because the buffer will start to grow before it runs dry.

Average video rate maximization. Now suppose that $C(t) = C$ for all t, where $R_{\min} < C < R_{\max}$. Informally, we can expect the buffer level to eventually converge to a value B^* where $f(B^*) = C$. In other words, the video rate selected will exactly match the capacity. This property is captured in the following theorem.

Theorem 2. [Average video rate maximization] Suppose that $R_{\min} < C < R_{\max}$. Then starting from any initial buffer level, $\lim_{t \to \infty} B(t) = B^*$, where B^* is the unique solution to $f(B^*) = C$.

Proof.

To prove the theorem, we will show that the system (1) will converge to an equilibrium point and (2) has a unique equilibrium point at B^*. In other words, we will show that the system is *globally asymptotically stable* at the target buffer level B^*.

It is common to establish global asymptotic stability by finding a negative-definite (or a positive-definite) function to represent the state of the system. When used in stability

analysis, such a function is often called a *Lyapunov function*. Intuitively, the derivative of the Lyapunov function represents the *energy* of the system. The existence of a Lyapunov function is a sufficient condition for global asymptotic stability [25]. We can construct a Lyapunov function for our system as follows:

First, since $R_{\min} < C < R_{\max}$, it is straightforward to check that at all $t > 0$, the buffer level will satisfy $0 < B(t) < B_{\max}$.

Now define V(B) as:

$$V(B) = C[\int_0^B (1/f(z))dz] - B.$$

Then since f is positive and strictly increasing, it is straightforward to show that V is strictly concave and has a unique maximum at B^*. If we now calculate $dV(B(t))/dt$ and substitute using (4.3), we obtain:

$$\frac{dV(B(t))}{dt} = \left(\frac{dB(t)}{dt}\right)^2.$$

In particular, this derivative is strictly positive as long as the system is not at the equilibrium point B^*. Thus V is a Lyapunov function for the system and ensures that the target buffer level B^* is globally asymptotically stable.

Note that if $C > R_{\max}$, (4.3) shows that the buffer will fill up and remain full; thus, the video rate will remain at $R_{\max}$, which is the best we can hope to achieve.[1]

Finally, note that more generally, capacity is not necessarily constant. In particular, it may vary over time. Using basic results in stochastic control [13], we can show that as long as the capacity is a Markov process (e.g., a deterministic mean capacity plus a Gaussian noise term), stationary Markov control policies, which select actions only based on the current state, are optimal. Because a buffer-based algorithm selects a rate only based on the buffer level, it is a stationary Markov control policy, and it is *optimal* for video rate maximization.

The preceding results shows that in an idealized setting, buffer-based algorithms can achieve the two goals simultaneously when using a rate map that satisfies the following two conditions: (1) $f(B) \to R_{\min}$ as $B \to 0$, and (2) $f(B)$ is increasing and eventually reaches R_{max}. In the next two sections, we see how our algorithms behave in a more realistic context and show that essentially the same insights continue to hold.

---

[1]In practice, we would observe the "ON-OFF" behavior in the video stream due to finite chunk sizes, as we observed in Chapter 2.

4.4 A More Realistic Setting

In this section, we relax the first two assumptions in the idealized setting: infinitesimal chunk size and continuous video rate. These relaxations allow us to develop buffer-based algorithms to test in the field and get insights on further relaxing the other two assumptions in Chapter 5.

In practice, the chunk size is finite (V seconds long), and a chunk is only added to the buffer after it is downloaded. To avoid rebuffers, we always need to have at least one chunk available in the buffer. Thus, to handle the finite chunk size, as well as some degree of variation in the system, we shift the rate map to the right and create an extra *reservoir*, noted as r. When the buffer is filling up the reservoir, i.e., $0 \leq B \leq r$, we request video rate $R_{\min}$. Once the reservoir is reached, we then increase the video rate according to $f(B)$. Also because of the finite chunk size, the buffer does not stay at $B_{\max}$ even when $C(t) \geq R_{\max}$; thus, we should allow the rate map to reach $R_{\max}$ before $B_{\max}$. We call the buffer between the reservoir and the point where $f(B)$ first reaches $R_{\max}$ the *cushion*, and the buffer after the cushion the *upper reservoir*.

Since many video clients have no control over TCP sockets and cannot cancel an ongoing video chunk download, we can only pick a new rate when a chunk *finishes* arriving. If the network suddenly slows down while we are in the middle of downloading a chunk, the buffer might run dry before we get the chance to switch to a lower rate. Further, we are aiming to maintain the buffer level above the reservoir r. Thus, to prevent unnecessary rebuffers, $f(B)$ should be designed to ensure that a chunk can always be downloaded before the buffer shrinks into the reservoir area. Based on these observations, we say $f(B)$ operates in the *safe* area if it always picks chunks that will finish downloading before the buffer runs below r, when $C(t) \geq R_{\min}$ for all t. In other words, $Vf(B)/R_{\min} \leq (B - r)$. Otherwise, $f(B)$ is in the *risky* area.

Thus, the class of rate maps that we consider become the piecewise functions described in Figure 4.3. We illustrate there the reservoir and the cushion. We also illustrate the notion of safety described in the previous paragraph: we plot the boundary of the safe area as the red dashed line in the figure. Any $f(B)$ below the boundary will be a safe choice.

Now, we also assume the system only provides m discrete video rates $\{R_1, \ldots, R_m\}$ instead of any video rate between R_{min} and R_{max}. With the discrete set of video rates, the buffer level $B(t)$ no longer directly maps to a video rate, since $f(B(t))$ might not be
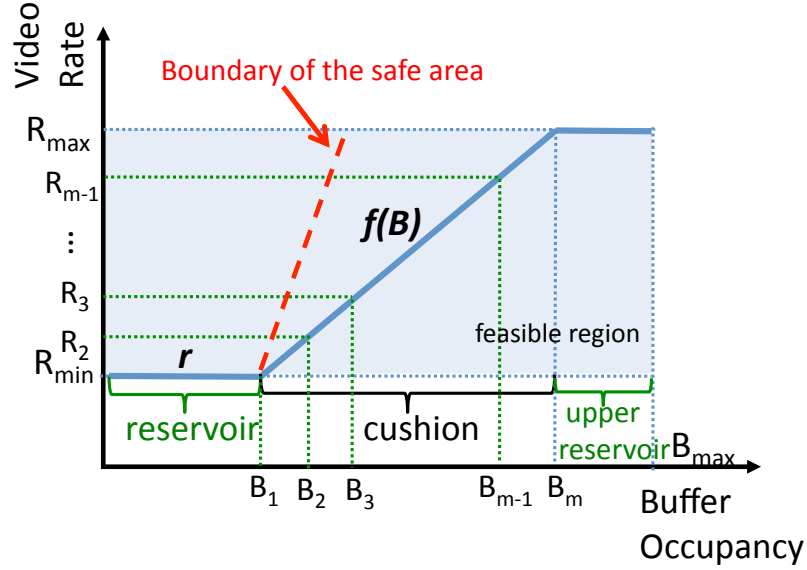
Figure 4.3: A rate map as a piecewise function. After taking finite chunk size into consideration, the rate map becomes a piecewise function.

an available rate. Thus Algorithm 1 (cf. Section 4.2) has slightly more complex dynamics, making the analysis more challenging.

Nevertheless, in the following we show that the revised rate maps can still achieve our two objectives after removing the first two assumptions.

No unnecessary rebuffers. Let's assume $C(t) \geq R_{\min}$ for all t and we use a *safe* rate map $f(B)$, i.e., $Vf(B)/R_{\min} \leq (B - r)$ when $B > r$. When the buffer is above the reservoir, the map selects a chunk that can be downloaded within $(B - r)$ seconds, i.e., before the buffer runs below r. Thus, we will not unnecessarily rebuffer, as long as the size of the reservoir is large enough to accommodate the buffer variation caused by the finite chunk size. In other words, the reservoir needs to be big enough to prevent the buffer from going empty while a chunk is still downloading. Since $f(B) = R_{min}$ for all $B \leq r$ and $C(t) \geq R_{\min}$ for all t, the reservoir size needs to be at least $Vf(B)/C(t) = V$ seconds (i.e., one chunk) to prevent rebuffers. Thus, we can generalize Theorem 1 and construct Theorem 3 as follows.

Theorem 3. [No unnecessary rebuffers] As long as $C(t) \geq R_{\min}$ for all t, the rate map satisfies $Vf(B)/R_{\min} \leq (B - r)$, and $f(B) \rightarrow R_{\min}$ as $B \rightarrow r$ where $r > V$, we will never unnecessarily rebuffer because the buffer will start to grow before it runs dry.

Average video rate maximization. When available video rates are not continuous but discrete, under Algorithm 1 the buffer level might not converge to B^*. This lack of convergence occurs because $f(B^*)$ might not map to one of the available video rates. Instead, the buffer will swing between two occupancies B_i and B_j, where $f(B_i)$ and $f(B_j)$ map to discrete rates above and below C (R_i and R_j, respectively, i.e., $R_i < C < R_j$).

Suppose we again use a safe rate map such that $f(B) = R_{\min}$ for all B such that $0 \le B \le r$, and $f(B_{\max}) = R_{\max}$. For such rate maps, informally, the rate will hover around the capacity C in steady state. In the following, we prove that the long-run average video rate, $\bar{R}$, equals the capacity C.

Theorem 4. [Average video rate maximization] Given a constant capacity ($C = C(t)$ for all t, and $R_{\min} < C < R_{\max}$), the average video rate $\bar{R}$ will be equal to C with any predefined discrete set of video rates $\{R_1, \ldots, R_m\}$.

Proof. Assuming the buffer level does not converge to B^*, where $f(B^*) = C$, and the buffer level swings between B_i and B_j, where $f(B_i)$ and $f(B_j)$ map to discrete rates above and below C (R_i and R_j respectively, i.e., $R_i < C < R_j$). We denote the difference between B_i and B_j as ΔB_{ij}.

Since $R_i < C$, when $R(t) = R_i$ the buffer is increased by the rate of $\frac{C}{R_i} - 1$. We will switch up to R_j once $B(t) > B_j$, thus the amount of time we stay at R_i can be calculated as follows:

$$
\begin{aligned}
T_i &= \frac{\Delta B_{ij}}{(\frac{C}{R_i} - 1)} \\
&= \frac{\Delta B_{ij} \times R_i}{C - R_i}
\end{aligned}
\tag{4.4}
$$

During T_i seconds, we download a total of $T_i \frac{C}{R_i}$ seconds of video with rate R_i.

Once we switch to R_j, since $R_j > C$, the buffer will start decreasing at the rate of $1 - \frac{C}{R_j}$. We will switch down to R_i once $B(t) < B_i$, thus the amount of time we stay at R_j can be similarly calculated as follows:

$$
\begin{aligned}
T_j &= \frac{\Delta B}{(1 - \frac{C}{R_j})} \\
&= \frac{\Delta B \times R_j}{R_j - C}
\end{aligned}
\tag{4.5}
$$

Following the same rationale, during T_j seconds, we download a total of $T_j \frac{C}{R_j}$ seconds of video with rate R_j.

The average video rate during $T_i + T_j$ would be the number of downloaded bits divided by the length of the video we download. In other words, $\bar{R} = \frac{C(T_i+T_j)}{T_i \frac{C}{R_i} + T_j \frac{C}{R_j}}$. With some replacement of variables, we will be able to derive that the average video rate $\bar{R}$ equals the constant capacity C.

So far we have shown that by adapting the rate map, the buffer-based design is still able to achieve the two goals while handling finite chunk size and discrete video rates. These relaxations allow us to put together a baseline algorithm and test the concept of buffer-based approach in the field. In the next chapter, we will first test the baseline algorithm through a real-world deployment. The observations made from the experiments then lead to techniques for relaxing the last two assumptions: CBR encoding and infinite video length.

Chapter 5

The Buffer-Based Approach: Netflix Deployment

In Chapter 4, we proposed a class of buffer-based algorithms (BBAs) and formally proved that in an idealized setting, BBAs will (1) never unnecessarily rebuffer and (2) always maximize the video rate. The idealized setting includes the following four assumptions: (1) infinitesimal chunk sizes, (2) continuous available video rates, (3) constant bitrate (CBR) encoding, and (4) infinite video length. To help apply the concept of BBAs to the real world, we relaxed the first two simplifying assumptions, expanding the proof to consider finite chunk sizes and discrete available video rates. We formally showed that BBAs can still achieve the original two goals by using a piecewise mapping function. This relaxation allows us to put together a baseline algorithm to test in the field, and the experimental results help us to further develop techniques to relax the remaining two assumptions: CBR encoding and infinite video length.

In this chapter, we conduct a series of experiments to help us understand how the buffer-based approach interacts with real-world settings, such as VBR encoding and finite video length. We first construct a naive baseline algorithm, BBA-0, which verifies the design of handling finite chunk size and discrete available video rates. BBA-0 also attempts to handle the VBR encoding with a large and fixed-size reservoir and cushion. We deploy the algorithm to Netflix's browser-based player and test it with real Netflix users. Through the real-world deployments, the result shows that although BBA-0 helps reduce rebuffers for some sessions, it is not sufficient for others. This result suggests that the size of reservoir

needs to be dynamically adjusted, because each VBR-encoded video varies around the average video rate differently and requires different amount of buffer to absorb the variation. Furthermore, when picking a video rate, an ABR algorithm also needs to take the instantaneous video rate into consideration. Since the variation on instantaneous video rate reflects on chunk sizes, our next algorithm, BBA-1, dynamically calculates the reservoir size from the time series of chunk sizes and considers chunk sizes when picking a video rate. The result shows that BBA-1 is able to handle the VBR encoding and avoids unnecessary rebuffers. However, BBA-1 has a lower average video rate compared to the *Control*. A closer look reveals that when there is abundant capacity, the *Control* takes much less *video time* to ramp up the video rate at the beginning of each new session. In contrast, the BBA keeps requesting R_{min} until the reservoir portion of the buffer is filled up, regardless the available capacity. In a production streaming service, this ramp-up period is a non-negligible fraction of the average session length. During this period, because the buffer itself is still growing from empty and contains very little information, a simple capacity estimation becomes necessary to ramp up the video rate faster. The further improved algorithm, BBA-2, therefore divides each session into two phases: the startup phase and the steady-state phase. During the startup phase, a simple capacity estimation based on the immediate past throughput is used to help ramp up the video rate. During the steady state, BBA-2 keeps the buffer-based approach and picks the video rate directly from the buffer occupancy without estimating the system capacity. The result shows that BBA-2 is able to reduce the rebuffer rate by 10–20% compared to Netflix's then-default algorithm, without compromising the video rate; the average video rate stays similar and the video rate is higher in the steady state. These experiments not only demonstrate the viability of the buffer-based approach, but also confirm that the buffer-based approach can both reduce the rebuffer rate and improve the video rate in a real-world system. Finally, we also develop techniques to protect against temporary network outage and to smooth video switching rate. By only allowing the reservoir to expand but never shrink, our last algorithm, BBA-Others, grows buffer occupancy to protect against temporary network outage and reduces the switching rate. Table 5.1 summarizes the design goals of these series of algorithms. In the following, we will dive into the designs of each algorithm and discuss their experimental results.

Algorithm	Design Goal
BBA-0	(1) Handling finite chunk size.
	(2) Handling discrete available video rates.
BBA-1	Handling VBR encoding.
BBA-2	Handling finite video length (ramping up faster).
BBA-Others	(1) Handling temporary network outage.
	(2) Smoothing video switching rate.

Table 5.1: Summary of the design goals of BBA-0, BBA-1, BBA-2 and BBA-Others.

5.1 The Baseline Algorithm

To test the buffer-based approach developed in Chapter 4, we first construct a baseline algorithm with a relatively simple and naive rate map. We start the design with a piecewise function as shown in Figure 4.3. We then determine the size of reservoir, cushion, and upper reservoir, as well as the shape of the rate map. We implement the algorithm in Netflix's browser-based player, which happens to have a 240 second playback buffer and the convenient property that it downloads the ABR algorithm at the start of the video session.

As discussed in Chapter 4, the size of reservoir needs to be at least one chunk (4 seconds in this testing environment) to absorb the buffer variation caused by the finite chunk size. However, since the algorithm is tested in a production environment that streams VBR-encoded video, the size of reservoir also needs to be big enough to absorb the buffer variation caused by the VBR encoding. As the first baseline algorithm, we set the size of reservoir to be a large and fixed-size value, 90 seconds. We thought a 90s reservoir is big enough to absorb the variation from VBR, allowing us to focus on testing the approach developed in Chapter 4.

The size of cushion is defined as the buffer distance between B_1 and B_m, as shown in Figure 4.3. As discussed in the proof of Theorem 4, the buffer distance between neighboring rates affects the frequency of rate switches. To maximize the size of cushion while leaving some room for the upper reservoir, we let the rate map reaches $R_{\max}$ when the buffer is 90% full (216 seconds). In other words, we set the cushion to be 126 seconds (between 90 to 216 seconds) and the upper reservoir to be 24 seconds (between 216 to 240 seconds). To further maximize the distance between each pair of neighboring rates, we use a linear function to increase the rate between $R_{\min}$ and $R_{\max}$.
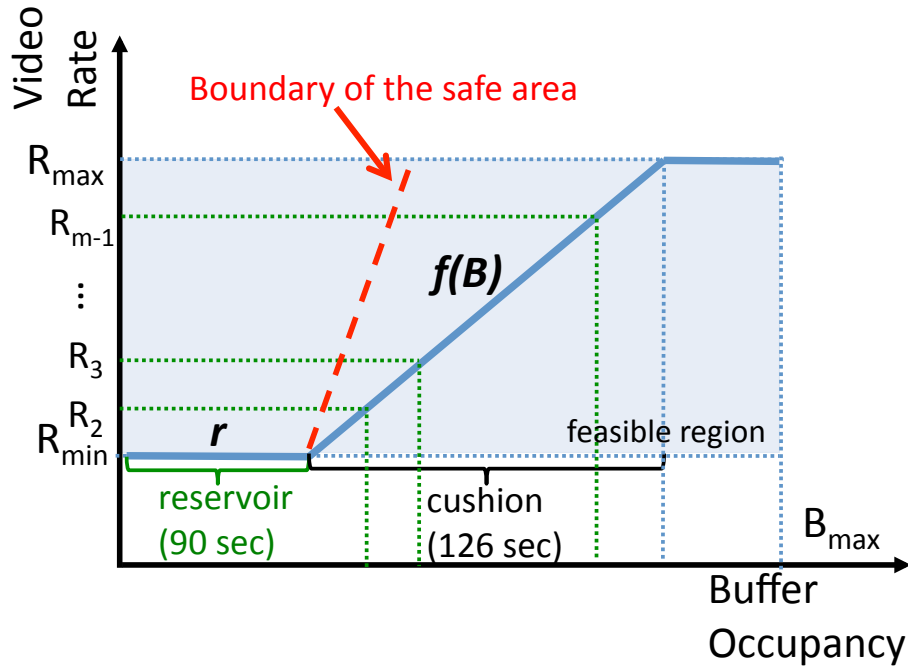
Figure 5.1: The rate map used in the BBA-0 algorithm.

The resulting $f(B)$ is a piecewise linear function, which stays in the safe area defined in Section 4.4. This rate-mapping function, together with Algorithm 1, constructs our first buffer-based algorithm. We call this algorithm *BBA-0* since it is the simplest of our buffer-based algorithms.

We evaluate the algorithm using the same A/B test setup as in Section 3.1. We conducted this experiment along with the experiment in Section 3.1 between September 6th (Friday) and 9th (Monday), 2013. We selected the same number of users as in each group to use our BBA-0 algorithm. To compare their performances, we measure the overall number of rebuffers per playhour and the average delivered video rate in each group.

5.1.1   Experimental Results

Rebuffer Rate. Figure 5.2(a) plots the number of rebuffers per playhour throughout the day. Figure 5.2(b) simplifies a visual comparison between algorithms by normalizing the average rebuffer rate to the *Control* group in each two-hour period. Peak viewing hours for the USA are highlighted in yellow. Error bars represent the variance of rebuffer rates from different days in the same two-hour period. The $R_{\min}$ *Always* algorithm provides

an empirical lower bound on the rebuffer rate. Note that because the users in the three groups are different and their environments are not exactly the same, $R_{\min}$ *Always* only approximates the lower bound for the other groups.

During the middle-of-night period in the USA just after peak viewing (6am–12pm GMT), BBA-0 matches the $R_{\min}$ *Always* lower bound very closely. At 10am GMT, even though BBA-0 has a lower average rebuffer rate than $R_{\min}$ *Always*, the difference is not statistically significant.[1] These two algorithms perform equally during this off-peak period, because the viewing rate is relatively low, overall Internet usage is low, and the network capacity for individual sessions does not change much. The rebuffer rate during these hours is dominated by random local events, such as WiFi interference, instead of congested networks.

During peak hours, the performance with BBA-0 is significantly worse than with the $R_{\min}$ *Always* algorithm. Nevertheless, the BBA-0 algorithm consistently has a 10–30% lower rebuffer rate than the *Control* algorithm. This performance difference is encouraging given the extremely simple nature of the BBA-0 algorithm. Still, we hope to do better. In Section 5.2 and 5.3, we will develop techniques to improve the rebuffer rate of buffer-based algorithms.

Video Rate. Figure 5.3(a) shows the average video rate throughout the day. Figure 5.3(b) shows the difference in the delivered video rate between *Control* and BBA-0. The daily average bitrate for the *Control* algorithm for each ISP can be found in the Netflix ISP Speed Index [30]. Since $R_{\min}$ *Always* always streams at $R_{\min}$ (except when rebuffering), its delivered video rate is a flat line and is excluded from the figure. The BBA-0 algorithm is roughly 100kb/s worse than the *Control* algorithm during peak hours, and 175kb/s worse during off-peak hours. There are two main reasons for the degradation in video quality. First, our BBA-0 algorithm uses a large and fixed-size reservoir to handle VBR, while the size of reservoir should be adjusted to be just big enough to absorb the variation introduced by VBR. Second, and more significantly, while the reservoir is filling up during the startup period, our BBA-0 algorithm always requests video at rate $R_{\min}$. Given that we picked a 90s reservoir, it downloads 90 seconds worth of video at rate $R_{\min}$, which is a non-negligible fraction of the average session length. We will address both issues in Section 5.2 and 5.3.

---

[1] The hypothesis of BBA-0 and $R_{\min}$ *Always* share the same distribution is not rejected at the 95% confidence level (p-value = 0.25).
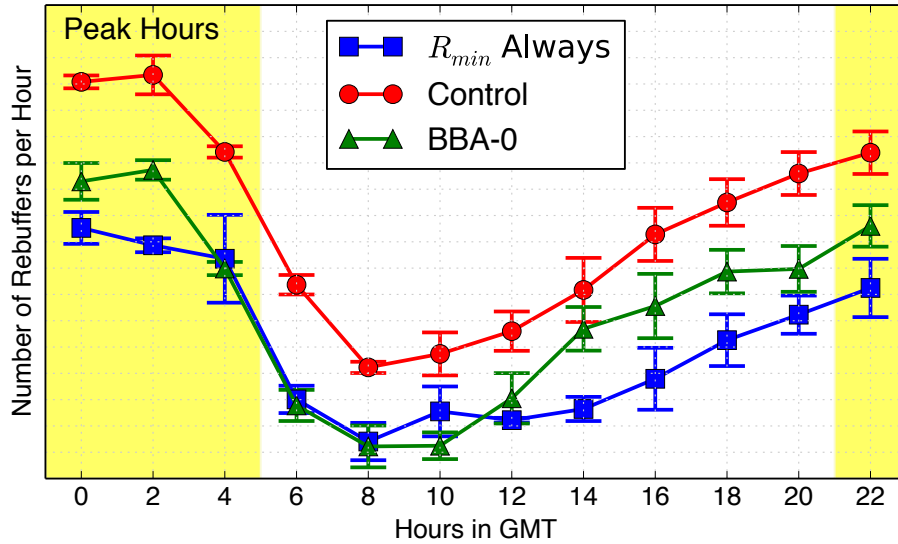
(a) Number of rebuffers per playhour during the day.
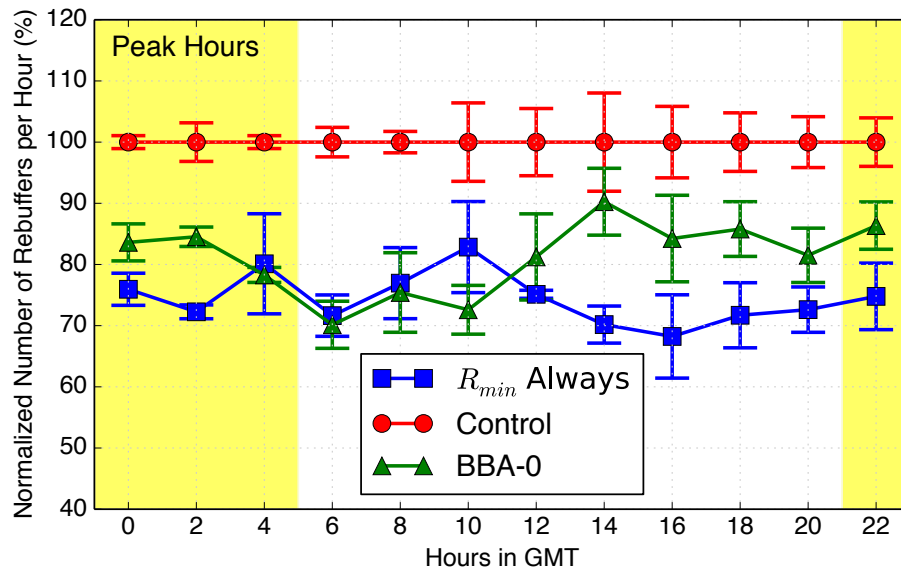
(b) Normalized number of rebuffers per playhour, normalized to the average rebuffer rate of *Control* in each two hour period.

Figure 5.2: Number of rebuffers per playhour for the *Control*, R_{min} *Always*, and BBA-0 algorithms. The error bars represent the variance of rebuffer rates from different days in the same two-hour period.
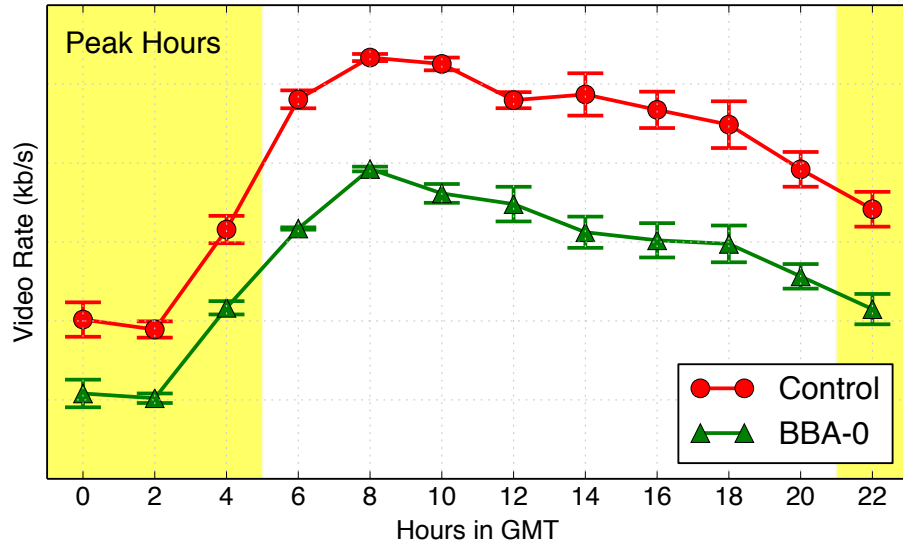
(a) Average video rate during the day.
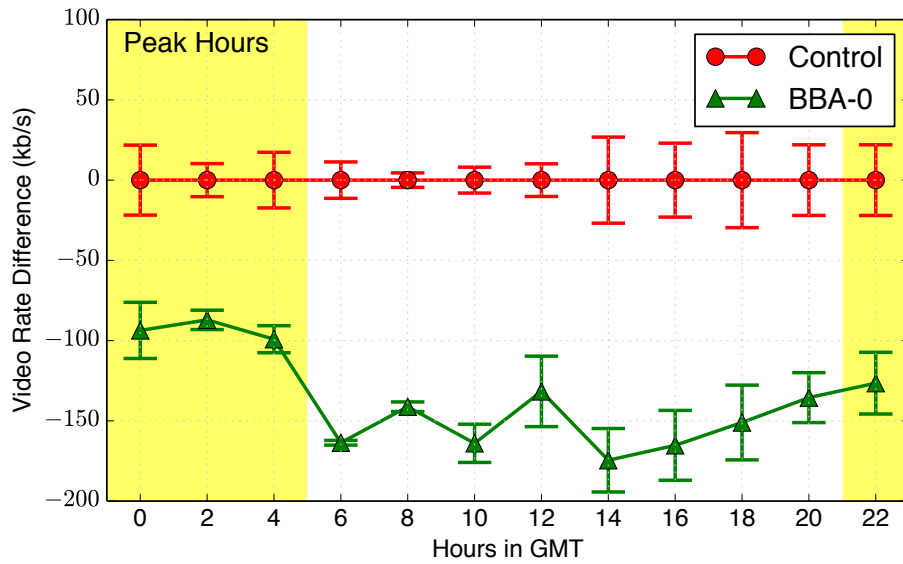
(b) The difference on video rate per two-hour window between *Control* and BBA-0. The Y-axis shows the difference in the delivered video rate between *Control* and BBA-0.

Figure 5.3: Comparison of video rate between *Control* and BBA-0. The error bars represent the variance of video rates from different days in the same two-hour period.
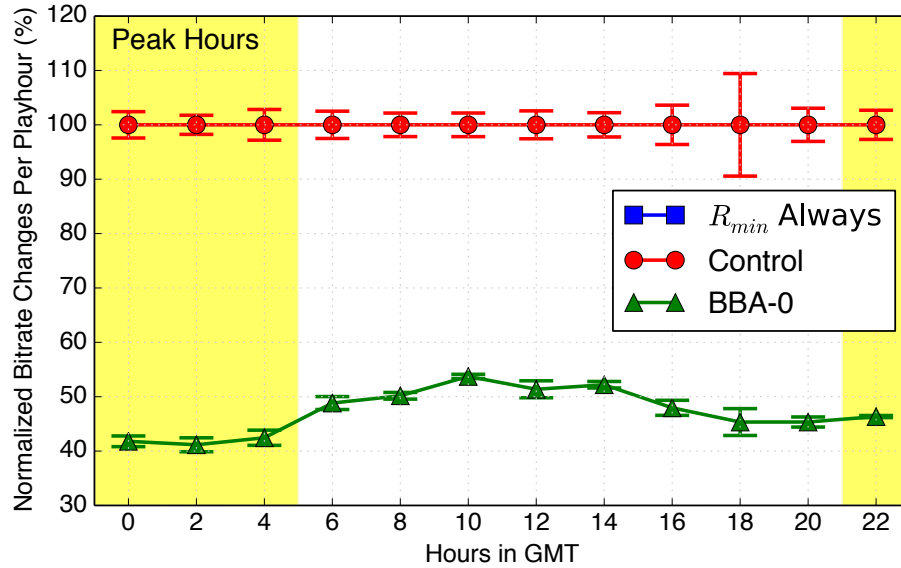
Figure 5.4: Average video switching rate per two-hour window for the *Control* and BBA-0 algorithms. The numbers are normalized to the average switching rate of the *Control* group for each window. The error bars represent the variance of video switching rates from different days in the same two-hour period.

Video Switching Rate. Since our BBA-0 algorithm picks the video rate based on the buffer level, we can expect the rate to fluctuate as the buffer occupancy changes. However, Algorithm 1 uses the distance between adjacent video rates to naturally cushion and absorb rate oscillations. Figure 5.4 compares BBA-0 with the *Control* algorithm. Note the numbers are normalized to the average switching rate of the *Control* group for each two-hour period. The BBA-0 algorithm reduces the switching rate by roughly 60% during peak hours and by roughly 50% during off-peak hours.

In summary, BBA-0 confirms that we can reduce the rebuffer rate by focusing on buffer occupancy. The results also show that the buffer-based approach is able to reduce the video switching rate. However, BBA-0 performs worse on video rate compared to the *Control* algorithm. In the next section, we will develop techniques to improve both rebuffer rate and video rate by considering the VBR encoding scheme.
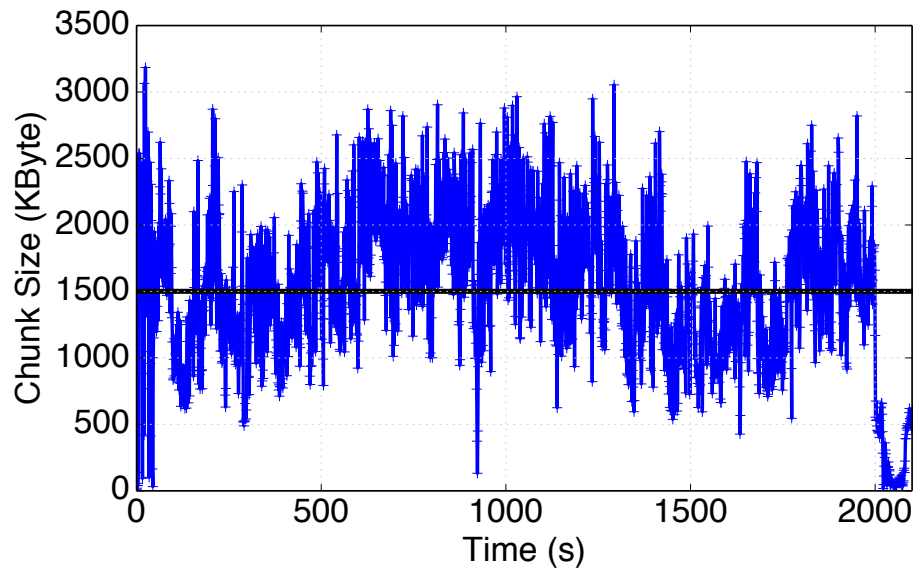
Figure 5.5: The size of 4-second chunks of a video encoded at an average rate of 3Mb/s. Note the average chunk size is 1.5MB (4s times 3Mb/s).

5.2 Handling Variable Bitrate (VBR)

In Section 5.1, the BBA-0 algorithm attempts to handle VBR by setting the reservoir size to a large and somewhat arbitrary value. Although we are able to get a significant reduction in rebuffering compared to the *Control*, there is still room to improve when comparing to the empirical lower bound. In addition, the average video rate achieved by the BBA-0 algorithm is significantly lower than the *Control* algorithm. In this section, we will discuss techniques to improve both rebuffer rate and video rate by taking the encoding scheme into consideration. A key advance is to *design* the reservoir based on the instantaneous encoding bitrate of the stream being delivered.

In practice, most of the video streaming services encode their videos in variable bitrate (VBR). VBR encodes static scenes with fewer bits and active scenes with more bits, while maintaining a consistent quality throughout the video. VBR encodings allow more flexibility and can use bits more efficiently. When a video is encoded in VBR at a nominal video rate, the nominal rate represents the *average* video rate, and the instantaneous video rate varies around the average value. As a result, the chunk size will not be uniformly identical in a stream of a given rate. Figure 5.5 shows the size of 4-second chunks over time from a
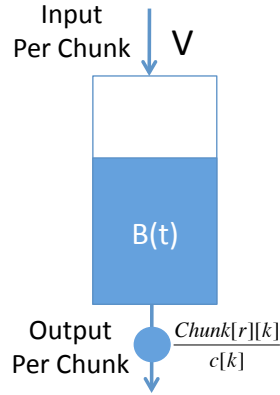
Figure 5.6: The revised buffer model for handling VBR. Since each chunk has a different size in bytes, the revised model takes individual chunk size into consideration, instead of the nominal video rate.

production video (*Black Hawk Down*) encoded at 3 Mb/s. The black line represents the average chunk size. As we can see from the figure, the variation on chunk size can be significant within a single video rate.

Given the variation on chunk size, we need to take the size of each chunk into consideration and extend the buffer model defined in Section 4.1. Let $r[k]$ be the video rate selected for the k-th chunk and $c[k]$ be the average system capacity during the download of the k-th chunk. For the k-th chunk from the stream of nominal video rate r, we denote the chunk size as $Chunk[r][k]$. Since each chunk still contains V seconds of video, the buffer now drains $Chunk[r][k]/c[k]$ seconds while it fills with V seconds of video. The revised model is shown in Figure 5.6.

5.2.1   Reservoir Calculation

Since the instantaneous video rate can be much higher than the nominal rate in VBR, we could still encounter a rebuffer event even when the capacity $c[k]$ is exactly equal to $R_{\min}$. However, we can avoid these rebuffer events if we reserve enough buffer to absorb the buffer oscillation caused by the variable chunk size.

Assuming $c[k] = R_{\min}$, when the chunk size is larger than the average, $VR_{\min}$, the video client will consume more video in the buffer than the input. On the other hand, when the chunk size is lower than the average, the buffer is consumed more slowly than the video is
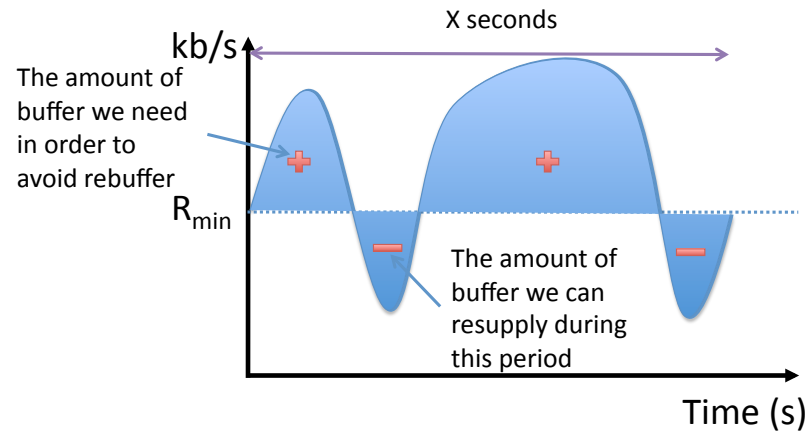
Figure 5.7: Reservoir calculation: We calculate the size of the reservoir from the chunk size variation.

inputted into the buffer and the buffer occupancy will increase. Thus, by taking the amount of buffer the client will consume and subtracting the amount of the buffer the client can resupply during the next X seconds, we can determine the amount of reservoir needed. We dynamically adjust the reservoir based on this prospective calculation over the lifetime of the stream. X should be set to at least as large as the size of the playback buffer, since users expect the service to continue for that period, even when bandwidth drops. Figure 5.7 visually summarizes this calculation. In the implementation of the updated algorithm, we set X to be twice the buffer size, to 480 seconds. The calculated reservoir size depends highly on the specific video and the playing segment. For example, when playing static scenes such as opening credits, since they are encoded with very few bits, the calculated reservoir size is negative; when playing active scenes that are encoded with much more bits, the calculated reservoir size can be even larger than half the buffer size (120 seconds).

As we discussed in Section 4.4, we always need to have at least one chunk available in the buffer to handle the finite chunk size. On the other hand, the reservoir cannot be too big, otherwise we do not have enough buffer space left for the cushion area. As a result, we bound the reservoir size to be between 8 and 140 seconds.[2]
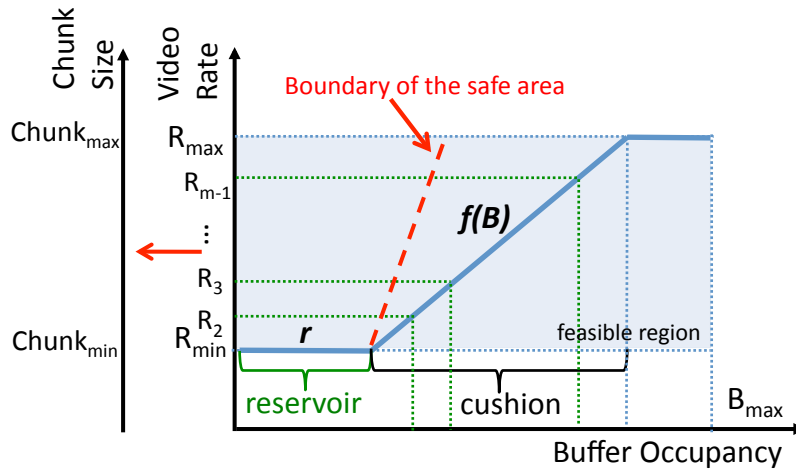
Figure 5.8: Handling VBR with chunk maps. To consider variable chunk size, we generalize the concept of rate maps to chunk maps by transforming the Y-axis from video rates to chunk sizes.

5.2.2   Chunk Map

Since the buffer dynamics now depend on the chunk size of the upcoming video segments instead of the video rate, it makes more sense to map the buffer occupancy to the chunk size directly. In other words, we can generalize the design space and change it from the buffer-rate plane to the buffer-chunk plane as shown in Figure 5.8. Each curve in the figure now defines a *chunk map*, which represents the maximally allowable chunk size according to the buffer occupancy. In the figure, the feasible region is now defined between $[0, B_{\max}]$ on the buffer-axis and $[Chunk_{\min}, Chunk_{\max}]$ on the chunk-axis, where $Chunk_{\min}$ and $Chunk_{\max}$ represent the average chunk size in $R_{\min}$ and $R_{\max}$, respectively.

We can now generalize Algorithm 1 to use the chunk map: the algorithm stays at the current video rate as long as the chunk size suggested by the map does not pass the size of the next upcoming chunk at the next highest available video rate ($Rate_+$) or the next lowest available video rate ($Rate_-$). If either of these "barriers" are passed, the rate is switched up or down, respectively. Note that by using the chunk map, we no longer have a fixed mapping between buffer levels and video rates. This could result in a higher frequency of video rate switches. We will explore techniques to address this issue in Section 5.4.

---

[2]We can also bound the reservoir size at the encoding stage by limiting the variation introduced by the encoder.
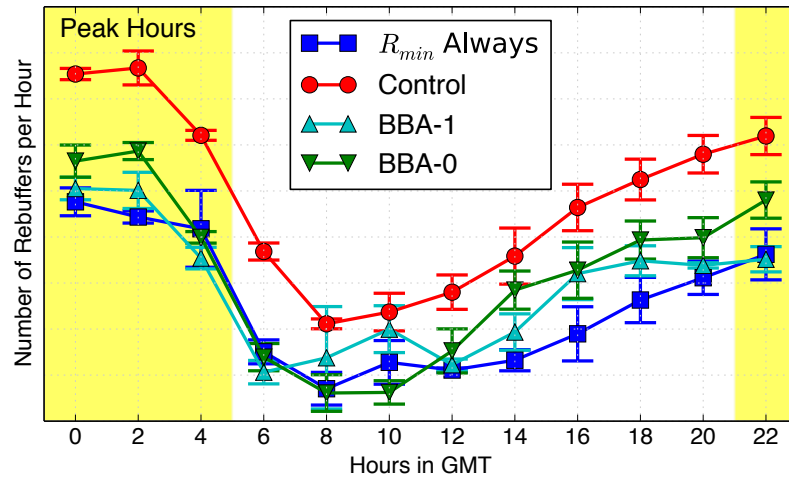
(a) Number of rebuffers per playhour throughout the day.
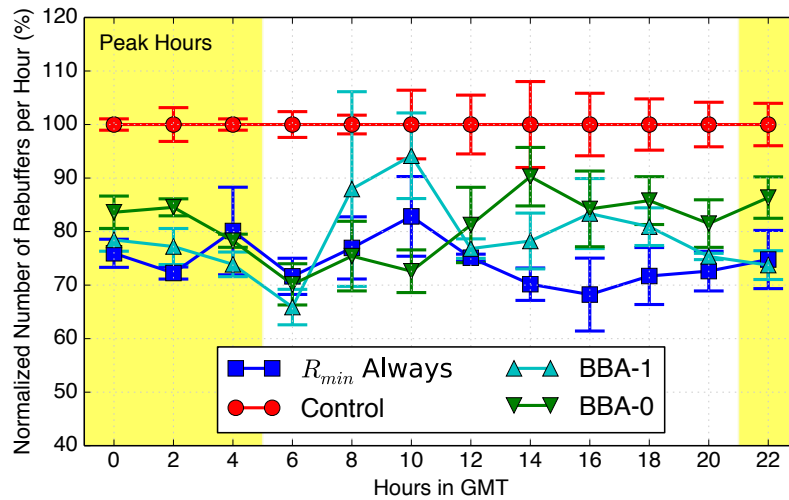
(b) Normalized number of rebuffers per playhour. Each percentage is normalized to the average rebuffer rate of the *Control* algorithm in a two-hour period.

Figure 5.9: Number of rebuffers per playhour for the *Control*, $R_{\min}$ *Always*, BBA-0, and BBA-1 algorithms. The error bars represent the variance of rebuffer rates from different days in the same two-hour period. The BBA-1 algorithm achieves close-to-optimal rebuffer rate, especially during the peak hours.
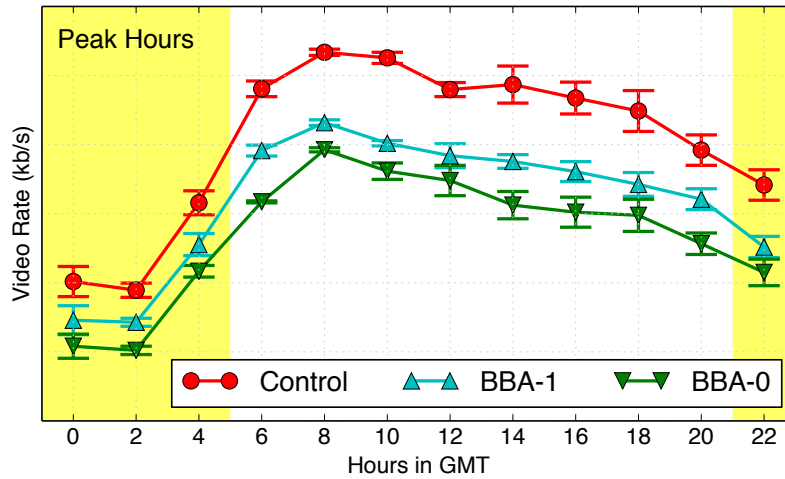
(a) Average video rate during the day.
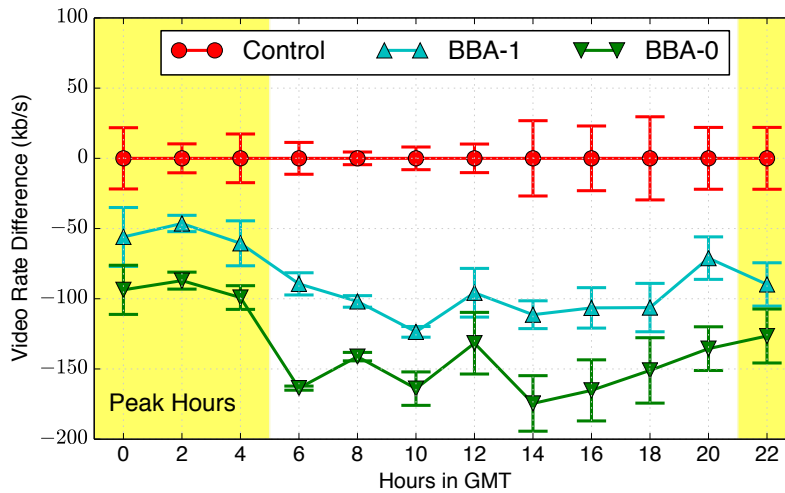
(b) The difference on video rate per two-hour window between the *Control*, BBA-0, and BBA-1 algorithms. The Y-axis shows the difference in the delivered video rate between *Control* and BBAs.

Figure 5.10: Comparison of video rate between the *Control*, BBA-0, and BBA-1 algorithms. The error bars represent the variance of video rates from different days in the same two-hour period. The BBA-1 algorithm improved video rate by 40-70kb/s compared to BBA-0 but still remains 50-120kb/s away from the *Control*.

5.2.3 Experimental Results

We use the same setup as in Section 5.1. We select the same number of users in each group to use our VBR-enabled buffer-based algorithm, which dynamically calculates the reservoir size and uses a chunk map. We will refer to the algorithm as *BBA-1* in the following, as it is our second iteration of the buffer-based algorithm. This experiment was conducted along with the experiment in Section 5.1 between September 6th (Friday) and 9th (Monday), 2013.

Figure 5.9(a) shows the rebuffer rate in terms of number of rebuffers per playhour, while Figure 5.9(b) normalizes to the average rebuffer rate of the *Control* in each two-hour period. We can see from the figure that the BBA-1 algorithm comes close to the optimal line and performs better than the BBA-0 algorithm. BBA-1 has a lower average rebuffer rate than R_{min} *Always* during 4–6am GMT, but the difference is not statistically significant.[3] The improvement over the *Control* algorithm is especially clear during peak hours, where the BBA-1 algorithm provides a 20–28% improvement in the rebuffer rate.

Figure 5.10(a) shows the average video rate throughout the day, while Figure 5.10(b) shows the difference in the average video rate between the *Control*, BBA-0, and BBA-1 algorithms. As shown in Figure 5.10(b), the BBA-1 algorithm also improves the video rate compared to BBA-0 by 40–70kb/s on average, although it is still 50-120kb/s away from the *Control* algorithm. This discrepancy in video rate comes from the startup period, when the buffer is still filling up. If we compare the average video rate of the first 60 seconds between the BBA-1 algorithm and the *Control* algorithm, the BBA-1 algorithm achieves 700kb/s less than the *Control*. Before the client builds up its buffer to the size of the reservoir, the BBA-1 algorithm will always request for $R_{\min}$, as it is the only *safe* rate given the buffer occupancy. In the next section, we will further improve the video rate by entering into the *risky* area and develop techniques to minimize the risk.

5.3 Ramping Up Faster During the Startup Phase

As discussed in the previous section, most of the differences in video rate between BBA-1 and the *Control* algorithm can be accounted for by the startup phase, i.e., after starting a new video or seeking to a new point. During the startup phase, the playback buffer starts

---

[3]The hypothesis of BBA-1 and $R_{\min}$ *Always* share the same distribution is not rejected at the 95% confidence level (p-value = 0.74).
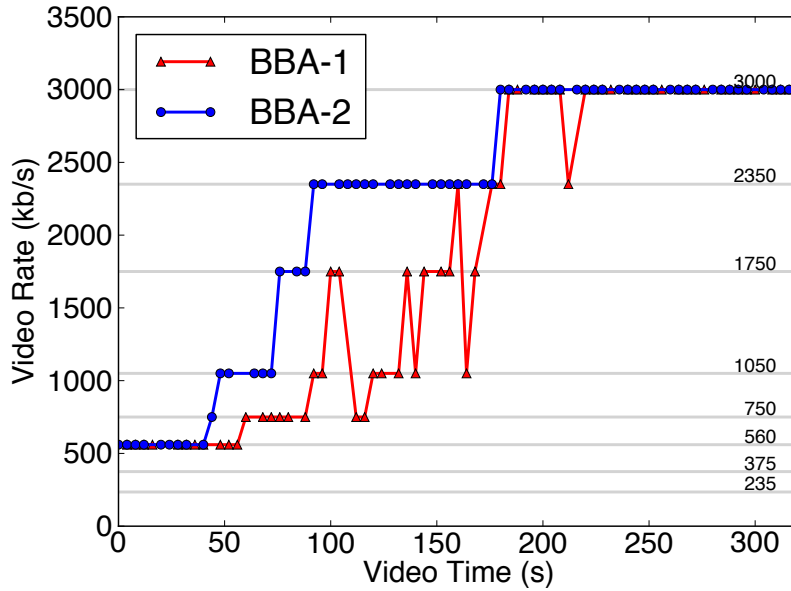
Figure 5.11: Typical time series of video rates for BBA-1 (red) and BBA-2 (blue). BBA-1 follows the chunk map and ramps slowly. BBA-2 ramps faster and reaches the steady-state rate sooner.

out empty and carries no useful information to help us choose a video rate. BBA-1 follows the usual chunk map, starting out with a low video rate since the buffer level is low. It gradually increases the rate as the buffer fills, as shown by the red line in Figure 5.11. BBA-1 is too conservative during startup. The network can sustain a much higher video rate, but the algorithm is just not aware of it yet.

In this section, we test the following hypothesis. During the startup, we can improve the video rate by entering into the risky area; in the steady state, we can improve both video rate and rebuffer rate by using a chunk map. Our next algorithm, BBA-2, tries to be more aggressive during the startup phase. When possible, BBA-2 ramps up quickly and fills the buffer with a much higher rate than what the map suggests.

From Figure 5.6, we know that the change of the buffer, $\Delta B = V - (ChunkSize/c[k])$, captures the difference between the instantaneous video rate and system capacity. Now, assuming the current video rate is R_i, to safely step up a rate, $c[k]$ needs to be at least R_{i+1} to avoid rebuffers. In other words, we require $\Delta B \geq V - (ChunkSize/R_{i+1})$. Further, since videos are encoded in VBR, the instantaneous video rate can be much higher than the nominal rate. Let the max-to-average ratio in a VBR stream be e, so that eR_{i+1} represents

the maximum instantaneous video rate in R_{i+1}. When the player first starts up, since there is no buffer to absorb the variation, $c[k]$ needs to be at least larger than eR_{i+1} in order to safely step up a rate. In other words, when considering VBR and the buffer is empty, ΔB needs to be larger than $V - (ChunkSize/(eR_{i+1}))$ for the algorithm to safely step up from R_i to R_{i+1}. According to Figure 5.5, the max-to-average ratio e is around 2 in our system. Since $e = 2$, $R_i/R_{i+1} \sim 2$, and a chunk size can be smaller than half the average chunk size ($ChunkSize \leq 0.5VR_i$), ΔB needs to be larger than $0.875V$s to safely step up a rate when the buffer is empty in our system.

Based on the preceding observation, BBA-2 works as follows. At time $t = 0$, since the buffer is empty, BBA-2 only picks the next highest video rate, if the ΔB increases by more than $0.875V$s. Since $\Delta B = V - ChunkSize/c[k]$, $\Delta B > 0.875V$ also means that the chunk is downloaded *eight times* faster than it is played. As the buffer grows, we use the accumulated buffer to absorb the chunk size variation and we let BBA-2 increase the video rate faster. Whereas at the start, BBA-2 only increases the video rate if the chunk downloads *eight times* faster than it is played, by the time it fills the cushion, BBA-2 is prepared to step up the video rate if the chunk downloads *twice* as fast as it is played. The threshold decreases linearly, from the first chunk until the cushion is full. The blue line in Figure 5.11 shows BBA-2 ramping up faster. BBA-2 continues to use this startup algorithm until (1) the buffer is decreasing, or (2) the chunk map suggests a higher rate. Afterwards, we use the $f(B)$ defined in the BBA-1 algorithm to pick a rate.

Note that BBA-2 is using ΔB during startup, which encodes a simple capacity estimate: the throughput of the last chunk. This design helps make the algorithm more aggressive at a point when the buffer has not yet accumulated enough information to accurately determine the video rate to use. Nevertheless, note that our use of capacity estimation is restrained. We only look at the throughput of the last chunk, and crucially, once the buffer is built up and the chunk map starts to suggest a higher rate, BBA-2 becomes buffer-based—it picks a rate from the chunk map, instead of using ΔB. In this way, BBA-2 enables us to enjoy the improved steady-state performance of the buffer-based approach, without sacrificing overall bitrate due to a slow startup ramp.

5.3.1 Experimental Results

We ran our experiments during the same time period and with the same pool of users as the previously described experiments, which all occurred on September 6th (Friday)
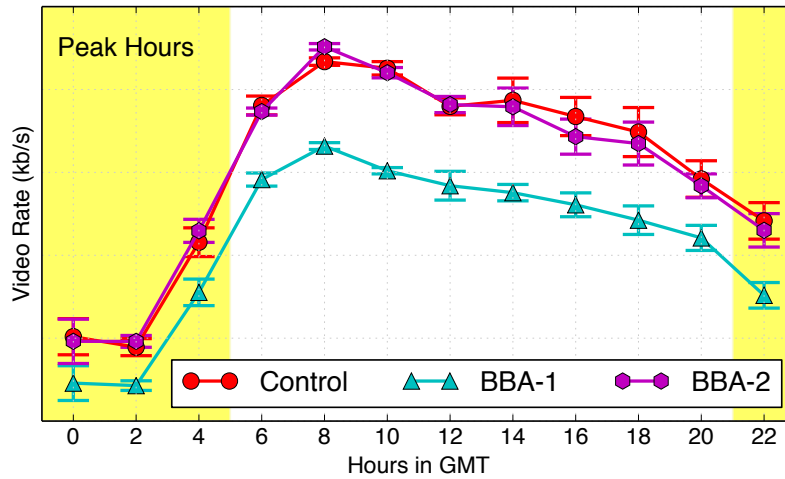
(a) Average video rate during the day.
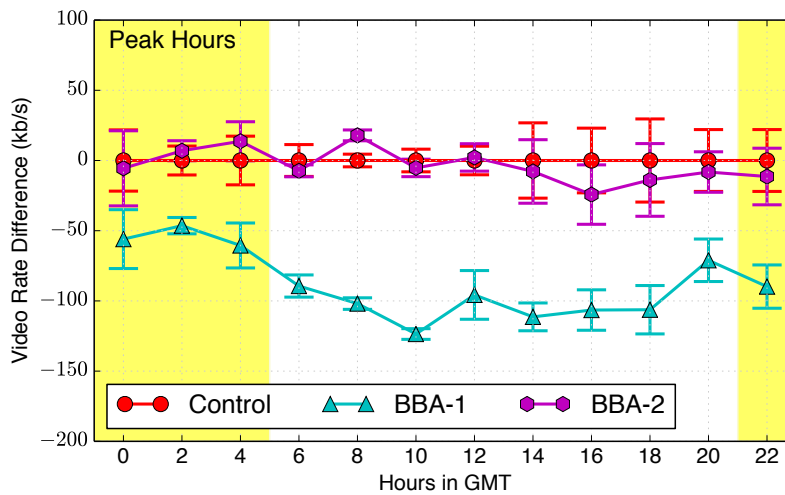
(b) The difference on video rate per two-hour window between the *Control*, BBA-1, and BBA-2 algorithms.

Figure 5.12: Comparison of video rate between the *Control*, BBA-1, and BBA-2 algorithms. The error bars represent the variance of video rates from different days in the same two-hour period. BBA-2 achieved a similar video rates to the *Control* algorithm overall.
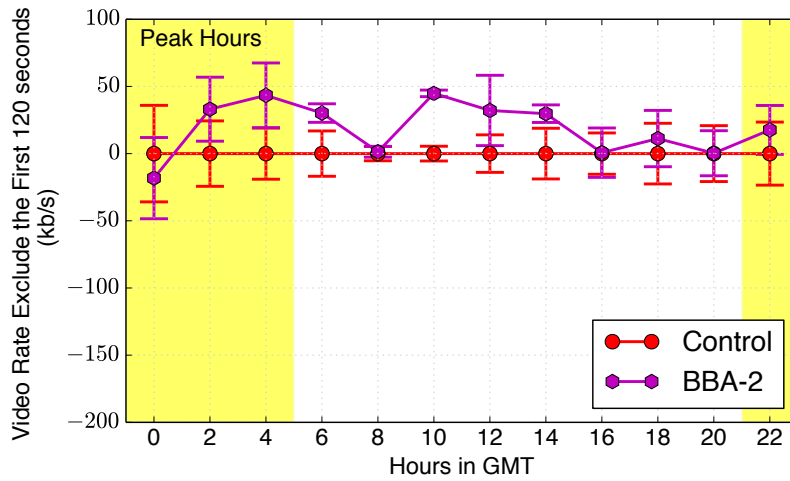
Figure 5.13: Comparison of average video rate during the steady state between *Control* and BBA-2. The steady state is approximated as the period after the first two minutes in each session. BBA-2 achieved better video rate at the steady state.

and 9th (Monday), 2013. Figure 5.12(a) shows the video rates from this experiment, and Figure 5.12(b) shows the difference in the average video rate between *Control*, BBA-1, and BBA-2. From the figures, we see that BBA-2 does indeed increase the video rate. With a faster startup-phase ramp, the video rate with BBA-2 is almost indistinguishable from the *Control* algorithm. This supports our hypothesis that the lower video rates seen by BBA-0 and BBA-1 were due to their conservative rate selection during startup. Furthermore, if we exclude the first two minutes as an approximation of the steady state, the average video rate of BBA-2 is mostly higher than *Control*, as shown in Figure 5.13. This observation also verifies our discussion in Chapter 4: The buffer-based approach is able to better utilize network capacity and achieve higher average video rate in the steady state.

Figure 5.14 shows absolute and normalized rebuffers. BBA-2 slightly increases the re-buffer rate. BBA-2 operates in the *risky* zone of Figure 5.8 and therefore will inevitably rebuffer more often than BBA-1, which only operates in the *safe* area. Nevertheless, the improvements are significant relative to *Control*: BBA-2 maintains a *10–20% improvement in rebuffer rate* compared to the *Control* algorithm during peak hours.

So far, we have successfully relaxed the four idealized assumptions made in Chapter 4. In BBA-0, we handle the finite chunk size and discrete available video rates through a piecewise mapping function. In BBA-1, we handle the VBR encoding through a variable
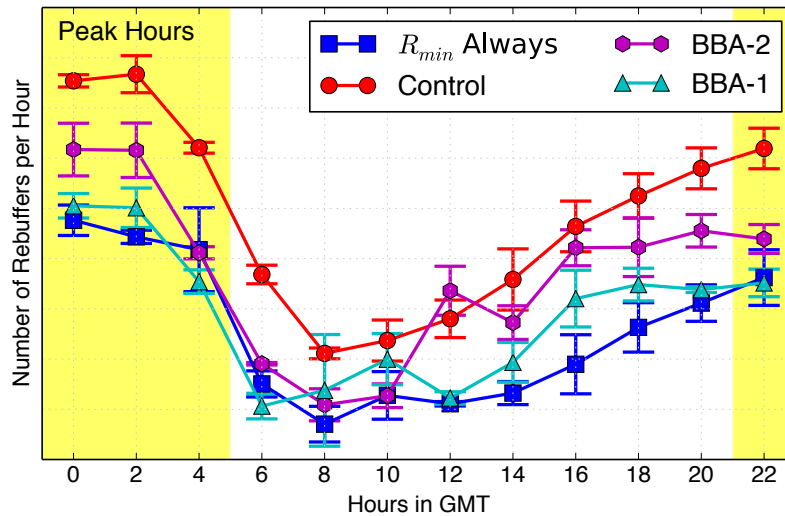
(a) Number of rebuffers per playhour throughout the day.
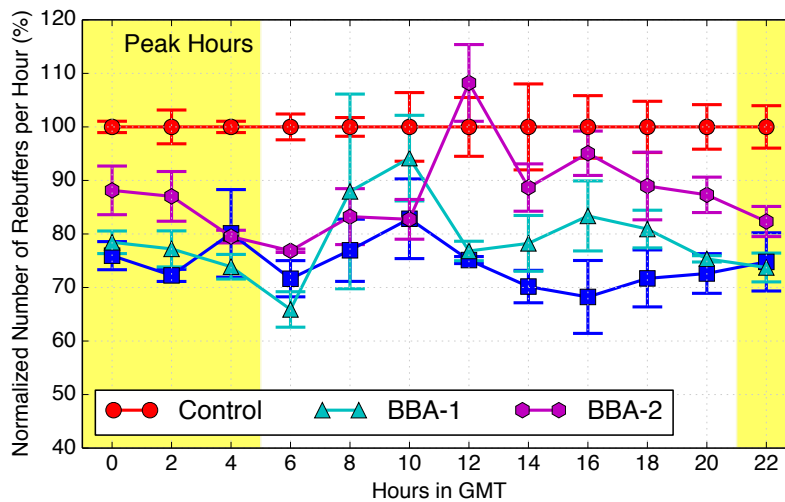
(b) Normalized number of rebuffers per playhour. The number is normalized
to the average rebuffer rate of the *Control* algorithm in each two-hour period.

Figure 5.14: Number of rebuffers per playhour for the *Control*, $R_{\min}$ *Always*, BBA-1, and BBA-2 algorithms. The error bars represent the variance of rebuffer rates from different days in the same two-hour period. BBA-2 has a slightly higher rebuffer rate compared to BBA-1, but still achieved 10–20% improvement compared to the *Control* algorithm during peak hours.

reservoir size and a chunk map. In BBA-2, we further handle the finite video length by dividing each session into two phases. BBA-2 still follows the buffer-based approach in the steady state, and it uses a simple capacity estimation to ramp up the video rate during the startup. The results demonstrate that by focusing on the buffer, we can reduce the rebuffer rate without compromising the video rate. In fact, the buffer-based approach improves the video rate in the steady state.

In the following section, we will further discuss how to extend the buffer-based approach to tackle other practical concerns.

5.4   Handling Other Practical Concerns

In the previous sections, we have shown that the buffer-based approach is able to both reduce rebuffer rate and improve video rate. In this section, we will extend the buffer-based approach and develop techniques to address two other practical concerns: temporary network outage and frequent video switches.

5.4.1   Handling Temporary Network Outage

We have shown that buffer-based algorithms never need to rebuffer if the network capacity is always higher than $R_{\min}$. In this section we explore what happens if the network capacity falls *below* $R_{\min}$, such as during a complete network outage. Temporary network outages of 20–30s are not uncommon, e.g., when a DSL modem retrains or a WiFi network suffers interference. To make buffer-based algorithms resilient to brief network outages, we can reserve part of the buffer by shifting the chunk map curve further to the right.

Figure 5.15 shows the chunk map with outage protection. The buffer will now converge to a higher occupancy than before, providing some protection against temporary network outages. We call this extra portion of buffer the *outage protection.*

How should we allocate buffers to outage protection? One way is to gradually increase the size of outage protection after each chunk is downloaded. In the implementation of BBA-1, we accumulate outage protection by 400ms for each chunk downloaded when the buffer is increasing and still less than 75% full. In the implementation of BBA-2, we only accumulate outage protection after the algorithm exits the startup phase and is using the chunk map algorithm. A typical amount of outage protection is 20–40 seconds at steady
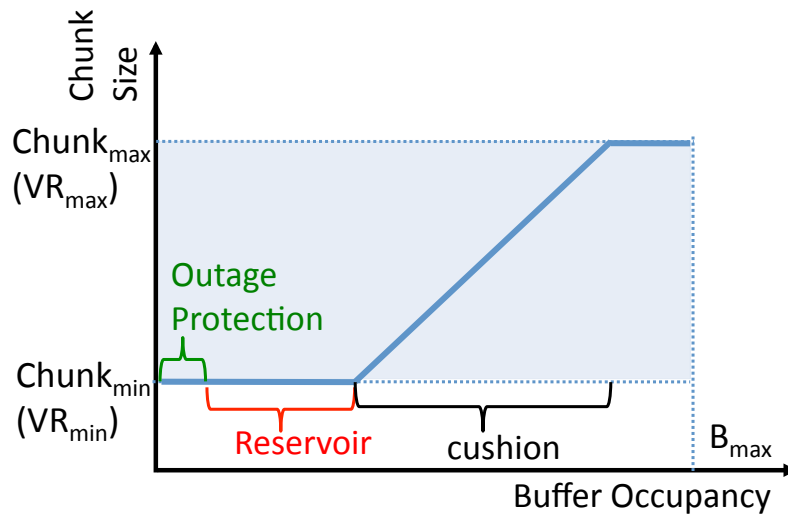
Figure 5.15: To protect against temporary network outage, we allocate part of the buffer as *outage protection*.

state and is bounded at 80 seconds. The downside of this approach is that the chunk map keeps moving and can cause video rates to oscillate.

In the following, we describe an alternative way to protect against temporary network outage, while reducing changes to the chunk map, by combining it with the dynamic reservoir calculation.

5.4.2   Smoothing Video Switch Rate

In Section 5.2, we showed that we can improve the video rate by using a chunk map and dynamic reservoir calculation. However, this choice makes the video rate change frequently, as shown in Figure 5.16. Note that it is debatable as to whether video switching rate really matters to the viewer's quality of experience. For example, if a service offers closely spaced video rates, the viewer might not notice a switch. Nevertheless, in the following we will explore mechanisms to reduce the switching rate and introduce a modified algorithm, *BBA-Others*, to address this issue. We will see that by smoothing the changes, we can at least match the switching rate of the *Control* algorithm.

There are two main reasons our buffer-based algorithms increase the frequency of video-rate switches. First, when we use the chunk map, there is no longer a fixed mapping function between buffer levels and video rates. Instead, buffer levels are mapped to chunk sizes, and

Figure 5.16: Comparison of the average video switching rate for the *Control*, BBA-1, and BBA-2 algorithms. The error bars represent the variance of video switching rates from different days in the same two-hour period. After switching from using a rate map to using a chunk map, the video switching rate of BBA-1 and BBA-2 is much higher than the *Control* algorithm.

the nominal rate might change every time we request a new chunk. Even if the buffer level remains constant, the chunk map will cause BBA-1 to frequently switch rates, since the chunk size in VBR encoding varies over time, as illustrated in Figure 5.17. We can reduce the chance of switching to a new rate—and then switching quickly back again—by looking ahead to future chunks. When encountering a small chunk followed by some big chunks, even if the chunk map tells us to step up a rate, our new algorithm *BBA-Others* will not do so to avoid a likely step down in the near future. The further this modified algorithm looks ahead, the more it can smooth out rate changes. If, in the extreme, we look ahead to the end of the movie, it is the same as using a rate map instead of a chunk map. In the implementation of BBA-Others, we look ahead the same number of chunks as what we have in the buffer. When the buffer is empty, we pick a rate by only looking at the next chunk; when the buffer is full, we look ahead for the next 60 chunks.[4] Note that BBA-Others only smooths out *increases* in video rate. It does not smooth decreases so as to avoid increasing the likelihood of rebuffering.

---

[4]Our buffer size is 240 seconds and each chunk is 4 seconds.
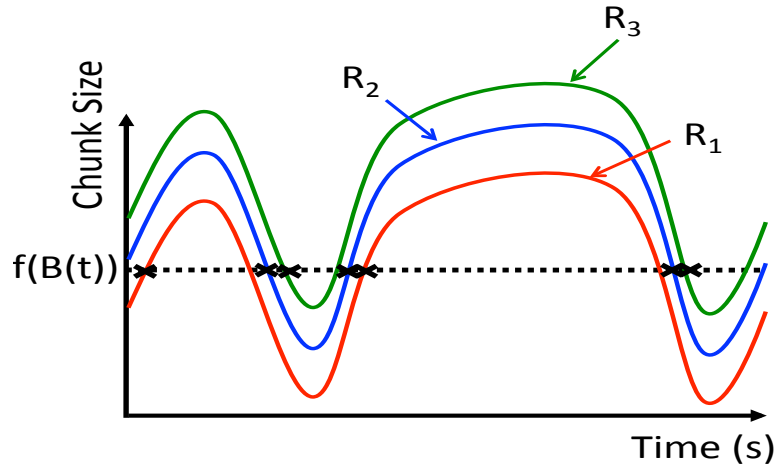
Figure 5.17: A reason using chunk map increases video switching rate. When using a chunk map, even if the buffer level and the mapping function remains constant, the variation of chunk sizes in VBR streams can make a buffer-based algorithm switch between rates. The lines in the figure represent the chunk size over time from three video rates, R_1, R_2, and R_3. The crosses represent the points where the mapping function will suggest a rate change.

To explain the second reason, we look at Figure 5.7. The size of the reservoir is calculated from the chunk size variation in the next 480 seconds. As a result, the reservoir will shrink and expand depending on the size of upcoming chunks. If large chunks are coming up, the chunk map will be right-shifted, and if small chunks are coming up, the chunk map will be left-shifted. Even if the buffer level remains constant, a shifted chunk map might cause the algorithm to pick a new video rate. On top of this, as described in Section 5.4.1, a gradual increase in outage protection will *also* gradually right-shift the chunk map. Hence, we reduce the number of changes by only allowing the chunk map to shift to the right, never to the left, i.e., the reservoir expands but never shrinks. Since the reservoir cannot be shrinked, the reservoir grows faster than it needs to, letting us use the excess for outage protection.

5.4.3   Experimental Results

As before, we randomly pick three groups of real users for our experiment. One third are in the *Control* group, one third always stream at $R_{\min}$, giving us an approximation of the lower bound on rebuffer rate, and one third run the BBA-Others algorithm, which smooths
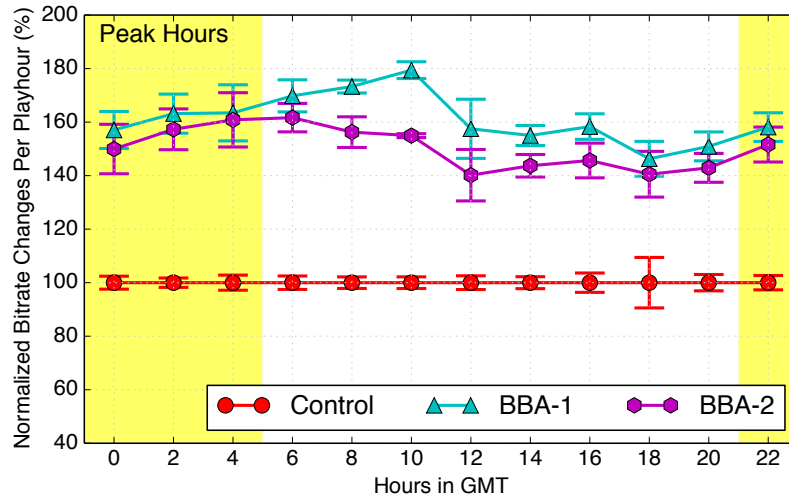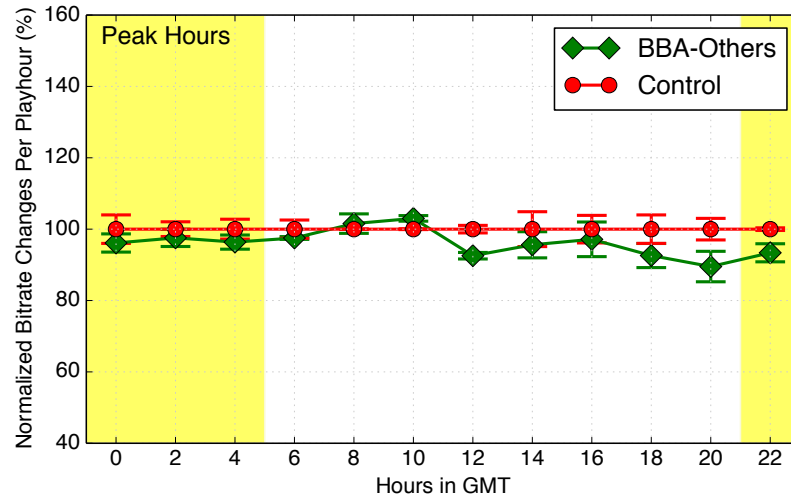
Figure 5.18: Comparison of average video switching rate between *Control* and BBA-Others. The error bars represent the variance of video switching rates from different days in the same two-hour period. BBA-Others smoothes the frequency of changes to the video rate, making it similar to the *Control* algorithm.

the switching rate by looking ahead and by only allowing the chunk map to be right-shifted. The experiment was conducted between September 20th (Friday) and 22nd (Sunday), 2013.

Figure 5.18 shows that the video rate changes much less often with BBA-Others than with BBA-1 or BBA-2 (Figure 5.16). In fact, BBA-Others is almost indistinguishable from *Control*—sometimes higher, sometimes lower.[5] Figure 5.19 shows the video rate for BBA-Others. Since we does not allow the chunk map to be left-shifted, BBA-Others switches up more conservatively than BBA-2. Although the video rate is almost the same as *Control*, we trade about 20kb/s of video rate compared to BBA-2 in Figure 5.12.[6] As other buffer-based algorithms, BBA-Others improves the rebuffer rate, since we do not change the frequency of switches to a lower rate. As shown in Figure 5.20, BBA-Others improves the rebuffer rate by 20–30% compares to the *Control* algorithm.

---

[5]The numbers are normalized to the average switching rate in *Control* for each two-hour window.

[6]This is only an approximation, since the experiments in Figure 5.19 and 5.12 ran in two different weekends in September, 2013.
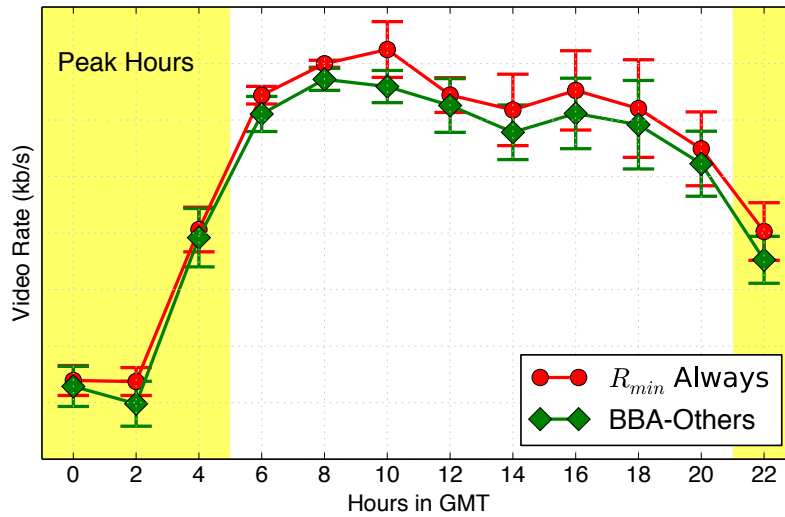
(a) Average video rate during the day.
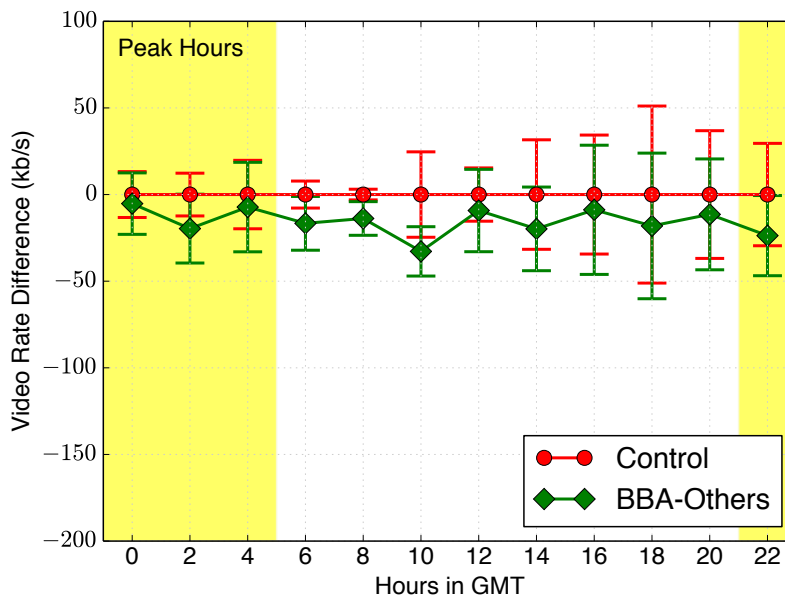
(b) The difference on video rate per two-hour window between *Control*, BBA-Others.

Figure 5.19: Comparison of video rate between *Control* and BBA-Others. The error bars represent the variance of video rates from different days in the same two-hour period. BBA-Others achieves a similar video rate during the peak hours but reduces the video rate by 20–30kb/s during the off-peak.
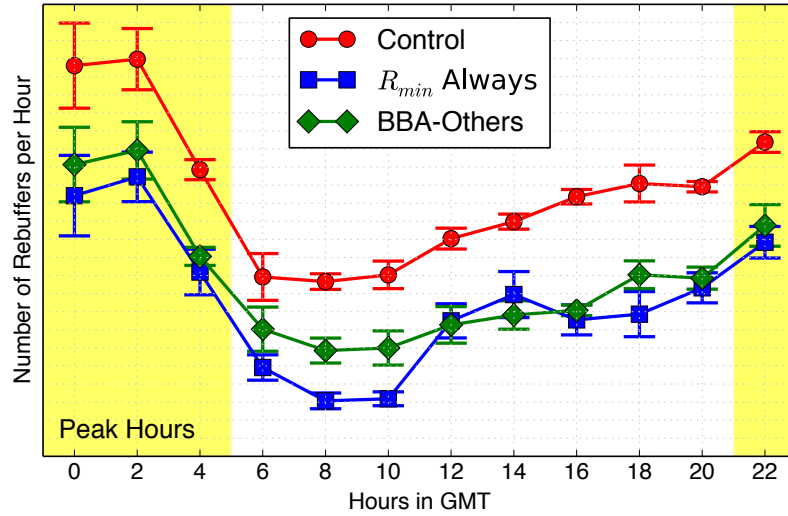
(a) Number of rebuffers per playhour throughout the day.
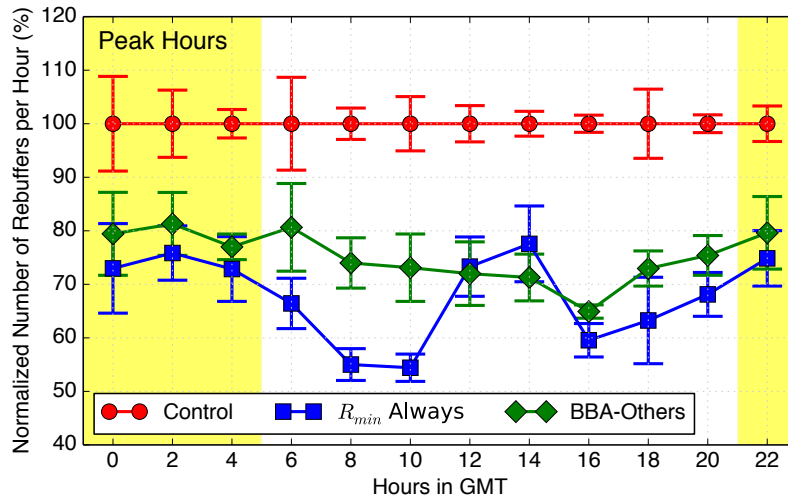
(b) Normalized number of rebuffers per playhour. The number is normalized to the average rebuffer rate of the *Control* algorithm in each two-hour period.

Figure 5.20: Number of rebuffers per playhour for *Control* and BBA-Others. The error bars represent the variance of rebuffer rates from different days in the same two-hour period. BBA-Others reduces rebuffer rate by 20–30% compared to the *Control* algorithm. Values are normalized to the average rebuffer rate in the *Control* algorithm for each two-hour window.
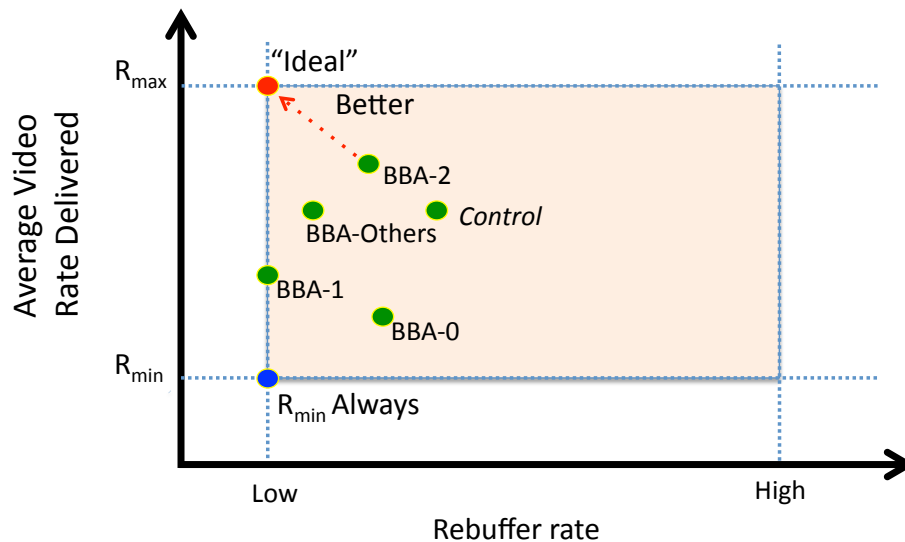
Figure 5.21: Summary of the performance of buffer-based algorithms.

5.5   Summary

In this chapter, we tested the viability of the buffer-based approach through a series of experiments spanning tens of thousands of real users on Netflix. We started with a simple design that directly chooses the video rate based on the current buffer occupancy. The experimental results revealed that capacity estimation is indeed unnecessary in steady state; however, using *simple* capacity estimation based on immediate past throughput is important during the startup phase, when the buffer itself is growing from empty. This simpler approach allows us to reduce the rebuffer rate by 10–20% compared to a production ABR algorithm while delivering a similar average video rate and a higher video rate in steady state. We also developed techniques to protect against temporary network outage and to smooth video switching rate. By accumulating more video in the buffer and trading 20kb/s video rate, we are able to achieve similar video switching rate and reduce 20–30% rebuffer rate compared to the *Control*. Figure 5.21 summarizes the performance of the series of buffer-based algorithms developed in this Chapter. Table 5.2 extends Table 5.1 and summarizes the key advances between the algorithms. In summary, when closely-spaced video rates are offered and video switching rate is not a concern, BBA-2 works the best. Otherwise, BBA-Others provides an alternative to balance between video rates and video switching rate.

Algorithm	Design Goal	Key Advances
BBA-0	(1) Handling finite chunk size. (2) Handling discrete video rates.	Use a piecewise linear function as the rate map.
BBA-1	Handling VBR encoding.	Dynamically calculate the reservoir size. Generalize rate map to chunk map.
BBA-2	Handling finite video length.	Divide sessions into two phases: startup phase and steady state.
BBA-Others	(1) Handling temporary outage. (2) Smoothing switching rate.	Minimize the changes on the mapping function.

Table 5.2: Summary of the key advances between BBA-0, BBA-1, BBA-2, and BBA-Others.

Chapter 6

Related Work

As Internet video becomes popular, considerable effort goes into understanding and improving users' experience of streaming video. While many works focus on the design of ABR algorithms, many others focus on other aspects of the video streaming system. These efforts include understanding users' viewing experience, developing video encoding schemes, and architecting the streaming system. Figure 6.1 summarizes all these efforts. In this chapter, we will first discuss related works on ABR algorithms before going into other related works.

6.1 ABR Algorithm Designs

The earlier studies on ABR algorithms focus on measuring and understanding the discrepancies of the current practice. These discrepancies mainly come from inaccurate estimates of network capacity, and many algorithms have been proposed to mitigate the impact of inaccurate estimates. Among these algorithms, the most closely related work is to use buffer occupancy to adjust the capacity estimates. We will call these algorithms *buffer-aware ABR algorithms* to differentiate them from our buffer-based approach. The main difference is that buffer-aware ABR algorithms still pick video rate based on capacity estimates and use buffer occupancy only to adjust the estimates. In contrast, our buffer-based approach picks video rate directly from the buffer occupancy and uses simple capacity estimation only during the startup. In the following, we will discuss both the earlier measurement studies and the buffer-aware ABR algorithms.
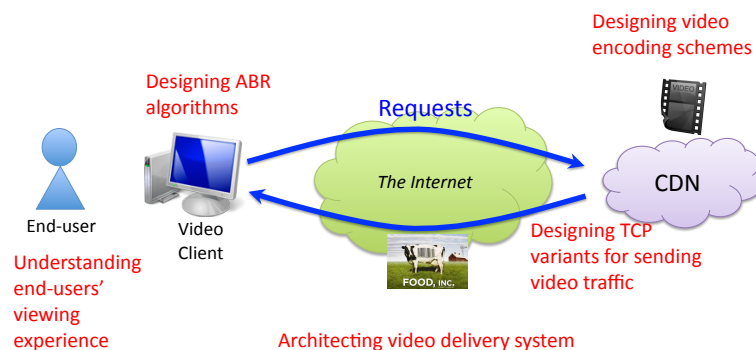
Figure 6.1: Overview of the related works on better streaming video over the Internet.

Understanding the Impact of Inaccurate Estimates.   Commercial ABR algorithms estimate the future capacity using a moving average of recent throughput measurements [36].   Prior works have shown that sudden changes in available network capacity confuse these ABR algorithms, causing the algorithms to either overestimate or underestimate the available network capacity [2, 3, 9, 16, 21].

The overestimation leads to unnecessary rebuffers [3, 9].   In this thesis, we quantify how often unnecessary rebuffers happen in a production system and show that 20–30% of rebuffers are unnecessary.   Based on this observation, we then propose the buffer-based approach to reduce unnecessary rebuffers.

The underestimation not only fills the buffer with video chunks of lower quality, but also leads to the ON-OFF traffic pattern in video traffic: when the playback buffer is full, the client pauses the download until there is space.   As we have shown in Chapter 2, in the presence of competing TCP flows, the ON-OFF pattern can trigger a bad interaction between TCP and the ABR algorithm, causing a further underestimate of capacity and a downward spiral in video quality. When competing with other video players, overlapping ON-OFF periods can confuse capacity estimation, leading to oscillating quality and unfair link share among players [2, 16, 21].   In our work, since we request only R_{max} when the buffer approaches full, the ON-OFF traffic pattern appears only when the available capacity is higher than R_{max}. When competing with a long-lived TCP flow, our algorithm continues to request R_{max} when the ON-OFF pattern occurs, avoiding the downward spiral. When competing with other video players, if the buffer is full, all players have reached R_{max}, and so the algorithm is fair.

Buffer-aware ABR Algorithms. Many algorithms have been proposed to use buffer occupancy, in addition to capacity estimation, to pick a video rate [8, 10, 26, 35]

QDash [26] upgrades the video quality based on capacity estimation and smoothes the downgrade of video quality with the buffer. Other algorithms leverage control theory and use buffer occupancy as one of the feedback signals to select a video rate [8, 10, 35]. Tian et al. [35] first predict future TCP throughput with a machine-learning algorithm. An adjustment function is then computed based on buffer occupancy through a controller, and a video rate is selected based on the adjusted throughput prediction. The results show that buffer occupancy is useful to balance the trade-off between responsiveness and smoothness for the video rate. Elastic [8] first measures the network capacity through a harmonic filter. Elastic then drives the buffer to a set-point through a controller, which considers buffer occupancy as the feedback signal and models capacity measurement as a disturbance. By keeping the buffer occupancy at the set-point and keeping the buffer from being full, Elastic is able to utilize the capacity fully and avoid the ON-OFF traffic pattern.

These prior works reveal that buffer occupancy provides important information for selecting a video rate. In this thesis, we take another step forward and show that buffer occupancy is in fact the primary state variable that an ABR algorithm should control. We begin with a simple design that directly chooses the video rate according to the current buffer occupancy and uses simple capacity estimation only when the buffer itself is growing from empty.

6.2   Quality Metrics and User Engagement

One ongoing research effort asks *which* streaming quality metrics (such as rebuffers and video rate) affect user engagement (such as play time and retention), and *how* these metrics affect engagement. The answers to these questions help guide ABR algorithm designs. For example, it is easier to handle the trade-off between video rate and join delay if the algorithm designer knows which quality metric weights more for user engagement.

Large-scale video streaming services approach the problem by analyzing traces from real users [7, 11, 19]. Correlations were first established between streaming quality metrics and user engagement [11, 19]. The analyses have shown a negative correlation between rebuffers and play time, a positive correlation between join delay and session abandonment, and a positive correlation between video rate and play time. However, correlational observations

are not enough to understand the relation between quality metrics and user engagement. There are two main reasons for the insufficiency [7, 19]. First, quality metrics are interdependent. For example, while streaming video at a higher video rate leads to better quality, it may also lead to a longer join delay and higher rebuffer rate. Second, many external factors confound the relation between quality metrics and user engagement. For example, in addition to video quality, the content of a video also affects the play time.

Quasi-Experimental Design (QED) is proposed to tackle these issues [19]. Instead of conducting controlled experiments, QED uses existing traces and matches sessions that share the values of all variables except the treating variable. In consequence, the differences of the outcome can be attributed to the treatment and establish the causality. The analyses strengthen the correlational observations and have shown that an increase on rebuffer causes shorter play time and an increase on join delay causes more viewing abandonment. More quantitatively, the results show that a user who experiences a rebuffer delay equal to 1% of the video duration watches 5% less of the video than a similar user who experiences no rebuffer at all. The results also show that when join delay is more than 2 seconds, each incremental delay of 1 second increases abandonment rate by 5.8%.

A QoE model is further proposed to unify the impact of quality metrics on user engagement [7]. Three main confounding factors—type of video (live vs. VOD), device (PC vs. mobile devices), and users' connectivity (wired vs. wireless)—are first identified through information gain analysis. The traces are then split on the basis of the confounding factors. The impact of quality metrics is modeled for each split through decision trees and unified through a logical union. The results show that the proposed QoE model provides 70% accuracy in predicting engagement, and using the QoE model to guide the choice of CDN and video rate improves user engagement by 20%.

These prior works are complementary to this thesis. This thesis focuses on both reducing rebuffers and improving video rate, and both quality metrics are shown to be important to user engagement. The results from this thesis can serve as a foundation for considering other metrics and algorithmic designs assisted by QoE models.

6.3 Video Encoding Schemes

The concept of adaptive quality for Internet video streaming was introduced in the 1990s, long before the recent development of HTTP-based adaptive streaming. These early algorithms adapt quality by using layered encoding schemes [20, 31].

In block-oriented encoding schemes, which are commonly used in streaming services today, each video is encoded into a number of video rates and each video rate is saved as an independent and separate file. One adapts the video quality by requesting a different file. Layered encoding schemes, on the other hand, encode a video into a hierarchy of cumulative layers. The base layer encodes the lowest video rate, and each additional layer contains only the additional information for the next highest video rate. As more layers are received by the decoder, the video is reconstructed with a higher video rate. Layered encoding schemes allow progressive video reconstruction—video players always first download the base layer and request extra layers when extra capacity is available. Thus, layered encoding schemes provide more resilience toward rebuffer than block-oriented encoding schemes. Nevertheless, video streaming today uses block-oriented codecs instead of layered codecs, because layered encoding requires 20–30% more bits to achieve the same quality as block-oriented encoding [17]. Since the cost of video streaming today is dominated by the bandwidth cost, block-oriented codecs are currently more favored in the industry [17].

In this thesis, we focus only on ABR algorithms for block-oriented codecs, but our results reveal that many aspects of encoding profoundly affect the performance of an ABR algorithm. For example, the max-to-average ratio determines the amount of buffer the algorithm needs to preserve to absorb the variation caused by the encoding, and the distance between neighboring video rates determines the effect of video rate switches. We believe all ABR algorithms need to take the codec designs into consideration, and codec design is an indispensable part of improving users' streaming experience.

6.4 Other Designs in Streaming Systems

As shown in Figure 6.1, there are also efforts on other parts of the video streaming system to improve users' streaming experience. These efforts include optimizing congestion control algorithms for video traffic and designing a centralized control plane to optimize global performance. Below, we will overview these efforts.

Global Control Plane. Client-side ABR algorithms try to make the best decision based on local observations. Their distributed nature yields system scalability, and arguably each client has the best position to observe local events. However, the decisions of these algorithms are reactive and optimize only the performance of a single client. Thus, a centralized control plane is proposed to optimize the global performance through aggregating measurements [22]. For example, a global control plane can proactively shift clients to a better performing CDN *before* clients experience congestion. The simulation results show that optimal CDN selection can potentially halve the buffering ratio, i.e., the amount of time spent on rebuffering over the overall video play time. The potential benefits from CDN augmentation mechanisms, such as CDN federation and peer-assisted CDN-P2P hybrid model, are also investigated [6]. Our work is complementary to these efforts and will benefit from the global optimization.

Congestion Control Algorithms. TCP was designed as a general-purpose transport protocol and has served general Internet traffic fairly well. Most variants of TCP today, however, are not optimized for video traffic, and many streaming services are exploring different congestion control algorithms for video traffic. YouTube proposes to pace the traffic-sending rate according to the requested video rate to smooth bursty TCP traffic [12, 18]. The result shows that pacing video traffic can reduce both RTT and packet loss rate. Akamai adopts FastTCP for video traffic, instead of mainstream TCP variants, such as TCP Cubic or TCP New Reno. FastTCP uses queueing delay, instead of packet losses, as a congestion signal [37]. The result shows that FastTCP can sustain a higher video rate on average and reduce rebuffers [1]. When adopting a new congestion control algorithm, it is important to avoid creating interactions between congestion control and video rate adaptation, as Chapter 2 have shown that the interactions can result in a downward spiral in video quality. Our work is complementary to this effort and will benefit from a better congestion control algorithm.

In summary, besides ABR algorithm designs, there are also many other efforts to improve various parts of video streaming systems, including quantifying user experience, developing video encoding schemes, designing new system architecture, and congestion control algorithms. Our work is complementary to these efforts and will benefit from a better system design.

Chapter 7

Conclusion

ABR algorithms are balancing between two overarching goals. They try to maximize video rate while minimizing rebuffers. Existing ABR algorithms approach the problem by estimating future capacity from past observations. In this thesis, we showed the surprising result that inaccurate estimation can confuse the algorithms: underestimation can mislead the algorithm to further underestimate the capacity and unnecessarily drop to the lowest video rate; overestimation can mislead the algorithm to pick an unsustainable video rate and cause rebuffers. However, in an environment with highly variable throughput, which is commonly observed in commercial services, accurate estimation is a significant challenge. In this thesis, we proposed a different approach to design ABR algorithms: we consider using *only* the buffer to choose a video rate, and then ask *when* capacity estimation is needed.

Following this design approach, we observed two phases of operation. In steady-state phase, when the buffer has been built up and encodes useful information on available capacity, we rely only on the current buffer occupancy to pick a video rate and dispense entirely with capacity estimation. We formally proved that this pure buffer-based approach helps reduce unnecessary rebuffers and improve video rate at the steady state. In startup phase, i.e., the first few minutes of a session, because the buffer is still growing from empty and carries little information, crude capacity estimation is useful to quickly ramp up the video rate and drive the algorithm into the steady state. This is very similar to the way slow-start works in TCP—when a connection starts, the congestion control algorithm knows nothing about network conditions and opens the window quickly to approach the available capacity until packet losses is induced. In our case, while ABR algorithms also ramp up the video rate quickly, they need to do it in a controlled manner to prevent unnecessary rebuffers.

We further develop algorithms to carefully navigate the startup phase by considering both the current buffer occupancy and the immediate past throughput.

We tested our algorithms in a Netflix browser-based video player in September, 2013 and demonstrated that this approach works well with over half a million users from three continents. We find that *this design approach can reduce the rebuffer rate by 10–20% compared to Netflix's then-default ABR algorithm, while improving the steady-state video rate.* With hindsight, it is perhaps not surprising that buffer occupancy is the right control signal to use at the steady state, because it is the very value we are trying to prevent from over-running or under-running at the client. While crude estimation is proved to be useful during the startup state, this thesis observed the following design principle: rather than presuming that capacity estimation is required, one should begin by using *only* the buffer and add capacity estimation *when* it is needed. Building on this simple principle, we believe there are many more algorithms yet to be invented, such as a better algorithm for the startup phase.

The demand for Internet video streaming services is rising, and user expectations for the viewing experience are also soaring. At the time of this writing (2014), major streaming services, such as Netflix and YouTube, can stream videos in 4K quality (15–22Mb/s). Given that the current Internet capacity in an average US household is only 4–8Mb/s, video rate adaptation will continue to be a challenge for many years to come. There are still many open questions in this field. For example, the coordination between encoding schemes and ABR algorithms, the division of labor between the global and the local control, and the impact of quality metrics on user engagement. This thesis focuses on designing ABR algorithms, and our results show that the buffer-based design approach provides a promising roadmap to address this challenge. We believe the design principle derived from this thesis will continue helping ABR algorithm designers to deal with the increasingly challenging environment in the future.

Bibliography

[1] Improving Online Video Quality and Accelerating Downloads. `http://www.akamai.com/dl/akamai/Akamai-Improving-Online-Video-Quality-and-Accelerating-Downloads.pdf`.

[2] Saamer Akhshabi, Lakshmi Anantakrishnan, Ali C. Begen, and Constantine Dovrolis. What Happens When HTTP Adaptive Streaming Players Compete for Bandwidth? In *ACM NOSSDAV*, June 2012.

[3] Saamer Akhshabi, Constantine Dovrolis, and Ali C. Begen. An Experimental Evaluation of Rate Adaptation Algorithms in Adaptive Streaming over HTTP. In *ACM MMSys*, 2011.

[4] Mark Allman, Vern Paxson, and Ethan Blanton. TCP Congestion Control. RFC 5681 (Draft Standard), September 2009.

[5] Mark Allman, Vern Paxson, and W. Richard Stevens. TCP Congestion Control. RFC 2581 (Proposed Standard), April 1999. Obsoleted by RFC 5681, updated by RFC 3390.

[6] Athula Balachandran, Vyas Sekar, Aditya Akella, and Srinivasan Seshan. Analyzing the Potential Benefits of CDN Augmentation Strategies for Internet Video Workloads. In *Proceedings of the ACM IMC*, Barcelona, Spain, October 2013.

[7] Athula Balachandran, Vyas Sekar, Aditya Akella, Srinivasan Seshan, Ion Stoica, and Hui Zhang. Developing a Predictive Model of Quality of Experience for Internet Video. ACM SIGCOMM, Hong Kong, China, August 2013.

[8] Luca De Cicco, Vito Caldaralo, Vittorio Palmisano, and Saverio Mascolo. ELASTIC: a Client-side Controller for Dynamic Adaptive Streaming over HTTP (DASH). In *IEEE Packet Video Workshop*, December 2013.

[9] Luca De Cicco and Saverio Mascolo. An Experimental Investigation of the Akamai Adaptive Video Streaming. USAB'10, Klagenfurt, Austria, November 2010.

[10] Luca De Cicco, Saverio Mascolo, and Vittorio Palmisano. Feedback Control for Adaptive Live Video Streaming. In *Proceedings of the Multimedia Systems Conference (MMSys)*, 2011.

[11] Florin Dobrian, Asad Awan, Dilip Joseph, Aditya Ganjam, Jibin Zhan, Vyas Sekar, Ion Stoica, and Hui Zhang. Understanding the Impact of Video Quality on User Engagement. In *Proceedings of the ACM SIGCOMM*, Toronto, Canada, August 2011.

[12] Monia Ghobadi, Yuchung Cheng, Ankur Jain, and Matt Mathis. Trickle: Rate Limiting YouTube Video Streaming. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2012.

[13] Xianping Guo and Onesimo Hernandez-Lerma. *Continuous-Time Markov Decision Processes*. Wiley, 2009.

[14] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, Timid, and Unstable: Picking a Video Streaming Rate is Hard. In *ACM IMC*, November 2012.

[15] Hulu Blog: A Big 2012. `http://blog.hulu.com/2012/12/17/a-big-2012/`.

[16] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. ACM CoNEXT '12, 2012.

[17] Hari KALVA, Velibor ADZIC, and Borko FURHT. Comparing mpeg avc and svc for adaptive http streaming. In *Proceedings of 2012 IEEE International Conference on Consumer Electronics*, 2012.

[18] Leonidas Kontothanassis. Content Delivery Considerations for Different Types of Internet Video. In *Proceedings of the ACM Multimedia Systems Conference (MMSys) – Keynote*, Febrary 2012.

[19] S. Shunmuga Krishnan and Ramesh K. Sitaraman. Video Stream Quality Impacts Viewer Behavior: Inferring Causality Using Quasi-Experimental Designs. In *Proceedings of the ACM IMC*, Boston, MA, USA, November 2012.

[20] Xue Li, Sanjoy Paup, and Mostafa Ammar. Layered video multicast with retransmission(lvmr): Evaluation of hierarchical rate control. In *IEEE INFOCOM*, 1998.

[21] Zhi Li, Xiaoqing Zhu, Josh Gahm, Rong Pan, Hao Hu, Ali C. Begen, and Dave Oran. Probe and adapt: Rate adaptation for http video streaming at scale. In `http:// arxiv. org/ pdf/ 1305. 0510`.

[22] Xi Liu, Florin Dobrian, Henry Milner, Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. A Case for a Coordinated Internet Video Control Plane. ACM SIGCOMM, Helsinki, Finland, August 2012.

[23] Yao Liu, Sujit Dey, Don Gillies, Faith Ulupinar, and Michael Luby. A Study on Quality of Experience for Adaptive Streaming Service. IEEE Packet Video Workshop, December 2013.

[24] John W. Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying 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.

[25] Horacio Marquez. Lyapunov stability. In *Nonlinear Control Systems: Analysis and Design*. Wiley, 2003.

[26] Ricky K. P. Mok, Xiapu Luo, Edmond W. W. Chan, and Rocky K. C. Chang. QDASH: a QoE-aware DASH system. In *Proceedings of the ACM Multimedia Systems Conference (MMSys)*, 2012.

[27] Netflix Quarterly Report: 2012 Q4. http://ir.netflix.com/results.cfm.

[28] Sandvine: Global Internet Phenomena Report 2012 H2. `http://www.sandvine. com/downloads/documents/Phenomena_2H_2012/Sandvine_Global_Internet_ Phenomena_Report_2H_2012.pdf`.

[29] Sandvine: Global Internet Phenomena Report 2013 H2. `https://www. sandvine.com/downloads/general/global-internet-phenomena/2013/ 2h-2013-global-internet-phenomena-report.pdf`.

[30] Netflix ISP Speed Index. `http://ispspeedindex.netflix.com/`.

[31] Reza Rejaie, Deborah Estrin, and Mark Handley. Quality adaptation for congestion controlled video playback over the internet. In *ACM SIGCOMM*, 1999.

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

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

[34] Hari Sundaram, Wu-Chi Feng, and Nicu Sebe. Flicker Effects in Adaptive Video Streaming to Handheld Devices. In *Proceedings of the ACM MM*, Scottsdale, AZ, USA, November 2011.

[35] Guibin Tian and Young Liu. Towards Agile and Smooth Video Adaptation in Dynamic HTTP Streaming. ACM CoNEXT, December 2012.

[36] Mark Watson. HTTP Adaptive Streaming in Practice. In *Proceedings of the ACM Multimedia Systems Conference (MMSys) – Keynote*, 2011.

[37] David X. Wei, Cheng Jin, Steven H. Low, and Sanjay Hegde. FAST TCP: Motivation, Architecture, Algorithms, Performance. In *IEEE/ACM Transactions on Networking*, 2006.

[38] Private conversation with YouTube ABR team.

ProQuest Number: 28121344

ProQuest