NETWORK AND END-HOST SUPPORT FOR HTTP ADAPTIVE VIDEO STREAMING

A Thesis
Presented to
The Academic Faculty

by

Ahmed Mansy

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computer Science
Georgia Institute of Technology
May 2014

NETWORK AND END-HOST SUPPORT FOR HTTP ADAPTIVE VIDEO STREAMING

Approved by:

Dr. Mostafa Ammar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Ellen Zegura
School of Computer Science
Georgia Institute of Technology

Dr. Constantine Dovrolis
School of Computer Science
Georgia Institute of Technology

Dr. Ling Liu
School of Computer Science
Georgia Institute of Technology

Dr. Ali C. Begen
Video and Content Platforms Research
and Advanced Development
Cisco

Date Approved: 10 March 2014

To my parents, for their encouragement and support

To my wife and my little girl

ACKNOWLEDGEMENTS

For many years, having a PhD degree has been like a dream to me but here I am writing the last page of my PhD dissertation. Completing my PhD has been the most challenging job I have done in my life so far. During my eight year journey at Georgia Tech, I have gone through many ups and downs and experienced both good and bad times, but finally I made it. It was only the help and support of many people that kept me going throughout these years. Although I know I can not give each person the credit they deserve, I will try to express my gratitude to some people who were close to me during this journey.

First and foremost, I would like to express my sincere gratitude to my thesis advisor, Prof. Mostafa Ammar. Completing this thesis would not have been possible without his patience, continuous support, and guidance. I cannot thank him enough for the freedom he gave me over the years in pursuing the research topics I am interested in, and I can only hope to be as good an advisor as he is if I become an advisor one day. I would also like to thank my committee members: Prof. Ellen Zegura, Prof. Constantine Dovrolis, Prof. Ling Liu, and Dr. Ali Begen for their helpful comments and insights that helped me improve the work in this thesis. I owe many thanks also to Bill Ver Steeg from Cisco and Dr. Jaideep Chandrashekar from Technicolor Research Center in Paris for hosting me as a summer intern where I had the chance to work on interesting problems that helped me develop several parts of the work in this thesis.

Having spent many years at the Networking group in the College of Computing, I would like to thank all of my lab mates who made my life at Georgia Tech much easier. Special thanks goes to Mukarram, Murtaza, and Partha for the interesting discussions and sharing research ideas. I also thank Samantha for being always caring and supportive; Amogh, Bilal, and Anirudh for being such a good company to hang out with. Special thanks also goes to Srini for using his thesis as a guideline for writing my thesis.

Moving from Egypt to the United States to start my PhD was difficult at first until I met

many people in Atlanta who made me feel like home; these people have been like family to me for the last few years. Special thanks to Ibrahim for being my closest friend in Atlanta and for always being there whenever I needed him. Thanks for the amazing meals he used to cook for us and the fun times we had with our Egyptian friends. I also thank A. Ebaid for the fun times we had and for introducing me to Ibrahim before leaving Atlanta. Many thanks go to Aly, Nazeem, Nader, and other Egyptian students at Georgia Tech with whom I shared the journey of my PhD. I thank Sherif Guenena for the amazing tennis matches we had and for helping me improve my game. I also thank Rehab and Tamer for being the best couple friends my wife and I had in Atlanta and for being our good neighbors for two years. Special thanks also go to Soumaya and Mohamed Khalifa for putting the effort to bring the Egyptian community in Atlanta together and for introducing me to many good and kind people.

Last but not least, I would like to thank my parents, my sister Randa and my brother Mohamed for their unconditional love and support. There are no words enough to thank my wife, Esraa, for her love, support and patience over the last four years so I will just say "thanks!". I also thank our little girl, Yara, for bringing all the joy and happiness in the world to our lives; God bless her. Finally, I cannot forget to thank my father in law, Dr. Sherif Eltayeby, for his endless love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Video streaming is widely recognized as the next Internet killer application. It was not one of the Internet's original target applications and its protocols (TCP in particular) were tuned mainly for efficient bulk file transfer. As a result, a significant effort has focused on the development of UDP-based special protocols for streaming multimedia on the Internet. Recently, there has been a shift in video streaming from UDP to TCP, and specifically to HTTP. HTTP streaming provides a very attractive platform for video distribution on the Internet mainly because it can utilize all the current Internet infrastructure.

In this thesis we make the argument that the marriage between HTTP streaming and the current Internet infrastructure can create many problems and inefficiencies. In order to solve these issues, we provide a set of techniques and protocols that can help both the network and end-hosts to make better decisions to improve video streaming quality. The thesis makes the following contributions:

- We conduct a characterization study of popular commercial streaming services on mobile platforms. Our study shows that streaming services make different design decisions when implementing video players on different mobile platforms. We show that this can lead to several inefficiencies and undesirable behaviors specially when several clients compete for bandwidth in a shared bottleneck link.

- Fairness between traffic flows has been preserved on the Internet through the use of TCP. However, due to the dynamics of adaptive video players and the lack of standard client adaptation techniques, fairness between multiple competing video flows is still an open issue of research. Our work extends the definition of standard bitrate fairness to utility fairness where utility is the Quality of Experience (QoE) of a video stream. We define QoE max-min fairness for a set of adaptive video flows competing for bandwidth in a network and we develop an algorithm that computes the set of bitrates that should

be assigned to each stream to achieve fairness. We design and implement a system that can apply QoE fairness in home networks and evaluate the system on a real home router.

- A well known problem that has been associated with TCP traffic is the buffer bloat problem. We use an experimental setup to show that adaptive video flows can cause buffer bloat which can significantly harm time sensitive applications sharing the same bottleneck link with video traffic. In addition, we develop a technique that can be used by video players to mitigate this problem. We implement our technique in a real video player and evaluate it on our testbed.

- With the increasing popularity of video streaming on the Internet, the amounts of traffic on the peering links between video streaming providers and Internet Service Providers (ISPs) have become the source of many disputes. Hybrid CDN/P2P streaming systems can be used to reduce the amounts of traffic on the peering links by leveraging users upload bandwidth to redistribute some of the load to other peers. We develop an analysis for hybrid CDN/P2P systems that broadcast live adaptive video streams. The analysis helps the CDN to make better decisions to optimize video quality for its users.

CHAPTER I

INTRODUCTION

Video streaming traffic has become very predominant in the last few years. According to Cisco visual networking index [21], video streaming traffic was accounted for 57% of the total Internet traffic in 2012 and it is expected to exceed 69% in 2017. This growth has happened in part due to the increasing availability of broadband access networks that give users high bandwidth access to the Internet which enables them to stream video with high quality. In addition, a significant part of this growth can be attributed to mobile devices (smartphones and tablets) which are driving online video consumption. Two commercial streaming providers – Netflix and YouTube – are responsible for over 50% of Internet traffic in North America [22]. Netflix is the number one traffic source in North America accounting for 32% of downstream traffic followed by YouTube at 19% [22].

The Internet was not originally designed for multimedia streaming. It was even widely believed that TCP (Transmission Control Protocol) was not a good transport protocol for streaming multimedia. This led the Internet community to develop the Real-time Transport Protocol (RTP) [14] as the main transport protocol for delivering multimedia on the Internet. For many years people believed that RTP over multicast [5] was the ideal way to deliver multimedia on the Internet. Due to many reasons, however, this turned out to be not true.

RTP is usually used on top of UDP [14] and reliable data delivery and congestion control were not part of its original specification. A lot of work has been done to augment RTP with congestion control for both the unicast and multicast cases [13, 50, 67, 96, 110]. However, both RTP congestion control and reliability remain to be open issues. Also, although video codecs are usually designed to deal with minor data loss, a packet loss in an I-frame or the header of an I-frame can cause serious disruptions to the video. In addition, although multicast is implemented on most routers on the Internet, Internet Service Providers (ISPs)

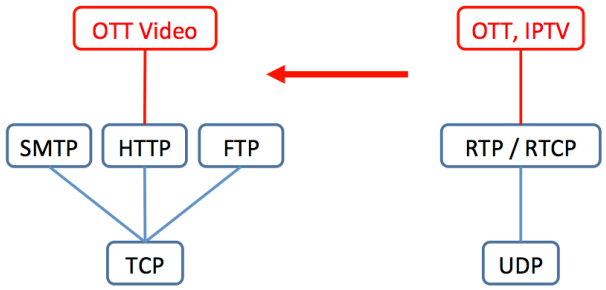


Figure 1: Summary of protocol organization for video over IP

generally do not enable multicast on their networks. Due to these reasons among others RTP on multicast did not provide an attractive platform for providing multimedia streaming on the Internet.

Video over IP has recently taken two forms; Internet Protocol Television (IPTV) and over-the-top video (OTT). IPTV [52] delivers linear TV channels over a managed network that is usually accessed through a special portal such as a Set-Top-Box (STB). IPTV can also deliver Video-On-Demand (VOD) or time-shifted content where a TV show can be played hours after its broadcast time. IPTV networks usually deploy multicast to deliver video content to their subscribers and hence they can enforce certain Quality of Service (QoS) constraints. OTT video, on the other hand, is delivered as a best effort service over the Internet which usually does not provide QoS guarantees.

Recently, video streaming over HTTP (on top of TCP) emerged as a good OTT alternative that avoids most of the problems associated with RTP; figure 1 illustrates the shift of OTT video from UDP to TCP. Since it is based on HTTP, standard Web servers instead of specialized streaming servers can be used to serve video streams. With the wide adoption of Content Distribution Networks (CDNs) as the main platform for content distribution on the Internet, HTTP streaming provided a good technology to reduce operation and setup cost for CDNs. In recent years, HTTP streaming has been used to stream video successfully to millions of Internet users. As a result, this approach has attracted a lot of attention and quickly became the standard for streaming multimedia on the Internet.

Dynamic Adaptive Streaming over HTTP (DASH) [112] defines a standard for OTT

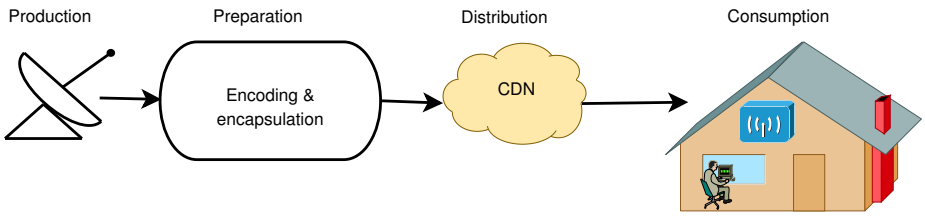


Figure 2: End-to-end video distribution

adaptive video streaming on the Internet. In DASH, a video stream is split into small non-overlapping video segments of equal length and each segment is encoded in multiple bitrates. The video server is basically a Web server that hosts video files representing these segments along with a manifest file that describes the segments and their bitrates. A client streams the video by downloading segments from the video server over HTTP. While downloading segments, the client estimates the available bandwidth to the server and switches among video bitrates accordingly.

An OTT HTTP adaptive video streaming system is usually composed of the following components (see figure 2):

1. **Production.** In this phase, video is captured in the case of live video or premium content is acquired in the case of stored video.

2. **Preparation.** In this stage, the produced video is first split into segments and then encoded into multiple bitrates and different resolutions to fit different viewing profiles. In addition, a manifest file that describes all the video segments and their bitrates is created.

3. **Distribution.** After that, video profiles are injected into the CDN to be distributed closer to viewers. Usually, content owners (e.g. NBC, ESPN, etc.) deliver video to a single (or multiple) entry point of the CDN then the CDN takes care of distributing the video to edge servers. CDN operators usually implement many optimization techniques to decide how video profiles get forwarded to edge servers.

4. **Consumption.** In the final stage, the video is consumed by users. A video player usually downloads the manifest file at the beginning of a streaming session to learn

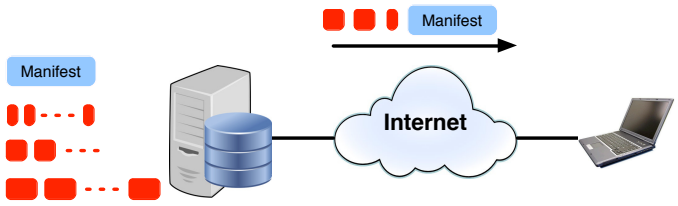


Figure 3: HTTP adaptive streaming

about the available video profiles. After that, it downloads video segments usually starting with the lowest available bitrate. While downloading video segments, the client estimates the available bandwidth and adapts among the different video profiles accordingly. Each segment is specified using an HTTP `GET` request and is downloaded over HTTP.

Below we give an overview of HTTP Adaptive Streaming in more details and introduce terminology that will be used throughout the thesis. We then discuss some of the problems associated with the operation of adaptive streaming on the Internet. After that we discuss the specific problems we investigate in this thesis, then we present the organization of the thesis.

1.1 HTTP Adaptive Streaming

HTTP Adaptive Streaming (HAS) divides a video file into a number of non-overlapping segments (or chunks), at a multiplicity of resolutions and bit rate encodings. These segments are hosted by a HTTP server and can be requested by clients through HTTP `GET` requests. Along with media segments, the server keeps a manifest file that describes the different video profiles, server IPs, and URLs to download segments for each profile; figure 3 describes a typical HAS system. When a streaming session starts, the client makes a request to the HTTP server and is sent the *manifest* file. Using the information in the manifest, the player starts requesting video segments and keeps them in the *playout* buffer. Typically, the client keeps a few segments in this buffer to avoid video stalls and rebuffering events that may result from short network transients. In the course of playing the video, the client continuously assesses available bandwidth and requests successive segments for the bitrates

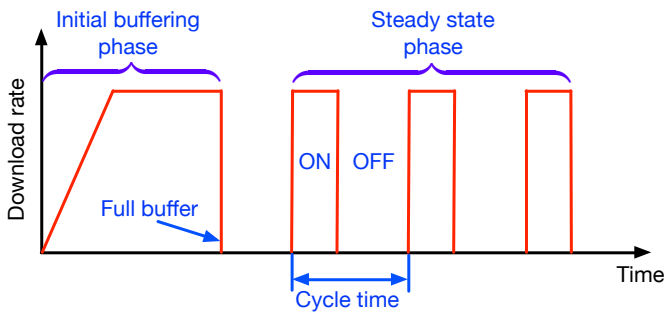


Figure 4: DASH player behavior

that can be supported. It is important to mention here that all the playback and adaptation logic resides in the client while the video server is a simple HTTP server.

ON-OFF Behavior. Video streaming starts with a buffering phase where the player downloads video as fast as possible to fill the playout buffer. Once this buffer is full, the player enters a steady state phase where it periodically downloads new chunks as needed [34]. In the steady state, the player is in the *ON* state when it is downloading a segment, and in the *OFF* state otherwise. Thus, we see a alternating of *ON* and *OFF* states during video playback; figure 4 illustrates typical behavior of a HAS client.

Cycle Time. The time between the start of two consecutive *ON* periods is termed *cycle* time. Since the player periodically downloads new segments of video to keep its buffer at a fixed level, the *cycle* time is a good estimate of the length of the video segment in seconds assuming all segments have the same length (which is the common case).

MPEG-DASH (Dynamic Adaptive Streaming over HTTP) [10] is a standard that defines media representation at the server. The standard defines guidelines for media segmentation and a collection of standard XML formats for the manifest file. Client implementation and adaptation technique, however, is not part of the standard. Although several commercial streaming services implement HTTP adaptive streaming (e.g. Netflix and YouTube), they all implement their own proprietary techniques both for media representation and for client adaptation.

1.2 Challenges of HTTP Adaptive Streaming

In this section we describe some of the problems that emerge from the the interaction between the current Internet architecture and adaptive HTTP streaming. These problems can potentially affect the quality of the user viewing experience in addition to affecting the quality of other non-video streaming applications that share bandwidth with video. In this thesis we focus on the following specific challenges:

1. **Characterization of adaptive streaming client behavior.** Several previous works have studied the behavior of adaptive streaming clients on the PC platform [101]. Adaptive streaming on mobile devices, however, was not investigated in details in the literature. Different mobile operating systems could provide different levels of support for adaptive streaming which could dictate certain implementation limitations on streaming application developers. Moreover, we need to understand whether the dynamics of mobile video players are different than the PC players. More specifically, we are interested to learn about the creation and termination pattern of TCP connections. Another interesting question is how efficient players are able to exploit the available bandwidth and how this affects behavior of different clients when they compete for bandwidth.

2. **Fair resource sharing for adaptive video streams.** Previous studies [32] have shown that when multiple adaptive streaming clients compete for bandwidth they may suffer from: i) bitrate instability, i.e. unnecessarily switching video bitrates, ii) unfairness, i.e. clients get unequal shares of the available bandwidth, and iii) under-utilization of the shared link. This happens mainly because the shift in synchronization between the ON/OFF periods of several video players can cause them to over or under estimate the available bandwidth. As a result, players keep switching between different bitrates over the course of the streaming session. Fairness in previous work [32, 40, 66] was mainly defined as receiving equal video bitrates for all the clients. This may not be the proper fairness metric for adaptive streaming mainly because different devices have different resolutions and screen sizes which may affect

the QoE of the same video profile on different devices. Defining and implementing QoE-based fairness of adaptive video flows could be of great importance specially when the available bandwidth is short of supporting the highest bitrate for all clients.

3. **Buffer bloat resulting from adaptive video streams.** Excessive buffering of network devices on the Internet is a well known problem which has been studied in different contexts [35, 91]. This problem was reintroduced recently by Gettys and Nichols in [54] under the name *buffer bloat*. Buffer bloat results from the interaction between bursty TCP traffic and large buffers on the Internet. It can introduce significantly large queueing delays in the middle boxes which can seriously harm time-sensitive applications (e.g. VoIP) whose traffic goes through the same boxes. We are interested in investigating whether DASH traffic can cause buffer bloat and further, how to mitigate this problem.

4. **Hybrid CDN/P2P architecture to reduce network bandwidth.** ISPs and video streaming providers usually have peering links with certain agreements about the amount of video traffic that should flow into ISP networks. Recent reports [83,85] reveal multiple disputes between ISPs and video streaming providers (Netflix and YouTube) due to the amount of traffic injected by these services into ISP networks. These disputes raise alarms whether these peering links could become the system bottleneck and cause serious degradation to the quality of video streaming. Although hybrid CDN/P2P systems where mainly proposed to increase CDN scalability [57,59], they can also be used to reduce the load on the peering links between ISPs and video providers. More specifically, clients' upload bandwidth can be used to redistribute parts of the video to other peers watching the same video stream.

1.3 Summary of research objectives

In this thesis we investigate a set of problems associated with the operation of HTTP adaptive streaming on the Internet. We summarize these problems below.

Characterizing client behavior of commercial mobile video streaming services. Before embarking on developing solutions for improving adaptive video streaming, we need to get a better understanding of adaptive streaming on mobile devices. We perform a detailed study of the three dominant video streaming services (Netflix, YouTube and Hulu), and characterize their behavior on the two most popular mobile platforms (iOS and Android) across WiFi and cellular 3G networks. In particular, our study seeks to answer three main questions: (i) Do service providers make consistent design choices across mobile platforms and network types? (ii) Are video players *network efficient*?, and (iii) Do players for the same service provide consistent playback quality across different platforms? and how do they interact with each other? We find that video player implementations vary significantly across mobile platforms in a way that can potentially cause many inefficiencies specially when multiple clients compete for bandwidth.

QoE max-min fairness for adaptive video streaming. Potential unfairness between adaptive video flows when they compete for bandwidth is a well recognized problem. We focus on the fairness problem which can exist in the last access hop when access bandwidth is low (e.g. at home) or even in other parts of the video distribution network (e.g. peering links between ISPs and video providers). We extend the definition of bitrate fairness to video Quality of Experience (QoE) fairness then we define QoE max-min fairness for a set of adaptive vide flows sharing a network. In addition, we develop an algorithm to compute the set of bitrates (if one exists) that should be allocated to users to achieve max-min fairness. Furthermore, we design and implement a system that runs on home routers and enforces QoE fairness on active video streaming sessions.

A client based technique for mitigating the buffer bloat effect of adaptive video flows. Since the behavior of a DASH player can be simply described as a periodic file

download (video segments) over HTTP (and in turn TCP). We ask the question: does DASH traffic cause buffer bloat? and if yes, how can we mitigate this problem? We use testbed measurements to show that, indeed, HTTP adaptive streaming can be harmful to other applications sharing the same residential network. Our results show that even a single video stream can cause up to one second of queuing delay and it even gets worse when the home link is congested. In addition, we introduce SABRE, a client based technique that can be implemented in the video player to mitigate this problem. We implemented SABRE in the VLC-DASH player. Using testbed experiments, we show that SABRE manages to significantly reduce queuing delays while not affecting the user viewing experience.

Analysis of adaptive streaming for hybrid CDN/P2P live video systems. Hybrid CDN/P2P video streaming systems can help solve two problems: increase CDN scalability and reduce traffic load on the peering links between ISPs and video streaming providers. Existing hybrid systems are mostly used to stream live video and this is why in this work, unlike previous parts, we consider live video streaming. We investigate the operation of live adaptive streaming in a hybrid system with the objective of developing operational guidelines for the CDN to optimize clients streaming experience. More specifically, we ask the following questions: how can the system decide when to switch between the CDN and the P2P modes while efficiently utilizing both server and peer upload capacity? and, in case the system is overloaded, that is the best bitrate adaptation strategy which specifies how different bitrates are allocated to different users while maximizing the overall user satisfaction? We develop mathematical analysis of the system to answer these two questions and we use simulations to validate our results .

1.4 Thesis organization

In chapter 2 we review some of the related work in different areas related to the work in this thesis. Chapter 3 presents a characterization study of commercial adaptive streaming services for mobile devices. In chapter 4 we define QoE max-min fairness for a set of competing video flows then we present the design, implementation, and evaluation of a

system that applies QoE fairness in home networks. A client based approach for mitigating the buffer bloat effect of adaptive video flows is presented in chapter 5. Chapter 6 presents our work on analyzing adaptive streaming for a hybrid CDN/P2P live video system. We summarize the contributions of the thesis and discuss some future work in chapter 7.

CHAPTER II

RELATED WORK

For many years video streaming on the Internet has been very challenging. This is mainly because the Internet was built to provide a best effort service and did not provide any Quality of Service (QoS) guarantees. Furthermore, video distribution to a large number of clients in a scalable and efficient manner is very challenging specially with the heterogeneity in client access speeds. In this chapter we review some of the research that was done to address these problems. More specific related work to each chapter will be discussed therein.

2.1 RTP and video multicasting

RTP [14] was proposed by the Internet Engineering Task Force (IETF) as a standard protocol for streaming real time data such as audio and video. A lot of research was done to augment RTP with application layer congestion control techniques. Such techniques usually took the form of rate control [45]. The idea is to match the rate of the video stream to the available bandwidth. Since video traffic coexists with normal TCP traffic on the Internet, some TCP-friendly congestion control techniques have been proposed for audio and video transmission [50].

Extensive research has been done over the years on using multicast for video streaming [124]. The difficulty with multicast is that it is very difficult to find a stream rate that is acceptable by a large number of receivers with different hardware capabilities and network resources. Two general approaches were proposed by researchers to solve this issue. In the first one, the Internet was assumed to provide QoS capabilities in terms of reserving resources and maintaining bounds on delay, jitter and loss [23, 94]. The second approach used adaptive bitrate techniques to adjust the stream bitrate to the available bandwidth [75, 104, 120]. These adaptive techniques in general use feed back loops to let clients give feedback to the server and the server can then act accordingly.

Multicast based adaptive bitrate techniques can be classified into three main categories;

single stream approaches, replicated stream approaches, and layered stream approaches [114]. In the single stream approach, a single video stream is transmitted to the multicast group and feedback is received from all clients participating in the group [63, 64]. In order to deal with the heterogeneity in clients network speeds, the replicated stream approach was proposed. In this approach, the same video is replicated in multiple streams; each with a different bitrate and different quality. In addition, each stream has a different multicast address and a client can join the stream that fits its capability [103, 119]. In layered stream approaches, on the other hand, the source sends the video stream in multiple layers with each layer in a different multicast group. Each client can then subscribe to a subset of layers that fits its processing power and network speed [75, 104, 120, 121].

2.2 Peer-to-Peer video streaming

Peer-to-Peer (P2P) networks have recently emerged as a new paradigm to allow Internet users to share content without the need for centralized servers [2, 4]. Due to the popularity and scalability of P2P networks, many video streaming systems adopted P2P networks as their platform to deliver live and VoD video on the Internet. Some P2P video systems have recently succeeded in attracting significant numbers of users [58, 59, 122, 125]. P2PLive [58] is one of the most famous P2P TV systems in China where users can play tens of live video channels and hundreds of on-demand movies.

P2P video systems can be generally classified into two categories based on the structure of their overlay network; *tree-based* and *mesh-based* [127]. In tree-based systems, peers are organized in a tree structure and video is usually pushed from the root to subsequent levels of the tree until it reaches the leafs [44,65,81,125,126]. Although this model is attractive due to its simplicity and mathematical tractability, one major disadvantage of tree-based systems is that they are severly affected by peer churn. In mesh-based systems, on the other hand, peers usually connect a random set of peers who watch the same content [68, 82, 87, 122]. Neighboring peers usually exchange information about their data availability and then they pull data from neighbors when it is ready. Since each client maintains a set of peers at any point in time, this model is much more robust to peer churn than the tree-based model.

Multiple adaptive streaming techniques have been proposed in P2P streaming systems. For example, the work in [99] uses layered video encoding to adaptively deliver different layers of the video to clients. Multiple description Coding (MDC) has also been used to propose adaptive streaming systems that support a large number of users in [47, 117]. Another approach was used in [89] where network coding is used to make SVC more feasible in adaptive streaming.

2.3 HTTP video streaming

Over the years, there has been a clear transition in video streaming from classic protocols (e.g. RTP) to the TCP-based HTTP streaming. This happened for many reasons including the following: HTTP streaming uses traditional web servers which are less expensive than streaming servers, this in turn came in favor of CDNs which are widely used to distribute video in a scalable manner. Also, using HTTP streaming makes all web caching schemes applicable since a video stream can be considered like any HTTP object. Furthermore, it is much easier to handle HTTP traffic behind NATs and firewalls [24].

One of the common models of HTTP streaming is *progressive file download*. In this approach, video streaming simply happens through a HTTP file download from a web server. The player allows the video file to be played before it finishes downloading it [25]. In addition, byte-range requests are used to implement seeking in the video. One major drawback of this approach is that different clients with different capabilities and different network speeds receive the same video quality. This led to the development of HTTP adaptive video streaming. Microsoft's Smooth streaming [20], Adobe's Dynamic Streaming [30], and Apple's HTTP Live streaming [95] use this approach.

The work in [112] provides the foundation for the MPEG-DASH standard. A DASH dataset of multiple videos with different resolutions and bitrates was provided in [76] to help the research community conduct experiments on DASH. In addition, several open source DASH players were developed to help evaluate new client adaptation techniques. Among these players are the VLC DASH plug-in [86] that was developed as an extension to the VLC player [16], and DASHJS; a JavaScript player [3]. No reliable open-source DASH

players exist for mobile devices (smart phones and tablet) yet.

An experimental study to evaluate the adaptation algorithms of multiple video players was presented in [101]. The same authors studied the interaction between multiple adaptive video players sharing the same bottleneck link in [102]. This work proposed three metrics that can be used to evaluate the performance of adaptive streaming players when they compete for bandwidth. These metrics are: *instability*, *unfairness*, and *utilization*. Instability is defined as the fraction of time the player switches between different video bitrates, while unfairness is defined as the difference between bitrates played by different players. Recently, some work was done both at the client and in the network to improve the performance of DASH players when they compete for bandwidth. New player adaptation techniques were proposed in [53,66] to improve DASH players in the simple scenario of users sharing a home link. The competition between adaptive video flows in cellular base stations was considered in [40]. In this work, the problem of computing fair video bitrates for the competing flows was modeled as linear optimization problem, then a scheduler at the cellular base station was designed to enforce these bitrates on clients. A server-based traffic shaping technique was developed in [33] to reduce video bitrate instability for DASH clients.

2.3.1 Measurement studies

Several measurement studies have been conducted to characterize video streaming traffic. A characterization study of YouTube traffic was done in [56] where they collect traces of $600,000$ streaming sessions in a campus network. In this work they characterize the transfer behavior of YouTube and compare it to other Web workload characteristics. In [39], the authors study YouTube videos' lifecycle and suggest new caching techniques that can significantly reduce load on video servers. The work in [128] studies YouTube traffic in a university campus network where they analyze the bitrates, durations, popularity, and access patterns of streamed videos. In [49], the authors investigate the differences in YouTube traffic patterns and users' behavior between mobile and PC users.

Other studies have focused on understanding the operation of popular commercial video

streaming services. One direction has focused on reverse engineering the network architecture of the service as a whole. In [26], the authors reveal multiple DNS namespaces and 3-tier cache hierarchy used by YouTube. The particular CDN selection and redirection mechanisms were examined in [27, 28, 116] for YouTube, Netflix, and Hulu (respectively). The work in [29] investigates load balancing techniques used by YouTube to distribute users' requests among its data centers. Yet another direction has focused on understanding behavior closer to the client [60, 62, 80, 98]. In [60], the authors show that bulk file download can significantly hurt the quality of a video stream when both are competing for bandwidth. The work in [62, 80] investigate the effect of memory limitations in smart phones on video streaming players. They both show that memory limitations can trigger undesired behaviors that cause video players to download unneeded data.

2.4 Hybrid CDN/P2P systems

Recently, some work has proposed hybrid streaming systems that combine CDNs and P2P technology [57,59]. These systems promise to achieve the scalability of P2P networks and the desired low delay and high throughput of CDNs. In a hybrid CDN/P2P system, a client can receive video from either video servers or from other peers viewing the same video. LiveSky [57] is an operational commercial live streaming system with more than ten million users that adopts the hybrid CDN-P2P approach. Using a nine-month trace from the MSN video service, the work in [59] shows that a hybrid CDN/P2P system could significantly reduce server bandwidth costs. In addition, some work has been done in developing mathematical models to analyse hybrid streaming systems [78,123], however, none of them studies adaptive streaming. Although hybrid systems have a significant potential for providing an attractive video streaming scheme, adaptive streaming has not been extensively explored in such systems.

CHAPTER III

CHARACTERIZING CLIENT BEHAVIOR OF COMMERCIAL MOBILE VIDEO STREAMING SERVICES

3.1 Introduction

Given the enormous traffic volumes involved in video streaming, there has been a significant amount of interest in the community in characterizing how these systems operate. In chapter 2, we reviewed some of the work done on studying traffic characteristics, CDN architectures, and client behaviors of several video streaming services. Despite these studies, there is still a lack of understanding about the impact of heterogeneity across players and mobile platforms. In particular, the player implementations on different platforms vary considerably and make varied design choices; these result from variations in the underlying APIs and operating system support. These variations may influence the network behavior of the video players.

In this chapter, we perform a detailed study of the three dominant video streaming services (Netflix, YouTube and Hulu), and characterize their behavior on the two most popular mobile platforms (iOS and Android) across WiFi and cellular 3G networks. In particular, our study seeks to answer three main questions: (i) Do service providers make consistent design choices across mobile platforms and network types? (ii) Are video players *network efficient*?, and (iii) Do players for the same service provide consistent playback quality across different platforms? and how do they interact with each other?

Addressing the first question enables us to highlight the primary differences in the design choices made across the different player implementations. Previous work has hinted at differences in network behavior of a service across platforms [62, 79] but has not explored this at depth.

The second question is of particular interest given the recent tussles between Netflix

and several prominent ISPs about who should bear the cost of increased traffic on peering links [83, 85]. Even small network inefficiencies could have a significant impact given the large user base (Netflix currently has 27 million subscribers). The third question relates to network bandwidth inequities when different implementations of the same service share bandwidth resources. As tablets and smartphones become ubiquitous, we can easily foresee households commonly possessing multiple mobile video devices, perhaps of different platforms, and we wish to understand how these may interact together.

The main contributions and findings in this chapter are summarized as follows:

- We find that video player implementations vary significantly across mobile platforms and network type which can potentially impact the caching and distribution mechanisms of the CDN.

- We find that mobile video streaming applications are typically designed to maximize user experience, but this comes at the cost of network inefficiencies caused by players downloading redundant data (up to 25% in our observed traces) on both WiFi and 3G. This unused, downloaded data has a negative impact on ISPs and also users with capped broadband data service .

- We find that, over a shared wireless network, the Android Netflix player client consistently carves out a larger share of the network bandwidth than the iOS version, and consequently has higher perceived video quality to the end-user. We present this in detail and explain this as a side-effect of specific implementation choices and API limitations.

In section 3.2 we describe our experimental setup and the collected dataset. Our findings about video adaptation and client download patterns are described in sections 3.3. We define download *redundancy* in section 3.4 and present some experimental results to show its significance. An experimental study of video performance when multiple clients share bandwidth is presented in section 3.5 then we summarize the chapter in section 4.7.

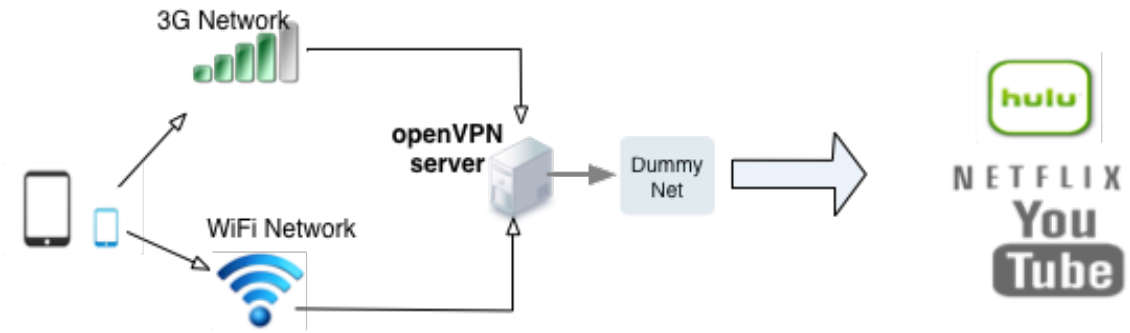


Figure 5: Experimental setup

3.2 Experimental Setup

There are three challenges in designing an experimental setup to study the dynamics of video streaming and observing client behavior in mobile devices. First, we require complete packet traces to accurately determine the ON-OFF periods and cycle intervals. Second, we need to collect these on WiFi *and* 3G networks. And finally, we require fine-grained control over the bandwidth seen by the client to study how different players adapt to network changes.

Typical traffic logging tools (tcpdump, wireshark) are not feasible on resource constrained mobile devices. While we could redirect traffic through a proxy server, these are not easy to configure for 3G interfaces. To address these limitations, we configure a VPN through which all traffic to and from the mobile devices passes (and where it is logged); the setup is depicted in Fig. 26. In order to support experiments (in §4,§5) which require restricting bandwidth, we use `DummyNet` on the VPN server which lets us shape the download bandwidth of the clients. To ensure that the VPN server itself does not affect playback, we use a high performance host with ample Internet bandwidth. One drawback of our VPN based setup is that we sacrifice geographic diversity, and cannot really study CDN dynamics effectively. However this aspect has been addressed in prior work and our own work focuses on understanding client behavior across platforms during playback.

Devices and Dataset: Table 6 lists the four devices used in our study. All of them are used while streaming over WiFi while 3G traces are collected only for the iPad and the Nexus 7 tablet. We select 10 videos from each of the three providers studied which have different

Table 1: Mobile devices used for trace collection

Device	OS	Screen Resolution	Interface(s) used
iPhone 4S	iOS 7	960×640	WiFi
iPad 2	iOS 7	1024×768	WiFi/3G
Nexus 7	Android 4.2	1280×800	WiFi/3G
Nexus 10	Android 4.2	2560×1600	WiFi

popularity ratings. Note that since there is almost no overlap in the content catalogs, we use a different set of videos for each provider. Also, we make sure to select videos that are longer than 15 minutes; the viewing time varies depending on experiment. Note that Youtube relies on progressive streaming for clips shorter than 15 mins, so we ensure that the videos selected are adaptively streamed. Our results did not show any differences in client behavior for videos of different popularity. Before running any experiments that involve constraining bandwidth, we use a Speedtest app to measure bandwidth from the device to the Internet (through the VPN). We observed a download bandwidth (latency) of $15Mbps$ ($20ms$) for WiFi and $4Mbps$ ($70ms$) on 3G.

3.3 Streaming behavior

We first take a detailed look at the traffic patterns generated by each client implementation on particular networks to understand how the platform and device affect how the client downloads the required data. This was gathered by carrying out repeated experiments (different videos, each viewed for 5 minutes), and extracting client behavior as previously described. Table 2 presents a high level summary of the findings and we discuss each of these in the rest of the section. Note that some clients are not adaptive on Android (indicated with * in Tab. 2): in these cases, when multiple bitrates are available, they have to be explicitly selected.

3.3.1 OS Support For Adaptive Streaming

Both iOS and Android provide different levels of platform support for adaptive streaming. Apple iOS provides a native API for its own adaptive streaming variant called HTTP Live Streaming (HLS) [6]. As a result, on iOS, all three services we study use HLS as the

Table 2: Streaming Player Characterization: persistent TCP connection indicates a single open TCP connection was used for a given bitrate; switching to a new bitrate starts a new connection.

<table>
<tr><th colspan="2" rowspan="2"></th><th colspan="3">Apple iOS</th><th colspan="3">Android</th></tr>
<tr><th>YouTube</th><th>Netflix</th><th>Hulu</th><th>YouTube*</th><th>Netflix</th><th>Hulu*</th></tr>
<tr><td rowspan="2">ON/OFF</td><td>WiFi</td><td colspan="3" rowspan="2">New HTTP request for each segment</td><td>same as iOS</td><td colspan="2" rowspan="2">rwnd based flow cntrl</td></tr>
<tr><td>3G</td><td>N/A</td></tr>
<tr><td rowspan="2">Segment size</td><td>WiFi</td><td rowspan="2">5 sec</td><td rowspan="2">10 sec</td><td rowspan="2">10 sec</td><td>50 sec</td><td rowspan="2">10 sec</td><td>150 sec</td></tr>
<tr><td>3G</td><td>N.A</td><td>350 sec</td></tr>
<tr><td rowspan="2">TCP conns.</td><td>WiFi</td><td rowspan="2">1/ HTTP req</td><td colspan="2">Persistent</td><td>1 /request</td><td colspan="2">Persistent</td></tr>
<tr><td>3G</td><td colspan="2">1/ few HTTP req.</td><td colspan="3">Persistent</td></tr>
</table>

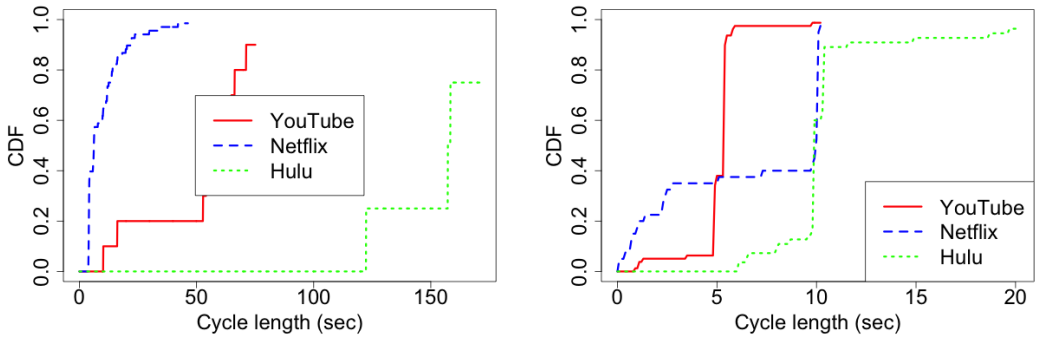
protocol. However, there are still variations in protocol parameters and design choices across these. On the other hand, the Android SDK does not provide uniform adaptive streaming support across all devices and versions making it difficult for programmers to rely on it. Consequently, developers make use of third-party libraries or develop their own streaming protocols.

Implications. This varying (or missing) platform support for adaptive streaming leads to potential differences in client behavior (both on the same platform and across platforms). In addition, it makes it difficult for online service providers to have a single video delivery infrastructure (affecting cost), and to ensure consistent behavior of their players across different platforms.

3.3.2 Adaptation and Traffic Shaping

HLS on iOS: HLS requires a video stream file to be split into multiple equal length video segments. Information about the mapping between the available network bandwidth and corresponding video bitrate to use along with information about the URL to download the particular segment is encoded in manifest files that are downloaded by the player.

Figure 6b plots the distribution of cycle lengths for iOS across the three services. We observe that Netflix and Hulu typically use cycle lengths of 10s (which is the HLS recommendation) while YouTube uses shorter cycle lengths of 5s or 5.4s. The cycle lengths were inferred directly from our methodology and posteriori verified against ground truth. While manifest files for Hulu and Netflix are encrypted, we believe that the results hold since much of the work is done by the common API. Moreover, in the case of YouTube, the manifest is



(a) Android (b) iOS

Figure 6: CDF of *cycle* period for Netflix, Hulu and YouTube on Android and iOS using WiFi interface

in the clear and we are able to verify the results.

Android: The lack of native support for adaptive streaming on Android leads to very different implementations across the three applications. Surprisingly, we observe that both Hulu and YouTube do not support adaptive streaming and require the user to *manually select* the streaming rate. The YouTube player provides a toggle to enable/disable HD streaming while Hulu enables users to choose between four quality levels: low, medium, high, and HD. Absent any automatic adaptation, both Hulu and YouTube aggressively download video segments to fill up the large playback buffers. As shown in Figure 6a the cycle intervals for Hulu and YouTube are much larger compared to HLS implementations on iOS (Figure 6b). The cycle period for Hulu is larger than 100s and the average cycle time for YouTube is 50s over WiFi.

In contrast, Netflix on Android implements its own adaptation mechanism. When the player starts a video it sends an HTTP `GET` request with an open ended range from the starting point until the end of the video. The player controls the download pace using TCPs flow control mechanism by setting the TCP receiver window *rwnd* to zero. This signals the server to stop sending data to the player until the time *rwnd* becomes non-zero. It is important to observe that the player's ON/OFF behavior is controlled by the value of *rwnd* as compared to having separate HTTP `GET` requests in iOS players. Interestingly, as seen in Figure 6a, the parameters chosen on Android lead to cycle times roughly similar to

that in the HLS based iOS application.

Implications: The different adaptation and traffic shaping mechanisms employed by the three HLS implementations lead to highly varying and potentially competing traffic patterns. Recent work [32] has shown that such patterns can potentially lead to unfairness and adaptation instability. On Android we observe that the lack of streaming support could potentially lead to a degraded quality of experience. Furthermore, the large playback buffers used by Hulu and YouTube on Android could lead to significant amount of wasted bandwidth if the user chooses to abandon the video without playing the content already fetched into the buffer.

3.3.3 TCP Connection Overhead

We now characterize the number of TCP connections used by the player to download the video segments at the same or different bitrate. This connection behavior impacts how well the client can estimate available bandwith (many short connections lead to underestimation), and also the granularity at which a hosting CDN can load balance incoming segment requests.

Across the diversity of players, platforms and networks, we observe two primary connection patterns: (i) individual TCP connections for *each* segment, and (ii) a long single TCP connection for *all* segments of the *same rate*. Interestingly, we find that players do not adopt the same strategy consistently across mobile platforms and network types. For example, as shown in Table 2, YouTube on Android/3G uses a single persistent TCP connection to download multiple segments, while in every other setting, it opens separate TCP connections for each segment. We now describe in detail the two patterns.

Single TCP connection for all segments of the same rate. Over a WiFi network, Hulu reuses the same TCP connection to download all segments *from the same bitrate*. This behavior is consistent across both Android and iOS. In a scenario where the mobile device has high access bandwidth, the player quickly converges to the best achievable video bitrate. As a result, the player can use only a small number of TCP connections; in our experiments Hulu and Netflix players established on average less than 5 connections during

the few minutes of video playback. This behavior is also consistent for Netflix on Android that does not use the segment based adaptation. At the point at which the player decides to switch bitrates, the player terminates the TCP connection, starts a new one, and issues another HTTP `GET` request again with an open ended range from the current position until the end of the video.

Single TCP connection for each segment. The YouTube application for iOS initiates a new TCP connection for requesting every segment over both WiFi and cellular networks. For every segment request, the application requires at least one and sometimes two HTTP redirection requests; the client requests a video segment form the server and gets a `302 Found` reply. The client gets the new server address from the `Location` HTTP header and then does another HTTP `GET` request to the new server. For a 200 seconds streaming session, the YouTube player downloads on average 50 video segments which uses 100 HTTP `GET` requests over 100 different TCP connections.

We observed another variation to the above described connection pattern for the Hulu and Netflix iOS players over 3G. Sometimes, the player downloaded more than one segment over the same TCP connection. However, this behavior was infrequent and only limited to this scenario.

Implications: The above described TCP connection request patterns impact the CDN caching and distribution mechanisms as well as network overhead over slow cellular links. The YouTube application on iOS potentially enables fine-grained load balancing and content placement strategies across the CDN as each segment request could potentially be re-directed to a new video server. Frequent TCP connections over cellular networks inherently support mobility but incur overheads on cellular networks. High round trip times and TCP slow start dynamics could potentially add significant delay in fetching each segment. Finally, when individual TCP connections are used in conjunction with short segments on high bandwidth links, the player might not correctly estimate the available (steady) bandwidth, which might cause instability in the bit rate selection.

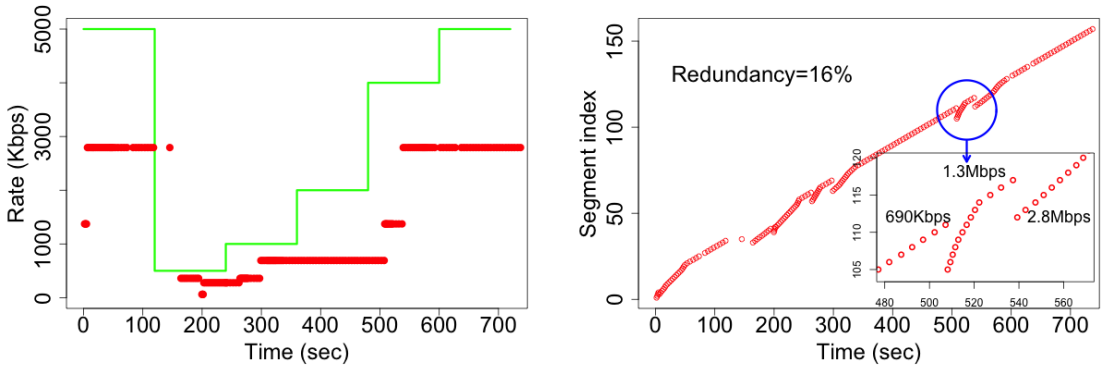
3.4 Traffic Redundancy

Our examination of the play time of segments revealed that during some bitrate shifts, players occasionally fetch previously downloaded segments (but at a higher bitrate). We are able to identify this behavior since we associate each segment with its unique index as well as bitrate (and play time). Clearly, some of these downloaded segments (typically the lower bitrates) are discarded by the player and correspond to wasted, or *redundant* traffic.

Previous studies have alluded to wasted traffic [62, 79], and related the wastage with players' early exiting (before finishing the video), or with memory pressure (forcing valid segments to be discarded) or with closing active TCP connections (losing all data in transit). In contrast, our traces suggest very strongly that the redundant traffic is caused by the players attempting to improve the bit-rates being used. In order to get a more systematic understanding of this, we conducted several experiments where the bottleneck bandwidth was manually controlled and changed (forcing bitrate adaptation in the clients). With this, we can quantify the extent to which players download repeated segments when switching between profiles, and how much this varies across players. Due to a lack of space, we only present details from a single experiment setting – for the YouTube player (since this gives complete ground truth about the segment play times and bitrates).

The experiment uses a bandwidth profile shown in Fig. 7a (in green). Starting initially with $5Mbps$, we force the following set of transitions that are spaced 2 minutes apart: $0.5 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ (Mbps). We repeat the experiment for all three players on both platforms and record the bitrates of the downloaded segments and the play time offset of the segment (as described previously). To form a quantitative comparison across players, we use *redundancy* as a metric; this is defined as the ratio of traffic (in bytes) associated with redundant segments to the total number of bytes over all the downloaded *non-redundant* segments i.e., bytes that are actually rendered by the player.

Fig. 7a presents a detailed trace of a 750s YouTube video on an iPad streamed over WiFi. The x-axis represents time, while the y-axis shows the inferred bitrate. We clearly see the player adapt to the bandwidth envelope that we enforce, changing the requested



(a) Bitrate adaptation

(b) Downloaded segment index

Figure 7: YouTube adaptation on iPad using WiFi

video bitrate among 5 different profiles ranging between $150Kbps$ and $2.8Mbps$. The companion figure 7b plots the index of the requested segment (on the y-axis) over time. Ideally, if no traffic was redundant, we would expect a monotonic pattern caused by incrementally increasing segment indices. However, as is visible in the figure, each bit-rate shift is associated with the player (re-)downloading previously downloaded segments of a higher bitrate. The detailed inset plot clearly indicates, at about time=510, the player starts to download segments encoded at 1.3Mbps for the same video segments that were previously downloaded at 690Kbps; this is also seen at time=520 for the next higher bitrate. We posit that this behavior is caused by the player trying to maximize quality by upshifting bitrates when it has enough playback headroom to download higher bitrate segments in time. We were also able to confirm that YouTube uses closed GOP switching which eliminates the possibility that this redundancy could happen due to inter-segment synchronization.

This behavior was observed for Hulu, Netflix, and YouTube on both WiFi and 3G for this and several other bandwidth envelope settings. For the envelope used in Fig. 7, we measure the redundancy to be 21%, 22.5%, and 16% for the three services respectively when WiFi is used.

Implications: The levels of redundancy that we observe have the following three significant implications. First, it may result in higher transit costs (or more unbalanced peering) for ISPs. For example, Georgia Tech has about 12,000 students that live in campus dorms,

and even if 10% of them (a conservative estimate) watch video at an average bitrate of $1Mbps$ with 10% redundancy, this roughly translates into $100Mbps$ of wasted traffic which is added to the transit cost of the campus network paid to its providers. Second, this wasted traffic is detrimental to end-users who subscribe to capped broadband plans. Third, the large amounts of redundant traffic increase the network contention in shared, resource constrained networks such as 3G and directly lead to network quality degradation for users.

It is important, however, to mention that the high redundancy ($10\% - 20\%$) we get from iOS video players is mainly because of the design of our controlled experiments. In real settings, the bandwidth envelope may not change as drastically as it does in our experiments. In order to measure redundancy in real large scale networks, we need to collect streaming traces from large networks (e.g. campus network or ISP) and observe streaming behavior in these traces. This is considered as future work.

3.5 Player Bandwidth Exploitation

Previous work has shown that when multiple, identical HTTP streaming clients compete for bandwidth (on wired networks), the phase synchronization in ON-OFF periods can cause clients to incorrectly estimate bandwidth over time leading to bitrate instabilities in playback [32]. In this section, we examine interactions between *heterogeneous player implementations* of the same service and on wireless networks which generally exhibit more network dynamics (than in the wired setting). Since the applications are implemented differently on different platforms, we expect differences in player performance and video quality due to the underlying bandwidth estimation and chunk fetching behaviors. We focus the study on Netflix as it is the only service that supports adaptive streaming on Android and iOS. We find that the Android client is able to use more available bandwidth than the iOS client. More surprisingly, we find this holds when both players simultaneously contend for shared bandwidth.

To understand how each player uses the available bandwidth, we first investigate how much bandwidth each player needs in order to attain a particular bitrate and we ask: *is this value different for each of the clients?* To answer this, we carry out a set of experiments

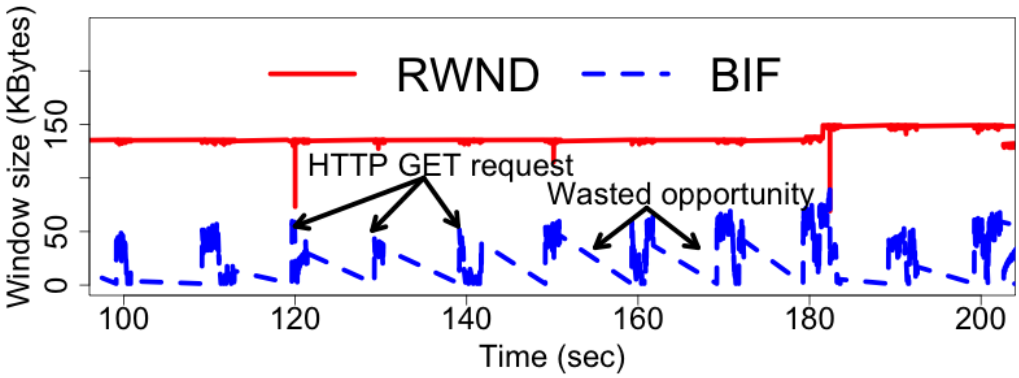
Table 3: Minimum Bandwidth required to get a video bitrate by Netflix on iOS and Android

	Video bitrate (kbps)			
	560	750	1050	1750
iOS	1500	2300	2600	4500
Android	850	1100	1500	2600

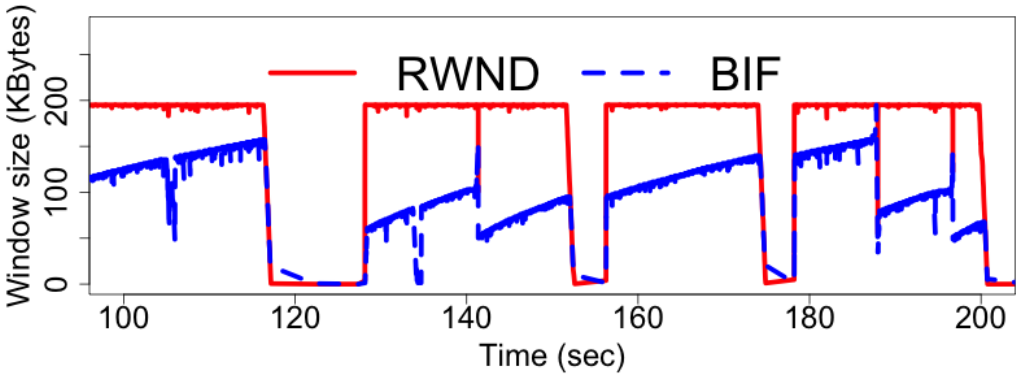
(individually, for each player) where we stream a videos with the download capacity at the VPN server fixed at a particular value. We repeat the experiment for bandwidth values between 1Mbps to 5Mbps with 0.5Mpbs increments.[1] At the end of each experiment, we record the bitrate achieved by the client (we did not observe any bitrate fluctuations after the initial buffering phase). The findings, summarized in Table 4, reveal that the iOS client is much more parsimonious in its bandwidth usage. In order to use a particular bitrate, the iOS client requires available bandwidth of about 2.6 times the bitrate; in contrast this number for Android is 1.5. The takeaway here is that for a given bottleneck bandwidth, the Android player has higher quality (larger bitrate corresponds to more quality, all other things being equal). To understand exactly how the clients' individual behaviors leads to this quality difference, we examined the detailed low level TCP dynamics during video playback. Figure 8 plots the TCP receiver window (*rwnd*) and the number of bytes-in-flight (BIF) over time, for iOS and Android clients (the bottleneck bandwidth was set at 2Mbps).

In figure 8a each one of the repeated BIF cycles represents a HTTP `GET` request for a new video segment; a segment is fully downloaded before the next one is requested. This "serialization" causes the iOS client to miss plenty of opportunities to download more data (two of these *wasted opportunities* are highlighted in figure 8a). Stated differently, the iOS client is not using the network when the last downloaded chunk is being consumed from the buffer. In contrast, the Android client (figure 8b) does not have this problem because, as explained in section 3.3, Android uses a single HTTP `GET` to download the entire video file (relying on TCP flow control for pacing). In figure 8b we can see that over a period of 100 seconds, the Android client had only 3 OFF periods starting at times $118, 152, 175$ when

[1] To verify that there are no other bottlenecks, we also streamed video without the bandwidth limitation and always achieved *download* rates greater than 5Mbps.



(a) Apple iPad

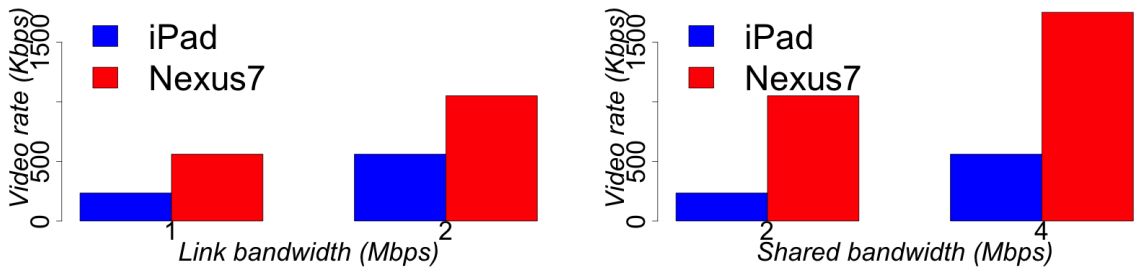


(b) Android Nexus7

Figure 8: Receiver window and bytes-in-flight (BIF) for an iPad and an Android tablet while streaming Netflix. The bottleneck link for each was $2Mbps$

rwnd goes to zero. An implication for the download opportunities wasted by iOS clients is that TCP congestion window growth is significantly slower for iOS than Android. This can be observed from figure 8b where BIF reaches over 150KBytes multiple times for Android while it is below 70KBytes for iOS most of the time. The side effect of using a single TCP connection and *rwnd* based flow control is the bytes can keep flowing as long as the client can keep consuming them.

The previous discussion raises the question of what would happen if the two player implementations contend for the same bandwidth. Would they share the bandwidth fairly? or will one client grab a much higher share of the bandwidth to the detriment of the other? To understand this better, we carried out experiments where we started streaming sessions on two players at approximately the same time where the clients share the same WiFi access point (and for different bandwidth bottleneck settings). We examine three pairs of



(a) Unshared bandwidth

(b) Shared bandwidth

Figure 9: Bitrates for Netflix clients running alone (9a) and together (9b)

interactions: (i) both clients are iOS, (ii) both clients are Android devices, and (iii) one client is iOS, the other Android. Again, all clients were for the Netflix service. Due to space limitations, we only discuss the third scenario.

Before starting the two client, shared bandwidth experiment, we obtain the baseline (stable) bitrate by running the single client (but with half of the bottleneck bandwidth used in the shared case). These baseline bitrates are reported in Fig. 9a for each client, and these match the rates reported in Table 4.

Subsequently, we double the bandwidth and start *both* clients at almost the same time. In all the experiments (each repeated multiple times), we observed both clients go through a short transient period of instability lasting less than a minute and then reach a stable bitrate which is maintained for the duration of the experiment. These stable bitrates are reported in Fig. 9b. Note that in the second experiment, the fair share bandwidth of each client is identical to the configured bandwidth in the first experiment. Thus, we expect both clients to reach exactly the same video bitrates as before; however, this is not the case! As seen in 9b, the iOS client achieves the same bitrate, but the Android client does much better: when the fair-share bandwidth is 1Mbps, it achieves a bitrate of 1050Kbps even as it attained 560kbps when it was individually given 1Mbps of bandwidth. Since we know from Tab. 4 that the Android client needs at least 1.5Mbps of bandwidth to be able to stream at 1050Kbps (0.5Mpbs more than its fair share), this seems to indicate that it takes away some bandwidth from the iOS client. The same behavior repeats when Android gets a bitrate of 1750Kbps when sharing 4Mbps with an iOS client (figure 9b), as compared

to only 1050Kbps when it has a 2Mbps bandwidth with no sharing (figure 9a). On the other hand, while the iOS client does not do worse, it is not able to leverage the additional bandwidth that might be available to reach a higher bitrate.

As stated earlier, we conducted similar experiments for two Netflix clients of the same platform; and here we summarize our observations. When two iOS clients compete for bandwidth they usually reach a stable state where they stream stable bitrates after a short period of transition. This is due to the conservative behavior of iOS clients where clients under-utilize the available bandwidth and settle for lower bitrates than what they can actually achieve. When two Android clients share bandwidth, on the other hand, they both try to utilize the available bandwidth to get a high bitrate. This aggressive behavior results in significant instability where both players keep switching between high and low bitrates. These observations clearly suggest that there is a trade-off in client design between video quality and the stability of streamed bitrates.

3.6 Summary

In this chapter we present a study of three major video streaming services on mobile platforms. By carrying out detailed packet trace analyses, we identify significant differences in the way these services currently deliver video to mobile clients. We found that only Netflix provides adaptive streaming on Android while all three services are adaptive on iOS. We demonstrate that several of the players download *redundant* data that is eventually not used for playback, most in conjunction with transient bandwidth upshifts. This redundant traffic, exceeding 15% of the total stream in some cases, imposes a significant burden on the network and has a negative impact for end-users. Given the popularity of these services, we believe addressing these inefficiencies will have a tremendous benefit.

We also identify variations in a clients (from the same service) ability to exploit the available bandwidth. In particular, we demonstrate that the Netflix iOS player always consistently operates at a lower bitrate than Netflix Android player for the same given bandwidth. Furthermore, when these two clients share bandwidth on a network link, we find that the Android client takes bandwidth away from the iOS client. These results

very strongly indicate that end-users quality of experience varies a great deal across the particular platform being used to stream the video.

As seen in this chapter, the interaction between TCP and the dynamics of video players can result in many inefficiencies with bitrate unfairness being one of them. This is expected to continue in the future because there is no standard for client implementation and video providers will continue to have their own implementations. In the next chapter we focus on the fairness issue when multiple adaptive streams compete for bandwidth.

CHAPTER IV

QOE MAX-MIN FAIRNESS FOR ADAPTIVE VIDEO STREAMING

4.1 Introduction

Previous studies [32,66] have shown that when multiple adaptive streaming clients compete for bandwidth they may suffer from: i) bitrate instability, i.e. unnecessarily switching video bitrates, ii) unfairness, i.e. clients get unequal shares of the available bandwidth, and iii)under-utilization of the shared link. In addition, recent studies [84] also show that different implementations of video players of the same service on multiple platforms may lead to unfairness when competing for bandwidth.

In this chapter we focus on the fairness problem when multiple adaptive streams compete for bandwidth. This problem can exist in the last hop when access bandwidth is low (e.g. at home) or even in other parts of the video distribution network. For example, recent reports [83,85] reveal multiple disputes between Internet Service Providers (ISPs) and video streaming providers (Netflix and YouTube) due to the amount of traffic injected by these services into ISP networks. This traffic usually travels through peering links between ISPs and streaming services which makes competition for bandwidth very likely to happen in these links. With the increasing popularity of adaptive video streaming and the massive amounts of video traffic flowing through the Internet everyday, competition between video flows is inevitable and could happen in various parts of the network.

The simple scenario of competition for bandwidth between multiple adaptive video streams at home was considered in [32, 66] while the same problem at a cellular base station was considered in [40]. In these works [32, 40, 66], fairness was defined as giving equal video bitrates to all competing clients. This definition does not take into consideration the Quality of Experience (QoE) achieved by streaming a given bitrate on different devices. For example, the QoE of streaming a low bitrate video with low resolution on a small screen device (e.g. a smart phone or a tablet) should be very different than the QoE of playing

the same video on a large screen high definition television (HDTV). This is why we believe bitrate equality is not the correct measure of fairness for video streaming specially with the heterogeneity of devices used to stream video today.

In this chapter, therefore, we first introduce a new video QoE metric that takes into account the screen size of the streaming device. Although video quality assessment is an active area of research, we have not found any metrics in the literature that take screen size into consideration. Inspired by the bandwidth *utility* concept introduced in [109] and the work in [38, 77, 108], we then define *QoE max-min fairness* for a set of video streams sharing a network where video QoE is the *utility* of bandwidth. In our model we assume a general network with capacity constraints and a set of adaptive video flows between a set of sources and destinations. We show that a QoE max-min fair bitrate allocation does not always exist because each video session has only a discrete set of bitrates that can be assigned. Alternatively, we define *QoE maximal fairness* which is easy to compute and is equal to max-min fair allocation if the latter exists. We then develop an algorithm to compute bitrate allocation in a general network that achieves QoE maximal fairness.

In order to evaluate the challenges of QoE fair allocation in a real setting, we consider the scenario of bandwidth sharing in the last access hop. We designed and implemented VHS (*Video Home Shaper*); a system that runs on home routers with OpenWrt firmware [11]. Even though VHS is designed to manage sharing on last hop links in home networks, similar methods and techniques can be used for other types of networks. VHS continuously monitors outbound HTTP requests to identify video streaming sessions initiated by clients connected to the router. For most popular streaming services, information about the streaming device and the video profile being played can be extracted from HTTP request headers. VHS is able to extract this information for Netflix and YouTube running on PC, iOS, and Android devices. We implemented our QoE fair allocation algorithm in VHS. When a new streaming session is identified or when an active session terminates the algorithm is used to identify the new set of bitrates that should be given to each of the existing video clients. VHS employs Linux *traffic control* (tc) to allocate adequate bandwidth to each client in order to enforce the desired video bitrates.

Our key contributions can be summarized in the following points:

- We develop a technique that can be used to extend existing video QoE metrics so they take the screen size and resolution into account.
- Based on our new QoE metric, we define QoE max-min fairness for multiple adaptive video streams competing for bandwidth. We show that max-min bitrate allocation does not always exist. Instead, we define maximal fairness and show that it is a good replacement of the max-min criterion. We then develop an algorithm to compute maximally fair bitrates for a set of competing video flows in a network.
- We present design, implementation, and evaluation of VHS; a system that implements QoE fairness at home. We evaluate VHS using a set of heterogeneous devices streaming video from real commercial streaming services (Netflix and YouTube).

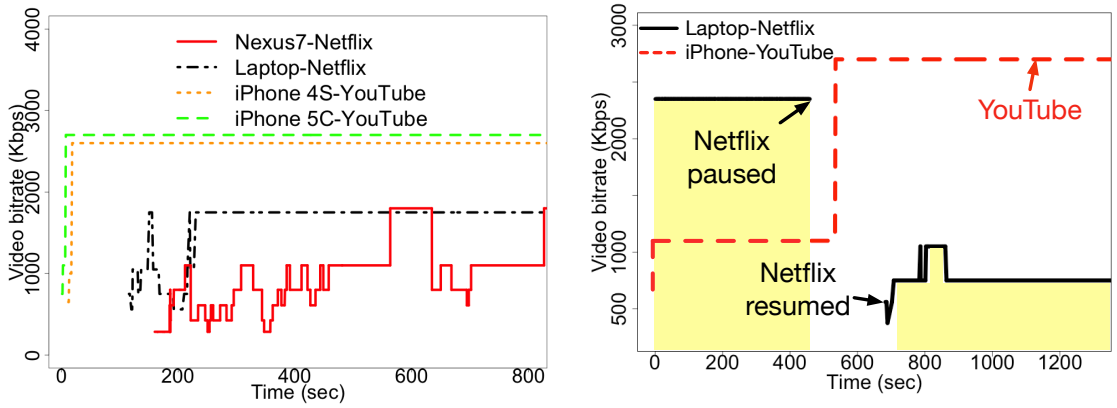
The rest of the chapter is organized as follows. In section 4.2 we briefly discuss background on adaptive HTTP streaming, video quality assessment metrics, and utility max-min fairness. We introduce our QoE metric, define QoE max-min fairness, and then develop our algorithm for computing a maximally fair allocation in section 4.3. A discussion on some implementation challenges is presented in section 4.4. In section 4.5 we introduce the design and implementation of VHS then we valuate the system in section 4.6. Section 4.7 concludes the chapter.

4.2 Background and Motivation

In this section we give a brief background on adaptive HTTP streaming fairness, video quality assessment metrics, and utility max-min fairness and review some of the related work in each of these topics.

4.2.1 Fairness in Adaptive HTTP streaming

MPEG-DASH [10] is a standard for adaptive HTTP streaming that defines media segmentation and representation at the server. Client implementation and bitrate adaptation techniques are not included in the standard which gives video streaming providers the flexibility of implementing their own clients. This, however, makes it very difficult to predict



(a) Four clients share a link of 8Mbps

(b) Two clients share a link of 4Mbps

Figure 10: Visualizing video bitrate *unfairness* when competing devices stream video from different streaming services

the performance of multiple streaming clients when they compete for bandwidth. With vendors implementing, and continuously updating, their own bitrate adaptation techniques and sometimes even implementing rate limiting techniques at the video server [55], we should not rely on video players to fairly share bandwidth.

We show an example to demonstrate how different implementations of video players of different streaming services could lead to significant unfairness in quality of the streamed video. Figure 10 shows video bitrates streamed over time for two experiments: a) two iPhone clients streaming a HD YouTube video, a Nexus7 tablet and a laptop streaming Netflix movies, all sharing a 8Mbps access link (figure 10a), and b) a HD YouTube video on iPhone and a Netflix movie on a laptop sharing a 4Mbps link (figure 10b). Looking at the two figures, one common observation is that commercial video players have managed to reduce the bitrate instability issue, except may be for Netflix on Android devices (this was observed in previous work [84]). We can also see in both cases that YouTube on iPhone can get a higher bitrate than Netflix on laptop and Nexus tablet. If we had a QoE fair allocation, we would expect small screen devices (i.e. iPhone) to get a video stream of a lower bitrate than larger screen devices (i.e. laptop and Nexus). This motivates us to develop our new video QoE metric and then develop QoE max-min fair bitrate allocation.

Table 4: Different video profiles (resolutions and bitrates in Kbps) provided by Netflix and YouTube

Netflix	Res.	320 × 240	384 × 288	512 × 384	512 × 384	640 × 480	720 × 480	1280 × 720	1280 × 720
	Bitrate	235	375	560	750	1050	1750	2350	3000
YouTube	Res.	256 × 144	426 × 240	426 × 240	640 × 360	854 × 480	1280 × 720	1920 × 1080	
	Bitrate	190	260	360	400	850	2150	4000	

4.2.2 Video quality assessment

Digital video usually suffers from a wide variety of distortions during encoding, compression, and reproduction which may result in a degradation in visual quality. In order to quantify the visual quality of compressed video, subjective quality assessment should be used. In this method, the video is viewed by a set of independent users and each one is asked to evaluate the perceived video quality with a score from 1 to 5 with 1 being the worst quality. The numbers are then averaged to give what is known as Mean Opinion Score (MOS). However, this method is is very costly and time consuming which makes it unattractive to researchers.

Alternatively, several objective quality assessment metrics have been developed over the years. The most well known one is Peak Signal-to-Noise ratio (PSNR) which computes the average distortion between a compressed video and a lossless version of it which is usually called the reference video. However, PSNR is usually criticized because it has a weak correlation with perceived video quality [48]. Structural Similarity Index (SSIM) [118] is another objective metric that takes advantage of the fact that images are highly structured and that pixels usually have strong dependence with spatially close ones. Experiments have shown strong correlation between SSIM and MOS which represents ground truth for perceived video quality [118]. None of the metrics in the literature, however, consider the screen size of the displaying device when evaluating video quality. We, therefore, extend the definition of QoE metrics in section 4.3 to include screen size then we apply the new definition to SSIM to get a new device-dependent QoE metric.

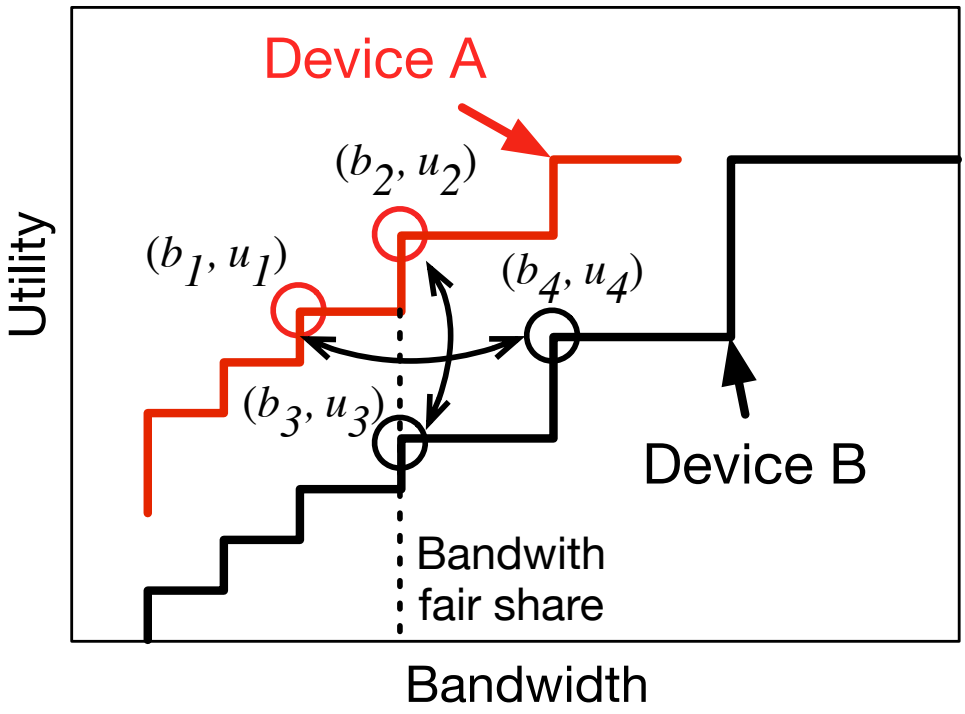


Figure 11: Adaptive video utility function

4.2.3 Utility max-min fairness

Bandwidth max-min fairness, defined in [36], is a widely used fairness definition for allocating shared bandwidth among competing flows in a network. The objective is simply to maximize the minimum bandwidth allocated to any of the flows. This can be achieved by giving equal share of the bandwidth to all connections bottlenecked at a link, assuming their demands exceed this share. However, bandwidth max-min fairness treats all flows equally in terms of the utility they get from bandwidth. This is not always correct in the case of multiple adaptive video flows as explained in the previous section. In [109], Shenker proposed to extend the Internet service model to be more tied to user application *utility*. In addition, he investigated the characteristics of the utility function of different classes of traffic including elastic TCP traffic (e.g. HTTP, FTP, etc.) and realtime UDP traffic (e.g. VoIP). Inspired by his work, *utility max-min fairness* was introduced and developed in multiple contexts [38, 77, 107, 108], and in this work we adopt a similar approach.

For adaptive video streaming, bandwidth *utility* is the perceived video quality. Since

video quality improves only when the video switches to higher bitrates, and since a video stream has only a limited number of bitrates, bandwidth *utility* is a step function with step utility increments when bandwidth equals video bitrates. In addition, we expect different devices – with different screen sizes – to have different *utility* functions for the same set of video bitrates. In figure 11 we plot the potential utility functions of two devices, A and B, for a set of video profiles. The point (b_1, u_1) – marked with a circle – means that *Device A* gets a utility of u_1 when it is allocated bandwidth b_1, where b_1 is one of the video bitrates[1]. The step function means that *Device A* will not receive any additional utility until the allocated bandwidth reaches b_2; only then utility will become u_2.

In order demonstrate the benefits of utility fair allocation as compared to bandwidth fair allocation we consider the following example. Consider a case when the two devices, A and B, stream a video with the bitrates in figure 11 and share a bottleneck link of capacity C. Assuming $b_1 + b_4 = b_2 + b_3 \leq C$, there are two ways bandwidth can be allocated to the two devices: 1) *Device A* gets bitrate b_1 with utility u_1 and *Device B* gets bitrate b_4 with utility u_4, and 2) A and B get (b_2, u_2) and (b_3, u_3), respectively. Case 1 is what we get from utility fair allocation while case 2 is what we get from bandwidth fair allocation (note that $b_2 = b_3 = C/2$ is the bandwidth fair share for both devices). It is clear that, while bandwidth fair allocation is agnostic of QoE ($u_2 >> u_3$), utility fairness gives similar QoE for both devices ($u_1 \simeq u_4$).

4.3 Adaptive video QoE fairness

In this section we first introduce our technique for extending existing video QoE metrics so they can be a function of the screen size and resolution. We then introduce the general network model and define QoE *max-min* fairness. We show that QoE *max-min* fairness is not always achievable for a set of adaptive video streams and thus we introduce QoE *maximal* fairness. We develop an algorithm for computing bandwidth allocations that results in QoE *maximal* fairness.

[1]Note that real adaptive video players will switch to a bitrate only when they estimate the available bandwidth to be higher than that bitrate, however, we only consider the theoretical case here. More details about system implementation are discussed in section 4.4

4.3.1 Device-dependent QoE metric

Any encoded video stream has two important features (among others); video resolution (the width and height of the video in pixels) and encoding bitrate. For a given video resolution, higher encoding bitrates usually mean better video perceptual quality. However, this is true only when the resolution of the display area is equal to the video resolution; i.e. when each pixel in the video maps to a single pixel on the screen. When the video resolution is lower than the physical screen resolution, the video gets scaled up [90] to the resolution of the viewing area which could significantly degrade the perceptual quality (depending on the difference between the two resolutions). Since there are usually different devices with the same physical resolution and with different screen sizes, screen size (in inches) is also an important factor to consider when measuring the degradation in video quality.

Since the best achievable video quality from a device happens when the video and physical resolutions are equal, we can use the ratio between the two resolutions to measure quality degradation. However, this measure does not incorporate the size of the display screen. Pixel density [12] defined as Pixels-per-Inch (PPI) is a common measure for digital screen resolution. PPI is defined as the number of pixels on the diagonal of the screen and can be computed using the formula

$$PPI = \frac{\sqrt{w^2 + h^2}}{d}$$

where w, h are the width and height of the screen in pixels respectively and d is the length of the screen diagonal (in inches). The advantage of PPI is that it combines resolution and screen size in one value that is comparable between different devices.

We define *normalized-PPI* (N-PPI) for a particular screen as the ratio between video PPI and physical PPI , where the first is computed using width and height from the video resolution, while the second is defined above. N-PPI measures the relative quality of a video to the best quality achievable on the device. Since there is no additional quality gain from playing a video of a higher resolution than the physical one, the maximum value of N-PPI is 1, in fact N-PPI is computed as

$$\text{N-PPI} = \min\left(\frac{PPI_v}{PPI_{phy}}, 1\right).$$

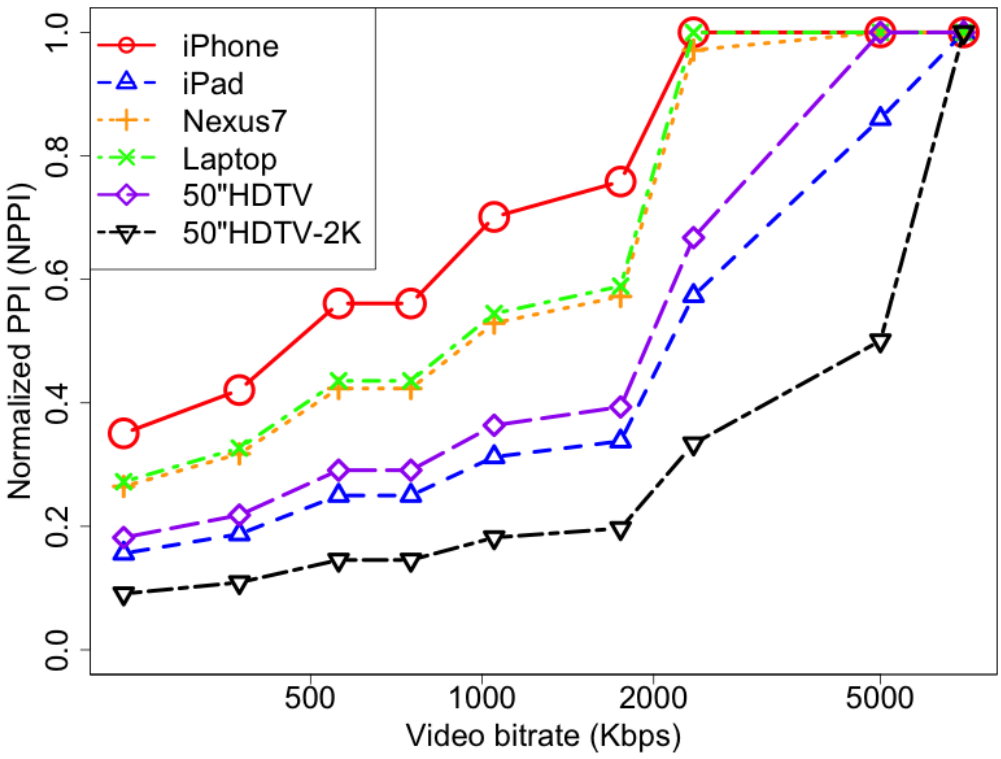


Figure 12: Normalized Pixel-Per-Inch (PPI) for different devices and various video profiles

In figure 12 we plot the N-PPI values for several devices with different screen sizes and resolutions. For each device, we compute a set of N-PPI values using the common set of video resolutions provided by Netflix [2] (table 4). Finally, we extend the SSIM [118] metric and we define our new video QoE metric to be

$$Q = \text{SSIM} * \text{N-PPI}$$

Note, however, that SSIM can be replaced by any other objective QoE metric. Figure 13 shows the new metric Q that corresponds to the N-PPI values in figure 12.

4.3.2 QoE max-min fairness

Consider the graph representation of a network used to deliver adaptive video streams. Define $G = (V, E)$ as the network graph with $|V|$ nodes and $|E|$ links and links connect the nodes in an arbitrary fashion. Each link in the network has a limited capacity; denote the capacity of link l as C_l. A routing algorithm is used to compute the best route between every

[2] Although Netflix manifest files are encrypted, they can obtained using the Tamper Firefox plug-in

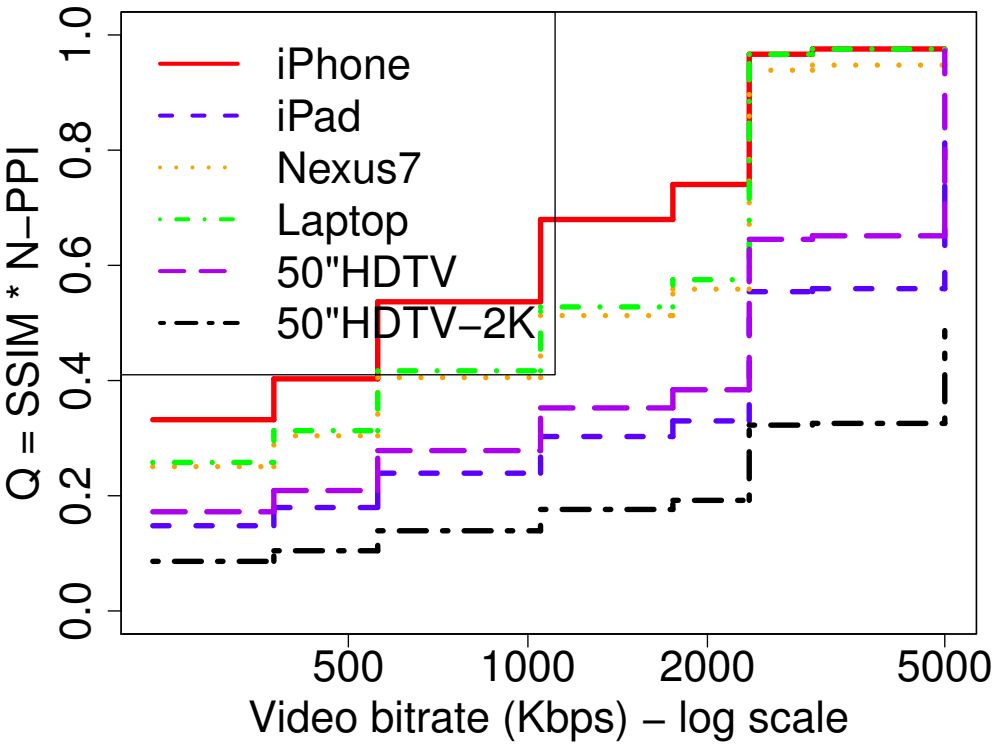


Figure 13: The new extended QoE metric $Q = \text{SSIM} * \text{N-PPI}$

source and destination in the network and the set of links composing a route between two nodes is called the data path between them. The network is used to deliver video streams from video servers (sources) to end-users (destinations), although we do not assume that sources and destinations belong to V. This means that G represents only a part of the video distribution network which matches perfectly with how video distribution on the Internet works. For example; G could be the network of the end-user Internet Service Provider (ISP) in which case only destinations belong to V, or G could be a transit ISP used to deliver video from content providers to the home ISP network in which case neither sources or destinations belong to V.

We define N to be the number of adaptive video streaming sessions where each video is available at the video server in multiple bitrates. Define $M_i, i \in \{1, \ldots, N\}$ as the number of video bitrates of stream i and define $R_{i,j}, j \in \{1, \ldots, M_i\}$ as the j^{th} video bitrate of the i^{th} stream. Without loss of generality, assume that $R_{i,1} < R_{i,2} < \ldots < R_{i,M_i}$ for all streams. We define $Q_{i,j}$ as a value representing the computed quality of video stream $R_{i,j}$

and $Q_i = \{Q_{i,j}\}$ as the set of all bitrates of video session i. Computing the quality metrics values was described in section 4.3.1. Define P_i as the data path of video session i or the set of links in E traversed by the video in session i and define S_l as the set of video sessions that traverse link l.

A QoE allocation q_i is the QoE value allocated to session i and hence $q_i \in \{Q_{i1}, \ldots, Q_{i,M_i}\}$. Define $\psi_i(q_i)$ as the video bitrate of session i with QoE value q_i, for example if $q_i = Q_{i,2}$ then $\psi_i(q_i) = R_{i,2}$. The N dimensional vector $\vec{q} = (q_1, \ldots, q_N)$ is a feasible QoE allocation if each video session is allocated a feasible QoE value, i.e. $q_i \in \{Q_{i1}, \ldots, Q_{i,M_i}\}$, and the total data rate on each link does not exceed link capacity, i.e.,

$$\forall l \in E : \sum_{i \in S_l} \psi_i(q_i) \leq C_l$$

Throughout this section we follow an approach similar to the one in [108] although their work introduced *max-min* utility allocation for multirate multicast networks while our work solves the problem for unicast adaptive video streaming.

Definition 1. QoE max-min fairness. *A feasible QoE allocation is max-min fair if it is not possible to increase the QoE of one session while maintaining feasibility without reducing QoE of another session that had equal or lower QoE. More formally, if $\vec{q}$ is a feasible QoE allocation then $\vec{q}$ is* max-min *fair if for any other feasible allocation $\vec{q'}$ if $q'_i > q_i$ for any index $2 \leq i \leq N$ then there exists an index j such that $q'_j < q_j$ and $q_j \leq q_i$.*

Observe that the set of feasible QoE values is a discrete set because each video session has only a limited number of video bitrates. It was shown in [100, 107] that max-min allocation may not exist for discrete feasible sets which is the same in our case; we describe an example to show that. Consider a simple case with a single link of capacity C shared by two adaptive video streams. Both videos are available in the same bitrates $(C/3, 2C/3)$ and produce the same QoE values (w_1, w_2) where $w_2 > w_1$. The set of all feasible QoE allocations is $(0,0), (0,w_1), (0,w_2), (w_1,0), (w_1,w_1), (w_1,w_2), (w_2,0), (w_2,w_1)$ (refer to the feasibility condition above). We can easily show that none of these allocations is max-min fair. For example $\vec{q} = (w_1, w_2)$ is not max-min fair because $\vec{q'} = (w_2, w_1)$ is a feasible allocation and $q_1 < q'_1$ but $q_2 > q_1$ and $q'_2 < q_2$ which violates the max-min condition

in the definition above. The reader can use a similar approach to see that none of the feasible allocations is max-min fair. Since QoE max-min fair allocation may not exist we use *lexicographic optimality* defined below.

Definition 2. Lexicographic order *For any vector* $\vec{X} = (x_1, \ldots, x_N)$ *the lexicographically ordered version of* $\vec{X}$ *is* $\vec{\bar{X}} = (\bar{x}_1, \ldots, \bar{x}_N)$ *where* $\bar{x}_i \leq \bar{x}_{i+1}$, *and* $\bar{x}_i = x_j$ *for some index* $1 \leq j \leq N$.

Definition 3. Lexicographic optimality *For any two feasible QoE allocation vectors* $\vec{X} = (x_1, \ldots, x_N)$ *and* $\vec{Y} = (y_1, \ldots, y_N)$, X *is said to be lexicographically greater that* Y *(written* $X >_{QoE} Y$*) if there exists* i *such that* $\bar{x}_i > \bar{y}_i$ *and* $\bar{x}_j = \bar{y}_j$ *for* $j < i$. *A feasible QoE allocation* $\vec{X}$ *is lexicographically optimal if* $X >_{QoE} Y$ *for all other feasible QoE allocations* Y.

Informally, a lexicographic optimal allocation is a feasible allocation in which the smallest component is the maximum smallest component of all feasible allocations and the same condition holds for the second smallest component and so on. In the previous example, both $(w1, w2)$ and $(w2, w1)$ are optimal lexicographic allocations because they are lexicographically equal to each other and they are lexicographically greater than any other feasible allocation according to the definition above. Although lexicographic optimality is not completely equivalent to max-min fairness, it has been used to define max-min fairness in some works [38]. Lexicographic optimal allocation always exists for discrete feasible sets if the feasible set is closed and bounded [107] which makes it more suitable in our case than max-min fair allocation which may not exist. However, if the latter exists for a discrete feasible set, it is equal to the lexicographic optimal allocation.

Computing a lexicographic optimal allocation for a discrete feasible set is not simple though, it was shown in [107] that it is a NP-hard problem. The high level idea of the proof is that finding a lexicographic optimal allocation is equivalent to finding the largest independent set of a graph which is known to be NP-hard. As an alternative, we use the relaxed definition of *relative fairness* introduced in [105]. A feasible QoE allocation $\vec{q}$ is *QoE-fairer* than another feasible allocation $\vec{q'}$ if the following condition holds: if there

exists an i such that $q'_i > q_i$ then there exists a j such that $q_j \leq q_i$ and $q'_j < q_j$.

Definition 4. Maximal fairness *A feasible QoE allocation $\vec{q}$ is maximally fair if no other feasible allocation $\vec{q'}$ is QoE-fairer than $\vec{q}$.*

Note that the definition of *maximal fairness* is very similar to that of max-min fairness, however a maximally fair allocation always exists for a discrete feasible set and can be computed in polynomial time [106]. In addition, if a max-min QoE allocation exists it will be equal to the maximally fair one and if it does not exist then a lexicographically optimal allocation is one of the maximally fair ones [106].

4.3.3 Computing QoE maximal fair allocation

In this section we present an algorithm for computing a *maximally fair* QoE allocation, algorithm 1 summarizes the steps of the algorithm. The high level idea is very similar to the progressive filling algorithm introduced in [36] for computing max-min fair allocation. The algorithm operates over iterations; in every iteration each session is allocated its fair share of QoE. A video session becomes *saturated* if it gets constrained by the residual capacity on the set of links it traverses such that it can not be upgraded to the next QoE level. This is similar in a sense to the *bottleneck link* definition in the original max-min algorithm in [36]. Once a session is *saturated*, its QoE allocation is fixed and the residual capacity is distributed on the rest of the sessions sharing the same links with that session. The algorithm terminates when all sessions get the best possible QoE (highest video bitrate) or when they all get saturated. We introduce some notations below then we describe the algorithm in details.

Define q_i^+ as the next higher QoE value of session i if the current value is q_i. For example, if the current value is $q_i = Q_{i,j}$ then the next higher value is $q_i^+ = Q_{i,j+1}$ where $1 \leq j \leq M_i - 1$ and $q_i^+ = q_i$ for $q_i = Q_{i,M_i}$. As mentioned above, a video session gets saturated when it is not possible to upgrade its QoE to the next level, this can be formally described by the following two conditions:

$$\sum_{i \in S_l} \psi_i(q_i) = C_l$$

Table 5: Summary of used symbols

G	Network graph
V	Set of nodes in the network
E	Set of edges in the network
C_l	Capacity of link l
N	Number of adaptive video flows
M_i	Number of bitrates of video session i
R_{ij}	Bitrate of video profile $\#j$ of session $\#i$
Q_{ij}	QoE of video profile $\#j$ of session $\#i$
Q_i	Set of video bitrates of video session i
P_i	Set of links in E traversed by video session i
S_l	Set of video sessions traversing link l
$\psi_i(q_i)$	Video bitrate of session i with QoE q_i
	Section 4.3.3
q_i^+	The next quality level of session i higher than q_i
$\phi_i(q)$	Largest quality level of session i that does not exceed q
H_l	Set of bitrates of all sessions going through link l
Θ_l	Set of unsaturated sessions going through link l
D_l^a	Bandwidth allocated to saturated links going through link l at the end of iteration a
β_l^a	Fair utility to all sessions passing through link l during iteration a
δ_i^a	Utility allocated to session i after iteration a

$$\text{or } \sum_{i \in S_l} \psi_i(q_i) + [\psi_k(q_k^+) - \psi_k(q_k)] > C_l, \;\; k \in S_l.$$

The first condition represents the case when the current QoE allocation causes link l capacity to be fully utilized. In that case all video sessions traversing l, i.e. S_l, are saturated. The second condition represents the case when link l still have some unused capacity but it is not enough to grant video session k the next QoE level.

Define $\phi_i(q)$ as the highest QoE value of session i that is less than or equal to q, more formally

$$\phi_i(q) = \max\{x : x \in \{Q_{i,1}, \ldots, Q_{i,M_i}\} \text{ and } x \leq q\}$$

Define H_l as the set of all QoE values that could be obtained from video sessions going through link l; formally, $H_l = \bigcup_{i \in S_l} \{Q_{i,1}, \ldots, Q_{i,M_i}\}$.

The algorithm operates in iterations, in each iteration the algorithm goes through a set of steps described in Algorithm 1. In the first two steps, variables are initialized including X_a and Y_a which are defined as the set of unsaturated and saturated video sessions at the

Algorithm 1 Computing QoE maximal fair allocation

1: **Inititialization:**
2: $a = 0, X_0 = \{1, \ldots, N\}, \Theta_l = S_l, D_l^{-1} = 0$
3: $H_l = \bigcup_{i \in S_l} Q_i,\ Y_a = \{\}$
4: **while** $|X_a| > 0$ **do**
5: $\beta_l \leftarrow \max_q \{q : q \in H_l \ \wedge \ D_l^{a-1} + \sum_{i \in \Theta_l} \psi_i(\phi_i(q)) \leq C_l\}$
6: For each session i, compute allocated utility:
7: $\forall i : \delta_i \leftarrow \phi_i(\min_{l \in P_i} \beta_l)$
8: Compute saturated sessions Z_a:
9: $Z_a \leftarrow \{i : \delta_i = Q_{i,M_i}$ OR
10: $\exists l \in P_i : \phi_i(\delta_i^+) + \sum_{j \in S_l, j \neq i} \phi_j(\delta_j) > C_l\}$
11: **if** $|Z_a| == 0$ **then**
12: Select session s and upgrade it: $\delta_s \leftarrow \delta_s^+$
13: **end if**
14: $Y_a \leftarrow Y_a + Z_a$
15: $X_a \leftarrow X_a - Z_a$
16: $\forall l : D_l^a \leftarrow \sum_{i \in Y_a} \psi_i(\delta_i)$
17: Update Θ_l, H_l
18: $a \leftarrow a + 1$
19: **end while**

end of iteration a, respectively. In each iteration the algorithm first computes the fair QoE value that should be assigned to each session without violating the capacity condition on any link (step 5). In step 7, the algorithm computes for each session the actual allocated QoE value which is computed as the minimum of the QoE values assigned to that session over all the links of its path. Based on the newly allocated QoE values, the algorithm computes in step 9 the set of saturated sessions Z_a. If Z_a turns to be empty, the device with the largest possible screen is upgraded to the higher quality without violating link capacity conditions (step 12). The algorithm then updates X_a and Y_a and terminates when X_a is empty.

4.4 Implementation challenges

As mentioned earlier in section 4.3, QoE fair bitrate allocation can be deployed in different parts of the video distribution network; examples are: home network, local or transit ISP, or video CDN. In a typical implementation we envision a centralized controller that collects information about all active video streaming sessions, their available video profiles,

streaming devices, and that controller will compute the QoE fair allocation and then delegate bitrate enforcement to other nodes in the network. A real system that is capable of doing that, regardless of where it is deployed, needs to implement certain functionality that makes it a very challenging task. In this section we first list the common functionality that needs to be implemented by any such system then we discuss why they are challenging to implement and how we can address these difficulties.

A QoE fair allocation system has to implement the following functions: HTTP traffic monitoring, computing the QoE metric, implementing the QoE fair allocation algorithm, and enforcing desired bitrates on video players. We discuss in detail below each one of these points.

4.4.1 HTTP traffic monitoring

In order to be able to implement the QoE fairness algorithm, the system needs to keep track of all active video streaming sessions. The only way to do that is to monitor all outbound HTTP requests from all users and detect the ones that represent adaptive video streaming sessions. In addition, for each streaming session the system needs to learn about the type of streaming device used (smart phone, tablet, set-top-box, etc.) and the video manifest file. The first gives information about the device screen size and resolution, while the manifest file gives information about all the available bitrates and resolutions of the video. These two pieces of information are important for computing both the QoE metric and QoE fair allocation. It is easy to identify video flows for most popular streaming services (e.g. Netflix and YouTube) from the format of their HTTP requests.

Monitoring HTTP traffic to extract the information described above poses several concerns; some for users and some for network operators. The following points summarize some of these concerns and why we think it is acceptable to monitor HTTP traffic.

Deep packet inspection (DPI). Peeking into HTTP request headers requires some nodes in the network to perform DPI on all HTTP traffic. Performing DPI usually raises two concerns: 1) DPI is expensive and usually requires special hardware to handle large volumes of data at line speed, and 2) users may worry about their browsing and video streaming

behavior getting exposed (to ISPs). The first point is usually true if DPI is required to be performed on all traffic, however, in our case we need to look only at outbound HTTP requests which should not be a large volume of traffic. HTTP traffic initiated by clients can be simply identified by the `(TCP, dstport 80)` pair. However, in order to distinguish HTTP request packets from `TCP ACK` ones, the packet inspection filter can add a packet length condition to filter out `ACKs`. Users' concerns about exposing their streaming behavior can be addressed by ISPs by adding terms in users' service contracts promising not to collect any data or do any user profiling based on the colleced data.

Reading manifest files. Getting information about video profiles available to each active streaming session is crucial for the operation of the system. A video player usually downloads a manifest file that includes this information at the beginning of a streaming session. The manifest usually includes additional information about the used CDN(s) and their ordering, IP addresses and URLs needed to download video segments from the CDNs, and other information. Some commercial services have the manifest file encrypted probably for security reasons to protect their CDNs (e.g. Netflix) while other services do not encrypt their manifest files (e.g. YouTube). In addition, MPEG-DASH [10] uses an unencrypted XML manifest file that should be standard in the future. For services with encrypted manifest files, we propose splitting the manifest into two separate files. The first file should not be encrypted and includes information about the available video profiles (bitrates and resolutions), while the second file is encrypted and includes all sensitive information about CDN server IPs and URLs of video segments.

An alternative to having access to the manifest file is to infer the bitrates of video streams. The nominal bitrate of a video stream can be computed by dividing the total size of the video in bits by its length in seconds. For some video services (e.g. YouTube) this information can be available in the URL of video segment requests. However, this information may not be available for other services. In that case, the bitrate of a video profile can be estimated as the average of the bitrate of several segments that belong to that profile, where the latter is computed as the size of the segment in bits by its length in seconds. The segment length can be estimated as the length of the player ON/OFF

cycle in steady state. Obviously, this approach requires keeping a database of the bitrates of different video profiles of all streams. Initially the database is empty and the system accumulates video bitrates to it as it monitors users' streaming sessions.

4.4.2 Computing QoE metric

In order to compute the N-PPI metric for all clients, the centralized controller needs to know about all video profiles of streaming sessions (discussed in the previous section) in addition to screen size and resolution of all streaming devices. HTTP request headers of popular streaming services usually include information about the device but not in the level of details we desire. For example, it is easy to identify an iPhone or an iPad but not the version of the device. Also, it is possible to identify a set-top-box but not the screen size and resolution of the TV connected to it. Thus, we propose that video players should include more information about the streaming device in the HTTP request headers.

4.4.3 Enforcing desired video bitrates

A video player decides which bitrate it streams based on its estimate of the available bandwidth which is usually computed as a function of the download rate of one or more of the recently downloaded video segments[3]. Since there is no standard specification for this function, each player implements its own estimation method. In addition, the common wisdom for video player design is that a player will switch to bitrate r only if it estimates the available bandwidth to be no less than $(1 + \alpha)r$ where $\alpha > 0$. This ensures that the player will be able to fill the player buffer with bitrate r which is crucial to avoid buffer under-runs and video stalls resulting from temporary drops in the available bandwidth. Moreover, recent work [84] has shown that α could be different for the same streaming service on different platforms.

After computing QoE fair bitrates, the centralized controller delegates the task of applying these bitrates to other nodes in the network. These nodes should be able to allocate adequate bandwidth to each client to effectively force it to stream its designated bitrate.

[3]Recent work [61] suggested performing bitrate adaptation based on the growth rate of the player buffer, however, bandwidth estimation remains to be the common way of doing adaptive streaming

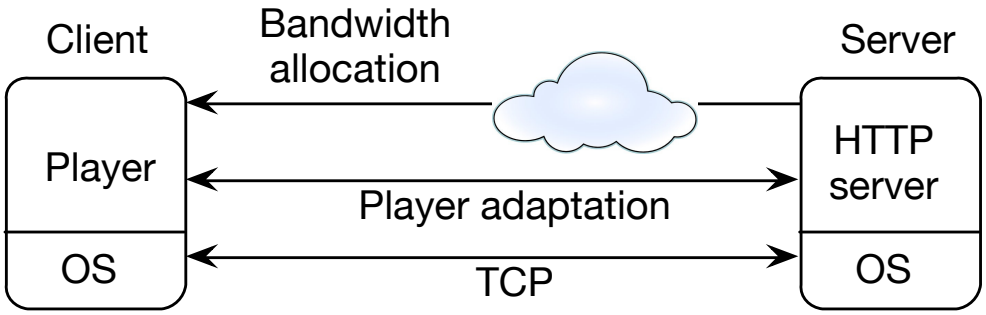


Figure 14: Multiple control loops affecting the operation of adaptive video players

However, figuring out the proper bandwidth value to stream a specific bitrate on a certain platform could be very challenging because, as explained above, this value is client implementation dependent. This can be handled in two different ways:

1. Carrying out performance experiments for all streaming services on all popular devices to find out the mapping between the available bandwidth and the bitrate selected by the client. The good thing about this solution is that streaming services usually have a fixed set of bitrates (and resolutions) for most of their video catalog which means these experiments have to be done only for a few number of video streams; Table 4 shows common video parameters for Netflix and YouTube. However, with such a solution new experiments have to be performed for new devices or when player code is updated.

2. Enabling a control channel at the video player which can be used by the network controller to communicate the desired bitrate. The client then will have to honor the network decision and commit to streaming that bitrate until a new bitrate is delivered through the channel. A simple way of implementing such a channel is using WebRTC [17]. WebRTC is an open source API supported by most popular web browsers to enable real time communication using native JavaScript without any additional plugin software.

It is important here to highlight that the operation of an adaptive video player is mainly controlled by three control loops; figure 14 illustrates these loops. The first loop results from

TCP behavior in the transport layer. It is known [41] that TCP tries to fill the pipe between the sender and the receiver by sending bursts of unacknowledged data until it detects packet loss, only then TCP backs off and reduces its sending rate. This adaptive behavior helps network nodes sending TCP traffic to avoid congestion collapse. The second control loop is the adaptive bitrate behavior implemented by video players. A video player usually uses the download rates of downloaded video segments to estimate the bandwidth available to the server. This is done by computing an average of these values over a window of video segments (note, however, that bandwidth estimation is implementation specific and differs for different players). Based on the player estimation of the available bandwidth, it decides the bitrate of the next video segment to download. Traditionally, adaptive video players have only these two loops, however, our work introduces a new control loop in which video players adapt to bandwidth allocation enforced by the network. Understanding the different time-scales of the operation of these loops and the interaction between them is very crucial for controlling the operation of adaptive video players. This is beyond the scope of this thesis and we defer it for future work.

4.5 VHS: QoE fairness in a home router

In this section we present the design and implementation of *VideoHomeShaper*; a system that implements QoE fairness in home networks. As seen in figure 15, a typical scenario for VHS is in the home network where multiple users use different devices and different streaming services. VHS runs on home routers where it monitors users' traffic, computes QoE fair bitrates, and then enforces them. We implemented VHS and evaluated its efficacy for the two most popular streaming services, Netflix and YouTube, with players running on a variety of platforms (PC, iOS, and Android).

As depicted in figure 16, VHS is composed of four main modules: traffic filter, feature collector, session manager, and bandwidth manager. We describe each one of these modules in detail below.

Traffic Filter. VHS monitors all outbound traffic going from the home router to the Internet. The traffic filter module is used to identify HTTP requests of video streaming sessions.

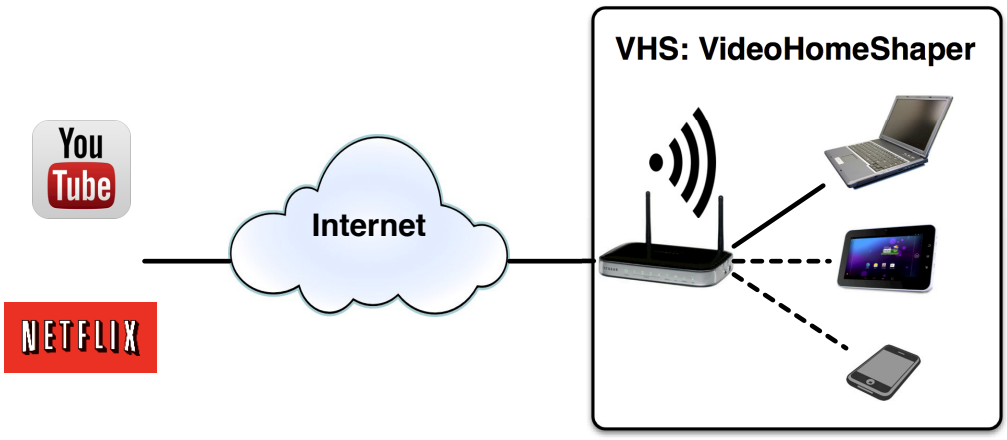


Figure 15: VHS runs on home routers and controls bandwidth allocated to each video stream at home to achieve QoE fairness

This is done by first filtering out all non-HTTP traffic using the pair `(TCP, port 80)` and then matching HTTP request headers to pre-defined patterns that represent video streaming services. Note that a video request here means either the manifest file or a video/audio media segment. In our implementation we added pattern matching only for Netflix and YouTube over three platforms: PC, iOS, and Android[4], however, it is fairly simple to extend to other streaming services. When a HTTP video request is identified, it is forwarded to the next module; the *feature collector*.

Feature Collector. When a new HTTP video request is forwarded to the feature collector, it first checks with the next module, *session manager*, to see whether this is a new streaming session or it belongs to an existing session. If the request is identified as a new streaming session, the *feature collector* module parses the HTTP request header fields to identify the device type, video stream identifier, and video profile identifier. This module is also responsible for parsing manifest files (if available) and extracting all video profiles (bitrates and resolutions) form them; this information is then forwarded to the *session manager* for QoE fairness computation.

Session Manager. This module keeps track of all active streaming sessions. For every new HTTP video request, if the request does not belong to an existing session it creates a

[4]HTTP request format changes for the same streaming service over different platforms

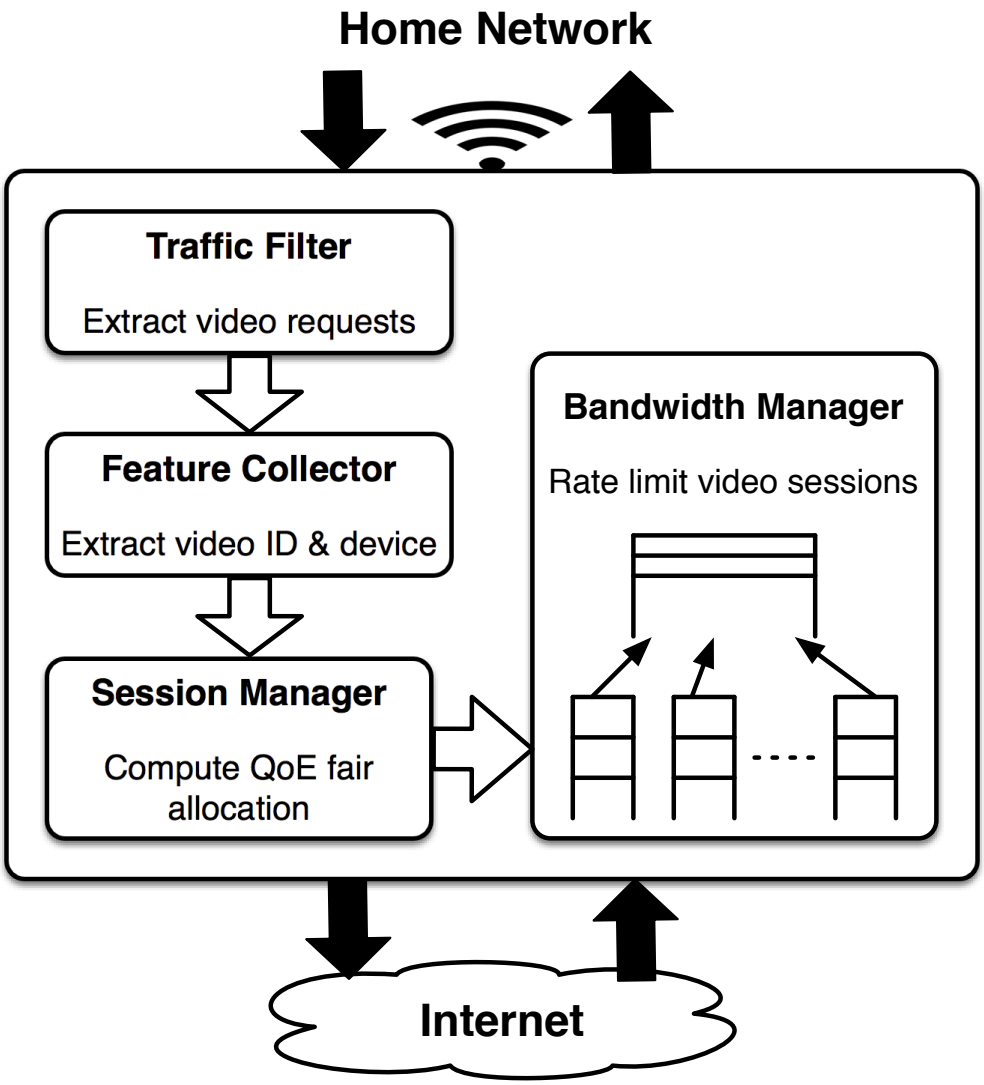


Figure 16: Inside a home router: VHS design

new session and adds it to the *session table*. The *session table* is a hash table that stores all information about active video sessions, the entry key in this table is the (*client IP, video identifier*) pair, where *client IP* is the local IP address of the client initiating the video request. Since, in a single streaming session, a video player can potentially download video segments from multiple video servers over multiple TCP connections (and hence multiple TCP ports), *video identifier* is the best field to identify a streaming session together with the client IP address. Moreover, the session manager keeps the timestamp of the last video request within each session and periodically computes the idle time of each session, if the idle time exceeds a threshold (we used 50 seconds in our implementation) then the session is deleted. Upon creating or deleting a session, the session manager computes a new set of QoE fair bitrates and passes them to the *bandwidth manager* module to enforce them.

Bandwidth Manager. This module is responsible for enforcing the QoE fair bitrates computed by the *session manager*. As mentioned earlier in section 4.4, a video player needs to estimate the available bandwidth to be $(1+\alpha)r_i$ for some $\alpha > 0$ in order to switch to bitrate r_i. We conducted several experiments to infer the value α for Netflix and YouTube for PC, iOS, and Android devices. Using the inferred values of α and the computed bitrates from the *session manager*, *bandwidth manager* uses Linux traffic control and `iptables` rules to allocate bandwidth to each video stream. Each stream is allocated a lower and an upper bandwidth values. The lower value is the guaranteed minimum bandwidth allocated for that session while the upper value represent its maximum allowable bandwidth. The upper value is needed so that a session does not switch to a higher bitrate than the desired one. Hence, it is set to a value lower than the minimum bandwidth required to get the higher video profile, i.e. $(1+\alpha)r_i < upper < (1+\alpha)r_i^+$ where r_i^+ is the bitrate of the profile one level higher than r_i.

The bandwidth manager always maintains at least two queues; a parent queue that is directly connected to the router bridge interface, and a child queue that is connected to the parent queue. The child queue is the default path for all non-video streaming traffic. For each new streaming session, the bandwidth manager creates a new child queue, assigns its upper and lower limits, and adds a new rule in `iptables` to forward streaming traffic

```
tc qdisc del dev $IFACE root

tc qdisc add dev $IFACE root      handle 1:    htb default 30
tc class add dev $IFACE parent 1:  classid 1:1  htb rate 6000kbit ceil 6000kbit burst 30k

tc class add dev $IFACE parent 1:1 classid 1:2 htb rate 1000kbit ceil 6000kbit burst 30k
tc qdisc add dev $IFACE parent 1:2 pfifo limit 64

tc class add dev $IFACE parent 1:1 classid 1:3 htb rate 2500kbit ceil 3500kbit burst 30k
tc qdisc add dev $IFACE parent 1:3 pfifo limit 64

iptables -t mangle -A POSTROUTING -o $IFACE -j CLASSIFY --set-class 1:2
iptables -t mangle -A POSTROUTING -o $IFACE -s $SERVER_IP -d $CLIENT_IP -j \
        CLASSIFY --set-class 1:3
```

Figure 17: Example of a Linux traffic control configuration similar to the one created by VHS

to that queue. When the *session manager* detects that a streaming session has stopped it signals the bandwidth manager which in turn deletes the proper child queue and its `iptables` rule. Figure 17 gives an example of a configuration similar to the one created by the *bandwidth manager*. This example creates a parent queue `classid 1:1` and two children queues `classid 1:2` and `classid 1:3`. Two `iptables` rules are defined; the first marks all packets to be forwarded to the default queue `1:2` while the second marks all packets from `SERVER_IP` destined to `CLIENT_IP` to be forwarded to queue `1:3`. Note that all non-video streaming traffic will be forwarded to the default queue. It is important to mention here that forwarding the traffic of each streaming session to a separate queue could introduce significant overhead at the router. Although this may not be true for the home case where only a limited number of video flows can potentially coexist during any period of time, implementing this solution at a router in a large scale network can be very challenging. In that case, the router needs to implement high performance scheduling and rate-limiting techniques to be able to allocate bandwidth to thousands of concurrent video flows; we leave this problem as future work.

Implementation. We implemented VHS on a NETGEAR home router model WNDR3700 running OperWrt firmware [11]. VHS is written in C++ and it is about 1500 lines of code,

Table 6: Devices used in the experiments

Device	OS	Screen Resolution	Hardware specs
iPhone 4S	iOS 7	960×640	
Nexus 7	Android 4.2	1280×800	
Nexus 10	Android 4.2	2560×1600	
Laptop I	OS X 10.9	1280×800	2.5 GHz Intel Core i5, 4GB RAM, 13"
Laptop II	Windows 7	1366×768	2.2 GHz Intel Core2 Duo, 3GB RAM, 14"
Laptop III	Ubuntu 12.04	1280×800	1.2 GHz Intel Core Duo, 1GB RAM, 13"

however, a significant part of the code is for monitoring and logging. Our code uses libpcap [9] to monitor inbound and outbound traffic on the bridge interface between the WAN port and the internal ports on the router. This enables us to detect traffic from clients connected to the router through both wired and wireless interfaces.

4.6 Evaluation

In this section we evaluate VHS in a real home setting. We conducted experiments with several devices streaming videos from Netflix and YouTube while sharing the home bottleneck link. We use these two commercial services for evaluation as compared to standard DASH players because DASH player implementations are very naive and their behavior is not a good representative of real commercial adaptive streaming players. In all the experiments we set the bottleneck link to be 6Mbps. We repeated the experiments with and without background traffic (mainly bulk file download) to demonstrate the potential advantages of VHS when there is significant background traffic. Table 6 describes the devices used in the experiments. Below we first define the evaluation metrics then we present experimental results.

Computing the SSIM metric for any encoded video requires having both the reference (lossless) video and the encoded video. However, since we do not have the reference videos for Netflix and YouTube videos, we need to follow a different approach. We download the raw (lossless) video of a movie widely used in research [1], then we use the resolutions and bitrates in table 4 to encode the movie. We then compute the SSIM metric using the

downloaded raw video and the encoded one.

4.6.1 Metrics

As mentioned earlier in section 5.1, previous work [32, 66] has shown that when multiple clients compete for bandwidth they can suffer from *unfairness*, *inefficiency* (i.e. link under-utilization), and *instability*. Although our main objective is to achieve QoE fairness, we need to make sure that VHS does not introduce bad behavior with respect to the other two metrics; link utilization and bitrate stability. We formally define the three metrics as follows:

- **Fairness.** Based on Jain fairness [69], we define the QoE fairness index F_{QoE} as

$$F_{QoE} = \frac{\left[\sum_{i=1}^{N} q_i\right]^2}{N \sum_{i=1}^{N} q_i^2}$$

  where q_i is the QoE value of the bitrate allocated to video session i. The fairness index has a value between 0 and 1, with 1 being the best fairness and 0 being the worst.

- **Utilization.** We define link utilization $U(t)$ at any time t as the percentage of the access link capacity used to stream video regardless of any background traffic, formally

$$U(t) = \frac{\sum_{i=1}^{N} r_i(t)}{C}$$

  where $r_i(t)$ is the bitrate of video stream i at time t.

- **Instability.** Bitrate instability is defined as the rate of bitrate change among all streams over time. We use the number of bitrate changes every 100 seconds as a good representative of the user experience concerning the stability of video quality. Define the indicator function $I_i(t) = 1$ if the video bitrate of stream i changes between times $t, t+1$ or $r_i(t+1) \neq r_i(t)$ and *zero* otherwise, then for a streaming session of length T seconds, instability is computed as

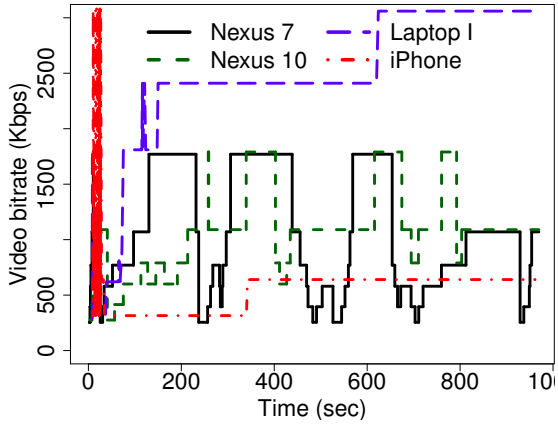
$$instability = \frac{\sum_{i=1}^{N} \sum_{t=0}^{T-1} I_i(t)}{T} \times 100$$

Since frequently switching video bitrate hurts users' video experience [43], having smaller values for the *instability* metric is desirable because it means more stable bitrates and better experience. Note that an alternative definition for bitrate instability is the *rate of change in video segment bitrate* which can be computed as the number of bitrate changes in consecutive video segments. This definition can be a good representative of the stability of the bitrate adaptation of the video player, however, it does not represent the viewing experience of the users. On the other hand, the number of bitrate changes every 100 seconds is a very good representative of the user viewing experience.

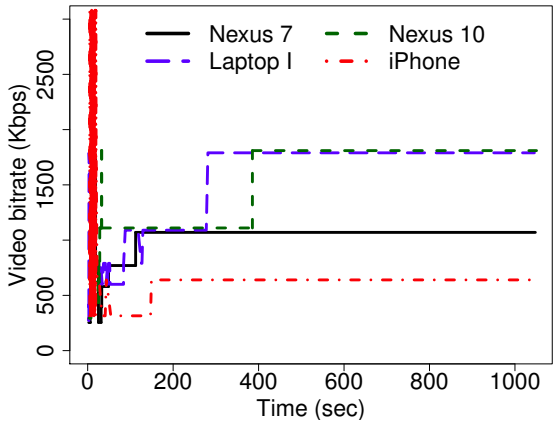
4.6.2 Experimental results

In the first set of experiments we use four devices; iPhone, Nexus 7, Nexus 10, and Laptop I, to stream the same movie from Netflix while setting the bottleneck link to 6Mbps. We repeated the experiment 10 times for each of the following cases: using VHS, same router not running VHS (we use the term Original to describe this system), and router allocating equal bandwidth of 1.5Mbps to each video session. For each experiment we record the streamed bitrates over time, and we use these values to compute the three performance metrics. Figure 18 shows the streamed bitrates over time for the three cases. We can clearly see form figures 18b and 18c that bandwidth allocation at the router improves bitrate stability of the two Nexus tablets for VHS and equal bandwidth, respectively.

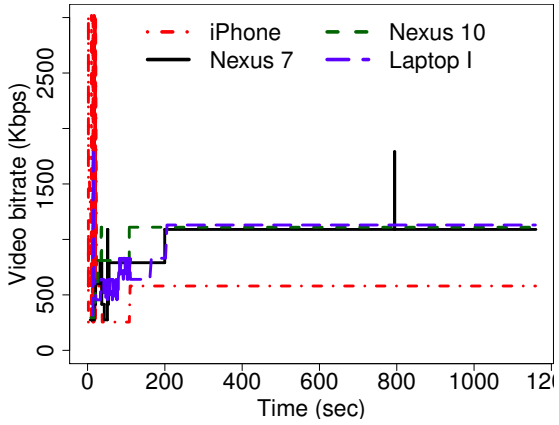
In figure 19 we plot the instability metric and the CDF of the fairness and utilization metrics; for the latter two we compute the metric every second and then compute the CDF. We can see that VHS significantly improves both QoE fairness and bitrate instability while achieving high link utilization. Figure 19a shows that VHS manages to maintain a fairness index greater than 0.94 for 90% of the time while the original system keeps it is less than 0.8 for 80% of the time. On the other hand, VHS achieves slightly lower link utilization than the original system. This is expected because we only have a set of discrete bitrates for each video stream and our main objective is achieve stability with the QoE fair computed bitrates. Otherwise, over utilizing the bottleneck may trigger video players



(a) Without VHS

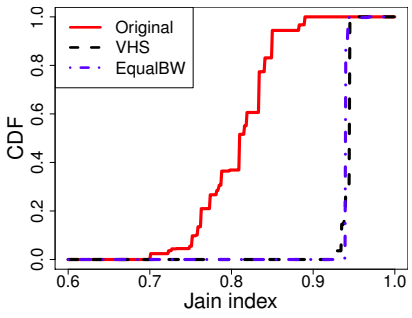


(b) With VHS

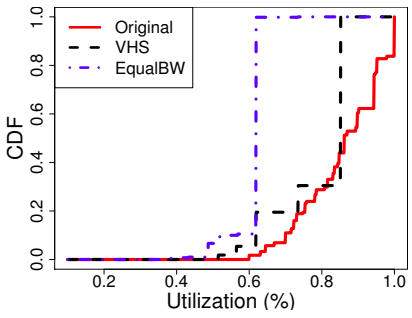


(c) Equal bandwidth allocated to each video session

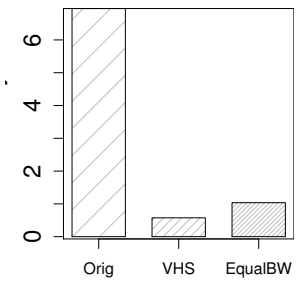
Figure 18: Four Netflix clients share a 6Mbps link with and without VHS



(a) Fairness

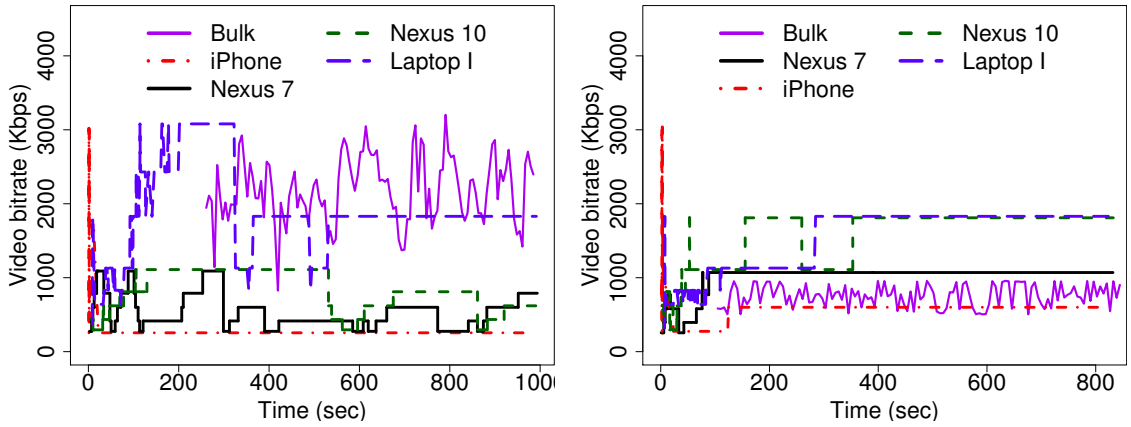


(b) Utilization



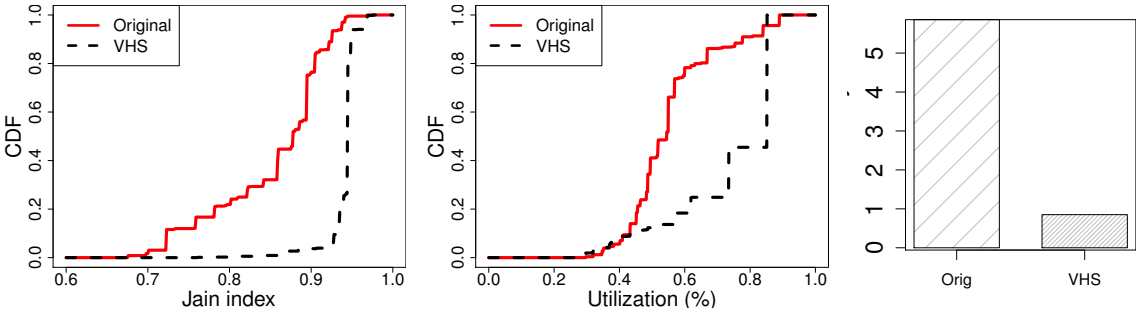
(c) Instability

Figure 19: Performance metrics: 4 Netflix clients



(a) Without VHS (b) With VHS

Figure 20: Four Netflix clients + file download share 6Mbps

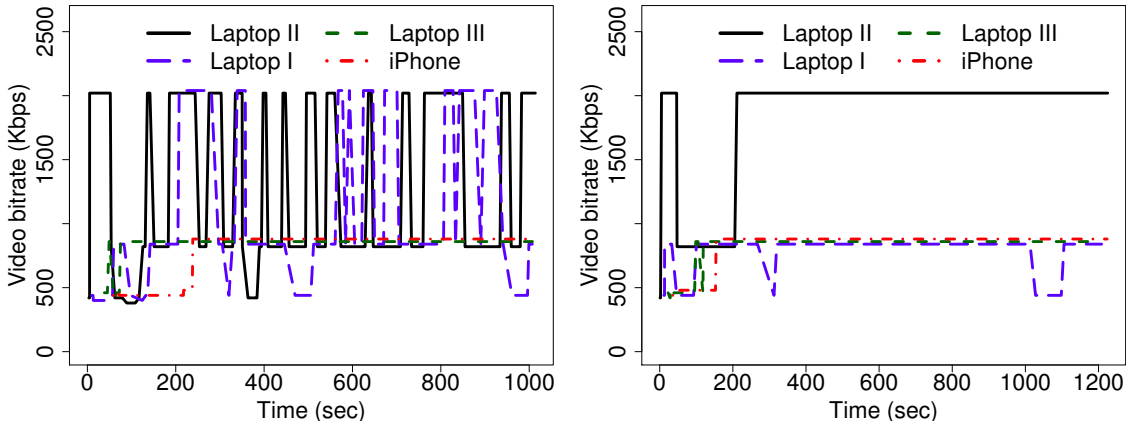


(a) Fairness (b) Utilization (c) Instability

Figure 21: Performance metrics: 4 Netflix clients + file download

to react and switch to other bitrates. This tradeoff between bitrate instability and link utilization was also observed in previous work [32, 40]. Moreover, although allocating equal bandwidth to each session improves fairness and bitrate stability, it falls short in utilizing the link capacity. Figure 19b shows that video players are able to utilize only 60% of the link capacity for this case. This is mainly because each player picks a bitrate lower than the estimated bandwidth. For example; the laptop and iPhone choose to stream bitrates 560Kbps and 1Mbps even when the allocated bandwidth is 1.5Mbps.

Previous work [60] has shown that a bulk file download can significantly affect the bitrate of an adaptive video stream when both are competing for bandwidth. In the second set of experiments we repeat the first set after adding a bulk file download. We used Laptop III to download a large file from a web server while streaming videos from the same four device



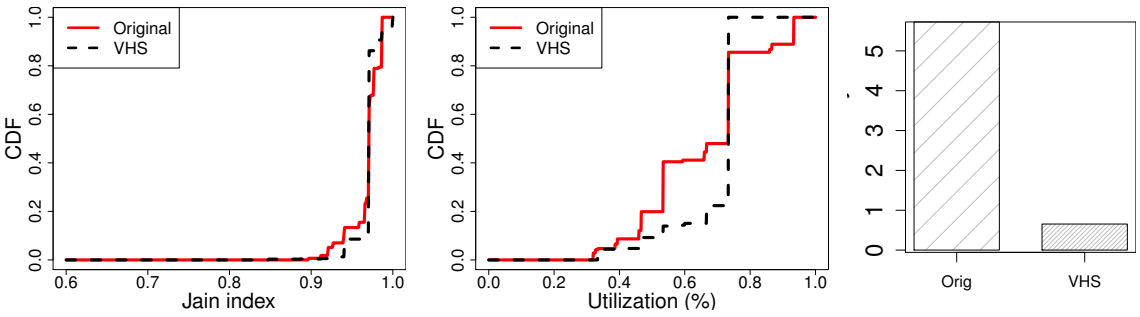
(a) Without VHS (b) With VHS

Figure 22: Four YouTube clients sharing 6Mbps

as the previous experiment.

Figure 20 shows the streamed bitrates for all clients in addition to the TCP throughput achieved by the file download. We can clearly see that the file download consumes about $20 - 30\%$ of the bottleneck link which significantly affects the video bitrates compared to figure 18a. On the other hand, figure 20b shows how VHS automatically limits the file download rate to about 1Mbps which leaves enough network bandwidth for video streams to get stable fair bitrates. This is done by forwarding file download packets to the default child queue in VHS which is limited to 1Mbps. Figure 21 shows the performance metrics for this experiment. We can see that VHS still outperforms the original system in QoE fairness and instability metrics but this time it also manages to utilize the link better than the original system (figure 21b). Remember here that we define utilization as the fraction of the bottleneck link bandwidth that is used to stream video. Since VHS controls the allocated bandwidth for background traffic, it will always be able to utilize the bottleneck link better than the original system when there is significant background traffic.

We conducted similar experiments to the previous ones for YouTube. Since YouTube application on Android does not support adaptive streaming [84] we decided to use three laptops and an iPhone in this set of experiments; devices specs are described in table 6. Figure 22 visualizes the streamed bitrates for the four devices when using VHS and with the original system. Figure 22a shows how competition between video streams can lead to



(a) Fairness (b) Utilization (c) Instability

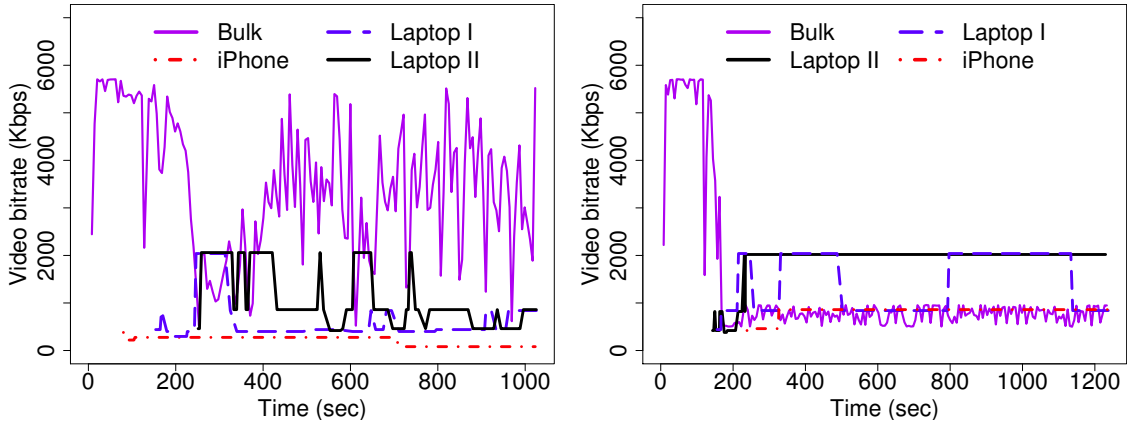
Figure 23: Performance metrics: Four YouTube clients

significant bitrate instability while figure 22b shows how VHS manages to keep the bitrates stable. The improvement in bitrate stability is quantified in figure 23c to be six times more for VHS than the original system. Comparing figures 18b and 22b, it is interesting to observe that Netflix clients exhibit more bitrate stability than YouTube when competing for bandwidth, although we recognize that we use a different set of devices in the two sets of experiments which could be a contributing factor.

Figure 23 shows that although the original system suffers from significant instability compared to VHS, it manages to provide high level of fairness comparable to VHS. This is explainable by figure 22a where we can see that two devices were lucky to receive their desired bitrates (Laptop III and iPhone) while the other two devices keep switching back and forth between their desired bitrate and another bitrate (desired bitrates can be observed in figure 22b).

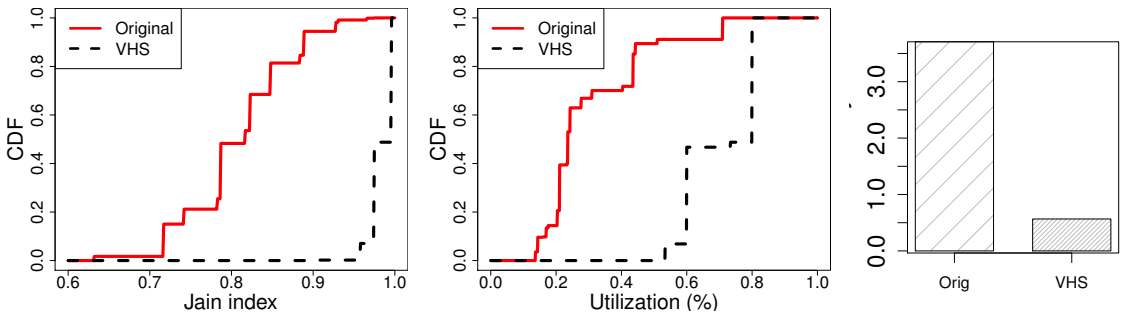
In the final set of experiments we add bulk file download as a background traffic competing with three devices streaming YouTube videos; two laptops and an iPhone. Similar to our observations with Netflix, file download manages to consume a significant portion of the available bandwidth but with YouTube it is even more significant than Netflix. Figure 24a shows that file download consumes between 50% and 65% of the bottleneck bandwidth which leaves only less than half of the available bandwidth for video streaming. VHS, on the other hand, is able to limit the bandwidth consumption of file download through its default child queue; this can be seen in figure 24b.

Figure 25 shows that we get improvement in all three metrics when VHS is used. VHS



(a) Without VHS (b) With VHS

Figure 24: 3 YouTube clients + file download on 6Mbps



(a) Fairness (b) Utilization (c) Instability

Figure 25: Performance metrics: Three YouTube clients + bulk file download

is able achieve QoE fairness of at least 0.96 for more that 90% of the time while the original system gives fairness less than 0.8 for 50% of the time. VHS is also able to utilize 80% of the available bandwidth for over 50% of the time while video streams manage to consume less than 40% of the bandwidth for 70% of the time with the original system.

4.7 Summary

In this chapter we focus on the fairness problem that emerges when multiple adaptive video streams compete for bandwidth. Due to the heterogeneity of streaming devices used today, we say that video bitrate is not the proper metric to define fairness. Instead, we develop a device-dependent QoE metric and then, based on that metric, we define QoE max-min fairness for a set of video streams sharing a network. We show that max-min fairness does not always exist for this problem, and instead, we define maximal fairness which always exists and is equal to max-min if the latter exists. Furthermore, we design and implement a system that works on home routers and is able to apply QoE fairness in the home network. Our system does not require any modifications to video players or servers and is tested on real commercial streaming services (Netflix and YouTube). Evaluation results show that our system can significantly improve QoE fairness while improving bitrate stability in the same time.

In the next two chapters we investigate two more problems that result from the interaction between HTTP streaming and the current Internet architecture. TCP is the transport protocol used by HTTP to deliver adaptive video streams. Since TCP is known to have bursty traffic, in the next chapter we study the effect of HTTP video traffic on the queueing delays in residential gateways. Another important problem that emerged recently is the excess video traffic load on peering links between ISPs and video streaming providers. One way to reduce the load on these links is to employ hybrid CDN/P2P systems; in chapter 6 we study this problem.

CHAPTER V

SABRE: A CLIENT BASED TECHNIQUE FOR MITIGATING THE BUFFER BLOAT EFFECT OF ADAPTIVE VIDEO FLOWS

5.1 Introduction

Excessive buffering of network devices on the Internet is a well known problem which has been studied in different contexts [35,91]. This problem was reintroduced recently by Gettys and Nichols in [54] under the name *buffer bloat*. In that study, the authors collected evidence to show that the Internet can suffer from significant congestion due to the existence of large buffers at different network devices. As a result, traffic can experience very high queuing delays that can reach several hundreds of milliseconds and sometimes even more than a second. High delays can be very harmful to many applications on the Internet such as VoIP, interactive games, and e-commerce.

The root cause of the *buffer bloat* problem is the way TCP (Transmission Control Protocol) works. In order for TCP to achieve the best throughput, it keeps a send buffer of approximately the bandwidth delay product (BDP) of the path between the source and the destination. This means that the maximum number of *bytes in flight* TCP can have is equal to the BDP. In addition, TCP uses packet losses to detect congestion. When TCP detects packet loss, it realizes that the path is congested and backs off to a lower transmission rate. While it is important for TCP to detect packet loss in a timely manner, large network buffers can store a large number of packets before loss can occur and hence loss detection is significantly delayed. This causes TCP to over-estimate the BDP and consequently send larger bursts of data that fill the large buffers and cause high delays.

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

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

In order to mitigate this problem we present as a second contribution a technique, SABRE (Smooth Adaptive Bit RatE), that enables a video client to smoothly download video segments from the server while not causing significant delays to other traffic sharing the link. Our scheme is based on a simple and effective idea that can be implemented in the application layer of any DASH player. The idea uses a technique to dynamically adjust the flow control TCP window (*rwnd*) in a DASH client. By doing that, we manage to control the burst size going from the server to the client and effectively reduce the average queue size of the home router. We implemented SABRE in the VLC DASH plugin [86] and evaluated it using testbed experiments. Our results show that SABRE can significantly improve queuing delay over traditional On/Off video players.

The rest of the chapter is organized as follows. In section 5.2 we introduce the experimental setup with results showing the buffer bloat effect of DASH video flows. In section 5.3 we show experimentally how Active Queue Management (AQM), often cited as a solution to buffer bloat problems, is not a workable solution in this context. Our SABRE technique is described in section 5.4 along with some evaluation results. We present some results when two clients share the same bottleneck link in section 5.5. We summarize the chapter in section 6.7.

5.2 The buffer bloat effect of ABR flows

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

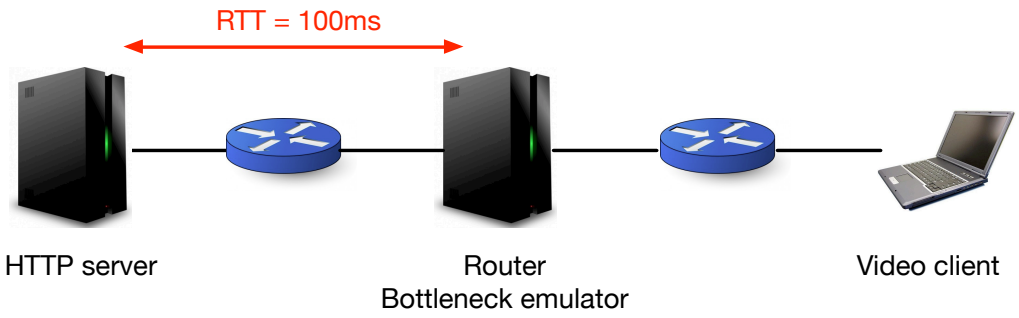


Figure 26: Experimental testbed

5.2.1 Experimental setup

Our testbed is shown in figure 26. The testbed consists of two workstations, two network switches, and a laptop. The workstations and the laptop are running operating system Ubuntu 12.04 LTS. The first workstation on the left acts as a remote HTTP video server and it runs a standard Apache webserver. The webserver hosts a video dataset that was obtained from a published DASH dataset [76]. The dataset includes a video of resolution 1280×1920 that is encoded into six different bitrates ranging from 2.04Mbps to 4.1 Mbps. The second worksation (in the middle) mimics a residential router that connects the client to the remote server. The laptop represents a video client that uses a VLC player equipped with a DASH plugin [86] to stream video from the server.

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

In this experiment we are interested in measuring the queuing delay experienced by VoIP traffic while a DASH streaming session is taking place. We emulate VoIP traffic by using *iperf* [8] to send UDP traffic from the video server to the client. We use UDP traffic of a constant bitrate of 80Kbps with small packets of 150 bytes each, this is similar to Skype

voice traffic [37].

The one way queuing delay of UDP packets is measured in the following manner. *Wireshark* [18] is used to capture UDP traffic at both the router and the client. Assume $t_0^{(R)}$ and $t_i^{(R)}$ are the timestamps when the first and the i^{th} UDP packets were received at the router. Moreover, assume that $t_0^{(c)}$ and $t_i^{(c)}$ are the timestamps when the first and the i^{th} UDP packets were received at the client. Then the queuing delay of the i^{th} UDP packet can be computed using the formula

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

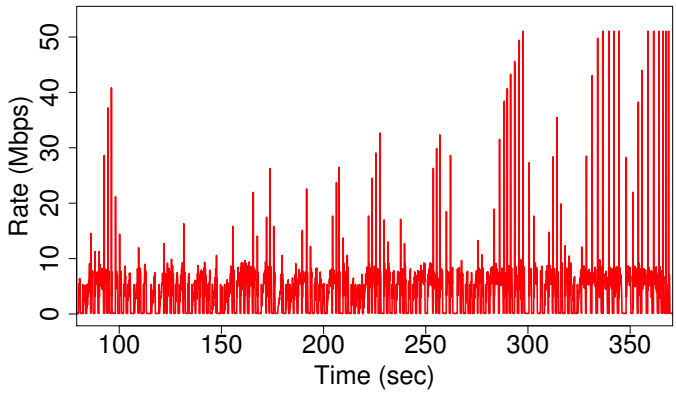
Note that computing the queuing delay in this way does not require synchronizing the clocks of both the client and the router machines.

5.2.2 Measuring buffer bloat

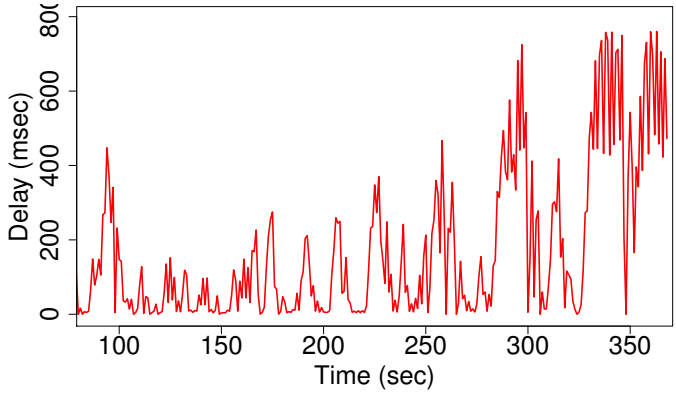
In this experiment we set the capacity of the bottleneck link to $6Mbps$. We first start UDP traffic then after five seconds we start streaming the video. In addition to computing queuing delay as described above, we use *Wireshark* [18] to capture incoming video traffic at the router. This enables us to compute the data rate on the link between the video server and the router, we call it the server link. In figures 27b and 27a we plot the queuing delay of UDP packets and data rate on the server link respectively. In these two figures we have a new sample every 100 milliseconds, meaning that we get 10 samples every second.

We can clearly see from the figures the correlation between the high data rate points in figure 27a and the high queuing delay points in figure 27b. For example, during the time period from $t = 95$ to $t = 100$ we can see in figure 27a two large bursts of data with a rate that exceeds 30Mbps. During the same period we observe that queuing delay approaches $400ms$. The explanation is that the huge bursts of video data fill the queue at the router which causes UDP packets to experience long delays until the buffer gets drained. This behavior repeats multiple times at $t = 255, 290, 335$ and so on.

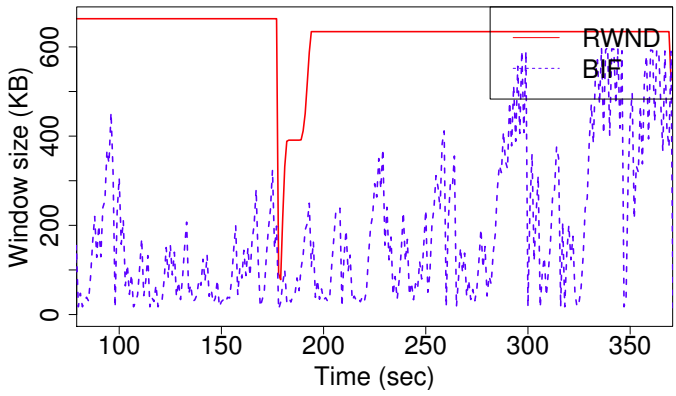
In order to understand why we see these large bursts of data we need to look at the receiver window ($rwnd$) returned by the client to the server, and the congestion window ($cwnd$) computed at the server. This is because the sender rate is governed by the value



(a) Data rate on the server link



(b) Queueing delay of UDP traffic



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

Figure 27: On/Off video client with tail-drop queue at the router

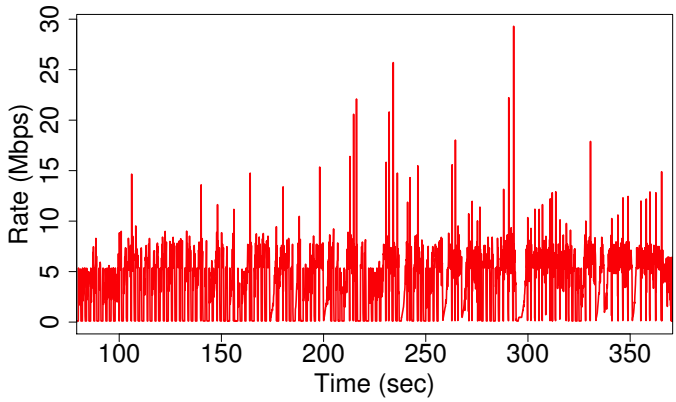
of $\min(rwnd, cwnd)$. We extract the *rwnd* values from the acknowledgement packets going from the client to the server. Since the actual value of *cwnd* can not be obtained without access to the server TCP code, we use the *bytes-in-flight* instead. The bytes-in-flight value is equal to the number of bytes that have been sent by the server but still awaiting acknowledgements from the client. This value is considered to be a lower bound on *cwnd*. Note that when bytes-in-flight equals *zero*, it does not necessarily mean that $cwnd = 0$, it could mean that no traffic was sent during the measurement period ($100ms$).

We plot both *rwnd* and bytes-in-flight (BIF) over time in figure 27c. We observe that *rwnd* is almost a constant value of $650KB$ except at time $t = 180$ when it drops to *zero*. The reason it becomes *zero* is that the persistent TCP connection resets periodically around every three minutes. We suspect this is a default setting in the Apache web server as we observe this behavior repeatedly. We also observe that *rwnd* is always greater than the value of *bytes-in-flight*. This means that the burst size sent by the server is completely controlled by the value of the server congestion window (*cwnd*). On the other hand, the *bytes-in-flight* value varies widely over time and, as expected, the high data rate bursts in figure 27a correspond to high values of *bytes-in-flight* in figure 27c.

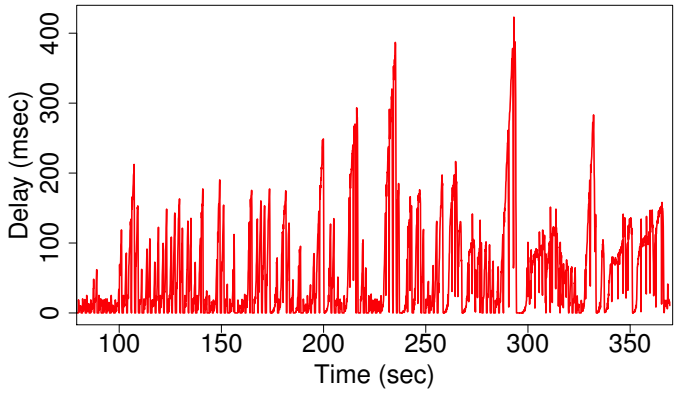
Another observation here is that although the VLC player uses persistent TCP connections, the *cwnd* value does not grow continuously from video segment to the next as one may expect. The reason for that is the On/Off behavior of the video player. During the off period, the TCP connection becomes idle until it starts downloading the next video segment. If this off period is longer than the retransmission timeout (RTO), then according to TCP congestion control specification [15], *cwnd* gets reset to the value of the initial window (IW). The initial window is typically two TCP segments which is between 1000 and 3000 bytes.

5.3 Random Early Detection (RED)

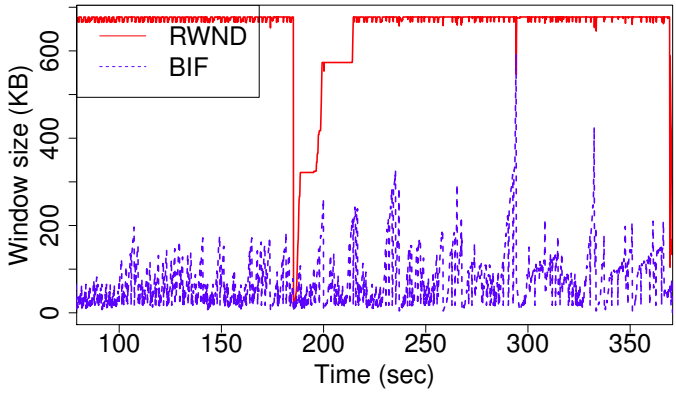
Before presenting our client based technique, we first consider an Active Queue Management technique, specifically RED. RED is a technique that looked like it might be able to solve the buffer bloat problem but has proven to be difficult to manage and tune.



(a) Data rate on the server link



(b) Queuing delay at the router



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

Figure 28: On/Off video client with RED queue at the router

RED [51] computes an average queue size using a weighted moving average. In addition, RED is configured with two parameters, *minimum* and *maximum*. When the average queue size is less than *minimum*, no packets are marked to get discarded. When the average queue size is larger than *maximum*, all incoming packets are marked to get dropped. When the queue size is between *minimum* and *maximum*, the probability of discarding a packet increases linearly with the queue size. Using this policy, RED guarantees that the queue size does not grow much over *maximum*.

We repeated the same experiment in section 5.2 after using the *tc* tool to replace the tail-drop queue with a RED queue. We set the max queue size to $200KB$ with the *minimum* and *maximum* as 20% and 80% of that value respectively. In figures 28a and 28b we plot the data rate on the server link and the queuing delay of UDP packets respectively.

Although we observe from figure 28b that RED manages to reduce queuing delay compared to the tail-drop queue (figure 27b), queuing delay still can be above $150ms$ for a large number of measurement intervals. This is still too much to be acceptable for a VoIP call. The reason queuing delay reach hundreds of milliseconds even when RED is used in the router, is that RED does not prevent the server from sending large bursts of data. This can clearly be seen from the data rate on the server link in figure 28a. As explained in section 5.2, the reason for these large bursts is that *rwnd* usually stays at very large values and *cwnd* occasionally grows to large values as well, which results in sending large bursts of data. This can be seen from figure 28c.

It is worth mentioning here that there could be another configuration for RED with different parameters that could produce better results. However, this is one of the main disadvantages of using RED. Tuning the algorithm to get the best performance is not an easy job [97]. Moreover, finding the set of parameters that would optimize the performance for one type of data flow does not mean that these parameters would work for all other applications. This is why we believe that the solution to the high queuing delay problem is by stopping the server from sending large bursts of data over short periods of time. That is because once the burst is on the wire, there is not much that can be done to prevent the queue from getting full. In the next section we present our solution to this problem.

5.4 SABRE

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

Alternatively, there are multiple ways the client can control the value of *rwnd*. It is important to mention here that the value of *rwnd* is a function of the empty space at the receive socket buffer at any point in time. This means that the size of the socket buffer represents an upper bound on the value of *rwnd*. Hence, one way to control the maximum value of *rwnd* is to set the value of the receive buffer at the client. This can be done using the *setsockopt* call with the option `SO_RCVBUF`. However, the DASH player may not be privileged to make the *setsockopt* call. This usually happens when the DASH player is implemented as a plugin to an existing video player which is the case for the VLC DASH player [86]. Another issue with this approach is that *setsockopt* can be used to set the buffer size only before establishing the connection. This means that a solution that dynamically uses *setsockopt* to set the buffer size will have to reset the connection everytime it needs to modify the buffer size. Resetting TCP connections frequently has many disadvantages. To name a few; it makes tracing network flows more difficult, it can be an overhead on the server, and it may require new key exchange if the flow is encrypted.

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

Below we present SABRE, our technique to control the burst size from the application layer. First, in section 5.4.1 we introduce the technique for the unconstrained bandwidth case when the available bandwidth is constant and higher than the highest video bitrate. After that, in section 5.4.2 we develop the full-fledged scheme to work for the general case

when there is variability in the available bandwidth.

5.4.1 The unconstrained constant bandwidth case

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

HTTP pipelining. In steady state, an on/off video client requests a new video segment every n seconds where n is the segment length. Usually, the client finishes downloading a segment before submitting a request to download the next one. Due to that behavior, the receive socket buffer is always empty when the client starts downloading a new segment. In addition, *rwnd* is typically computed as a function of the available space in the receiver socket buffer. Although the exact function varies among different implementations of TCP, an empty or almost empty receiver socket buffer will always result in a large value of *rwnd*. This will usually cause the large data bursts we saw in section 5.2. In order to mitigate this problem we pipeline requests for multiple video segments. HTTP pipelining allows the client to send multiple `GET` requests to the server before having to wait for them to finish.

The rationale here is that pipelining enough video segments guarantees that the server will send enough data to always keep the client receive buffer full. We dynamically compute the number of segments to pipeline as follows: using the *getsockopt* with option `SO_RCVBUF`, we get the actual size of the receive buffer, call it $rcvbuf$ bytes. For a video segment of length s seconds and bitrate r bps, the average segment size will be $\frac{rs}{8}$ bytes. Hence the number of HTTP requests the client should pipeline is

$$1 + ceil(rcvbuf * \frac{8}{rs})$$

In this formula, the second term is the number of segments to fill the socket buffer and the additional segment is the one being read by the video player.

Controlling download rate at the application. Filling the socket buffer does not guarantee avoiding large data bursts all the time. If the application reads from the socket

buffer at a high rate, this will drain the buffer quickly which will cause *rwnd* to grow to a large value. If we take into consideration large round-trip times between the client and the server, *rwnd* could potentially grow to very large values. In order to solve this problem, the application has to control the rate in which it drains the socket buffer.

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

As one may expect, traditional On/Off DASH players (including the VLC DASH player [86]) call *recv* as fast as possible. This causes the socket buffer to get drained as soon as any data arrives at the client. This in turn causes the value of *rwnd* to be always very large as we observed in figures 27c and 28c. On the other hand, it is important to observe that the video player does not need to read data from the socket buffer at a rate higher than the video bitrate itself. This is why we compute the rate of the *recv* call so that the achieved download rate at the client at any point in time is not much higher than the video bitrate streamed by the client. We call this download rate the $target\_rate$.

In order to prevent *rwnd* from growing large, we distribute the *recv* calls uniformly over the segment download time where the latter is the same as the segment length in seconds. For example, if the $target\_rate$ is r bps, the segment length is s seconds, and the size of the buffer given to the *recv* call is buf bits, then the time between consecutive *recv* calls can be computed using the formula

$$t = \frac{s}{(rs)/(buf)} = \frac{buf}{r} \text{ seconds}$$

This maintains a steady download rate close to r bps for a period of s seconds, while at the same time controlling the growth rate of *rwnd*.

Backoff/refill mode of operation. Remember, however, that a DASH video player estimates the available bandwidth on the path between the client and the server while downloading video segments. The client then uses the estimated bandwidth to decide whether it should switch to a higher or lower video quality or stay at the same video profile.

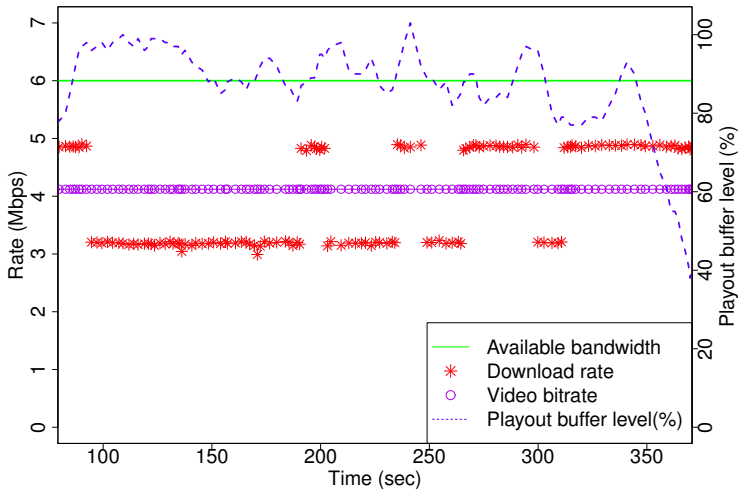


Figure 29: SABRE player: download rate, video bitrate, and level of playout buffer at a constant available bandwidth of 6Mbps

Controlling the rate of the *recv* calls affects the estimated bandwidth by the client. In fact, if the rate of *recv* calls was computed to achieve a $target\_rate$ of r Mbps, we do not expect the client to estimate the available bandwidth to be higher than r Mbps. This means that the client will not be able to switch to higher video bitrates even if there exists enough bandwidth in the path between the server and the client.

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

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

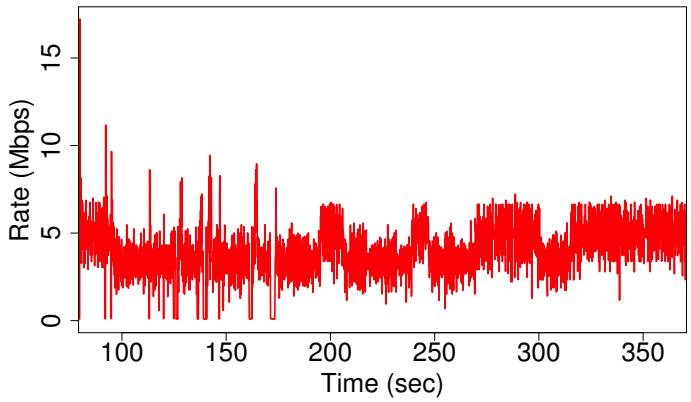
5.4.1.1 Experimental results

We implemented the above technique in the VLC DASH player [86]. In our implementation we set $\lambda = 1.2$ and $\delta = 0.8$. We repeated the experiment we did in section 5.2 while setting the bandwidth of the bottleneck link to 6 Mbps and we stream the video for 6 minutes. We consider the results of the steady state behavior starting from $t = 90$ seconds. Figure 29 shows the computed download rate, the requested video bitrate, and the level of the playout buffer. We can clearly see that the player is switching back and forth between the *backoff* and *refill* modes. For example, at time $t = 95$ the client enters the *backoff* mode and sets its $target\_rate$ to $0.8 \times 4.1 = 3.3$ Mbps. The client stays in this mode until $t = 190$, when the playout buffer drops below 85% then it enters the *refill* mode and sets it $target\_rate$ to $1.2 \times 4.1 = 4.92$ Mbps.

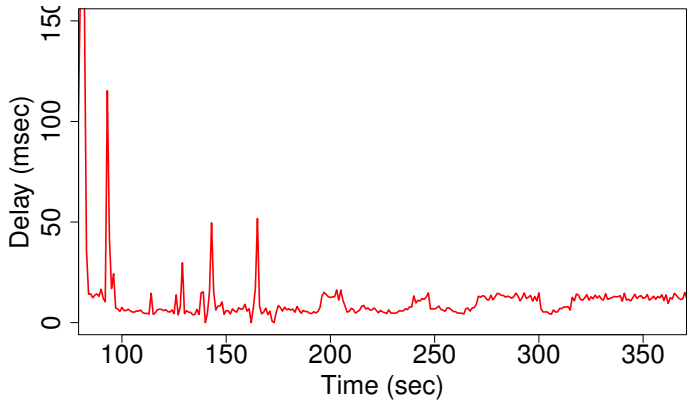
In figure 30 we plot the data rate on the server link, queuing delay, and the values of *rwnd* and bytes-in-flight over time. Compared to figure 27, we can see from figure 30a that bursts of high data rates do not exist anymore and this is why we do not see large queuing delays in figure 30b. Looking at figure 30c, we can clearly see that the value of bytes-in-flight is constantly being pushed down by the value of *rwnd*. We can also see that the value of *rwnd* is now close to 50KB most of the time. This is a huge reduction compared to figure 27c when *rwnd* used to be over 600KB. This is because HTTP pipelining is keeping the socket buffer full all the time which causes the *rwnd* to shrink to this low value.

We can also observe that sometimes *rwnd* grows to large values for short periods of time, this can be seen in figure 30c between times $t = 130$ and $t = 170$. As mentioned earlier, the exact implementation for computing *rwnd* is operating system dependent and it is not clear to us why we observe such behavior. This transient behavior could possibly result in receiving large bursts of data which could increase queuing delay. In section 5.4.2 we present a technique to mitigate this problem and to avoid having large queuing delays even if the socket buffer drops for several seconds.

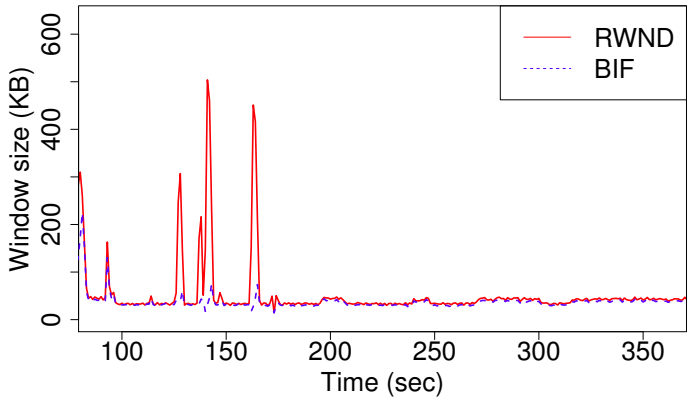
In order to compare the overall performance of the On/Off player versus SABRE in the unconstrained bandwidth case, we plot the CCDF of the queuing delay for both of them in figure 31. In this figure we see that SABRE manages to keep queuing delay below $50ms$



(a) Data rate on the server link



(b) Queuing delay



(c) Receiver window and bytes-in-flight

Figure 30: SABRE video client with tail-drop queue at the router

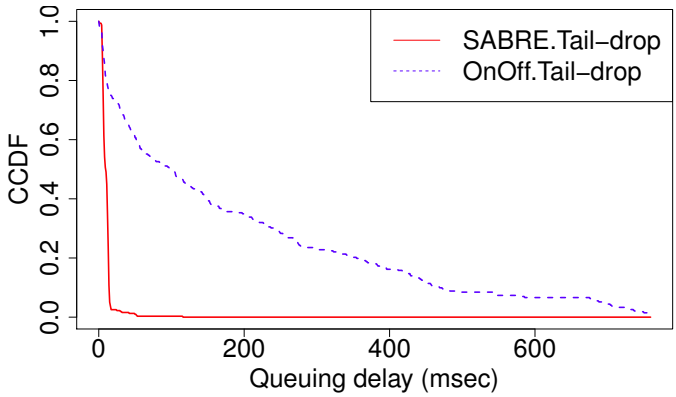


Figure 31: CCDF of queuing delay, On/Off player vs SABRE both using a tail-drop queue

for almost 100% of the time. The On/Off player, on the other hand, causes queuing delay to exceed $100ms$ for about 50% of the time with 15% of the time being over $400ms$.

5.4.2 The general case

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

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

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

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

where $0 < \alpha < 1$ is a smoothing parameter. In our implementation we set $\alpha = 0.8$. Similar to [101], the player then uses BW to decide whether it should switch to a different video

profile or stay at the current one. If R_i is the bitrate of the current segment and $BW < kR_i$, then the client switches to R_{i-1}, the next lower video profile. We use $k = 1.1$ as a slack parameter to compensate for variability in the computed download rate. If R_{i+1} is the bitrate of the next higher video profile and $BW > kR_{i+1}$ then the player switches to R_{i+1}. The rest of this section describes the operation of SABRE in steady state.

As described in section 5.4.1, SABRE pipelines video segment requests to keep the socket buffer full all the time. This means that the client is guaranteed to achieve its objective *target_rate* given that the available bandwidth is higher than that rate. When the network gets congested and the available bandwidth becomes less than *target_rate*, the socket buffer will get drained and the received download rate will be significantly less than *target_rate*. We identify a congestion event based on the condition $BW < \gamma \times target\_rate$, where $0 < \gamma < 1$ – we use $\gamma = 0.85$. When the socket buffer gets drained due to a congestion event, *rwnd* will increase rapidly which will cause queuing delay to increase significantly. This increment in delay reaches several hundreds of milliseconds and can last for several seconds. Once SABRE detects the congestion event, it can reduce its *target_rate* which will result in reducing *rwnd* and hence reduce the queue size. Due to space limitation we do not show results for this case.

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

We compute the occupancy level of the socket buffer every $200ms$. This helps us to react fast when sudden variations happen in the available bandwidth. When the buffer level is detected to be lower than 75%, we temporarily reduce the rate of the *recv* call. Reducing the rate of the *recv* call allows the socket buffer to refill quickly until it gets back to the normal level. Once the socket buffer level gets to 75%, the rate of *recv* is resumed

to its original value. Note that reducing the rate of *recv* calls can result in reducing the video segment download rate $D(i)$ computed by the application. This is because having a lower rate of *recv* calls means that the client will need more time to finish downloading the segment. Depending on how long SABRE has to reduce the rate of *recv*, $D(i)$ can vary from the $target\_rate$ (which can be $1.2R$ in the refill mode, where R is the video bitrate) to θR, where θ is the drop rate of *recv* – we use $\theta = 0.5$.

Drops in the socket buffer level can happen due to two different events; random drops like the ones observed in section 5.4.1.1 and drops due to changes in the available bandwidth. Although SABRE should not react to the first kind, it should reduce the requested video bitrate in the second. SABRE distingueshes between these two kinds of events using the following method. A drop event is considered significant only if it results in a segment download rate $D(i)$ that is less the video bitrate R. If SABRE detects consecutive significant events for a certain period of time, this event is considered a change in the available bandwidth, otherwise it is considered a random drop in socket buffer. In our implementation we set this period to 10 seconds.

When SABRE detects a drop in the available bandwidth it down-shifts to a lower video profile. However, since SABRE always keeps the socket buffer full, it can not estimate the new available bandwidth. Instead, SABRE uses a multiplicative-decrease additive-increase approach to find the best video profile that fits the new available bandwidth. Let R_i be the video bitrate at the time when SABRE decided to down-shift and R_0 be the lowest video profile. Then, when a drop in the available bandwidth is detected, SABRE will follow a multiplicative-decrease behavior and down-shift to the video profile $R_{i/2}$. SABRE then waits for a stabilization period of $wait\_to\_probe$ seconds to see whether or not it needs to down-shift to a lower profile. If SABRE does not detect any further drops in the available bandwidth, it starts probing to see whether it can achieve a higher video profile or not.

SABRE then follows an additive-increase behavior and up-shifts to the next video profile $R_{1+i/2}$. Using the same previous method, SABRE can detect whether or not the available bandwidth is enough to support the new video profile. Note here that detecting a drop in the available bandwidth while streaming a certain video profile is equivalent to detecting an

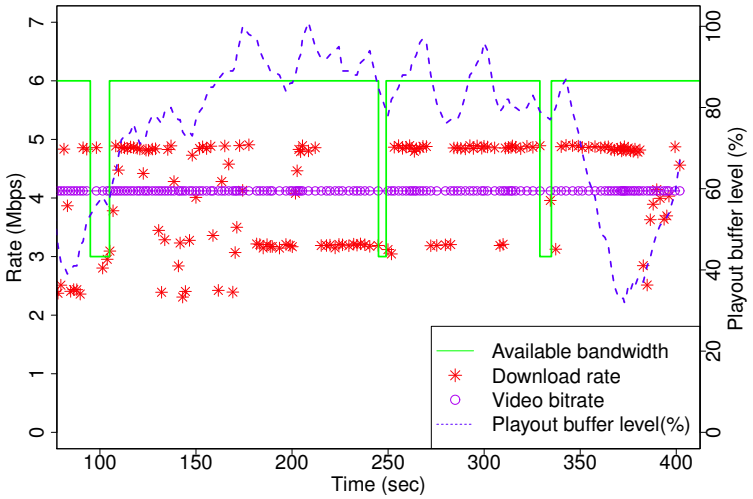
up-shift to a profile that is higher than the available bandwidth. If SABRE manages to up-shift to the higher profile without detecting a drop in the available bandwidth, it reduces the value of *wait_to_probe* to half. The idea here is that the client should probe more aggressively when there is a lower chance of congestion. On the other hand, if SABRE detects that it has to down-shift again to a lower profile, it doubles the value of *wait_to_probe*. In our implementation, we set the default value of *wait_to_probe* to 16 seconds and allow it to reach a maximum of 32 seconds and a minimum of 4 seconds.

5.4.2.1 Experimental results

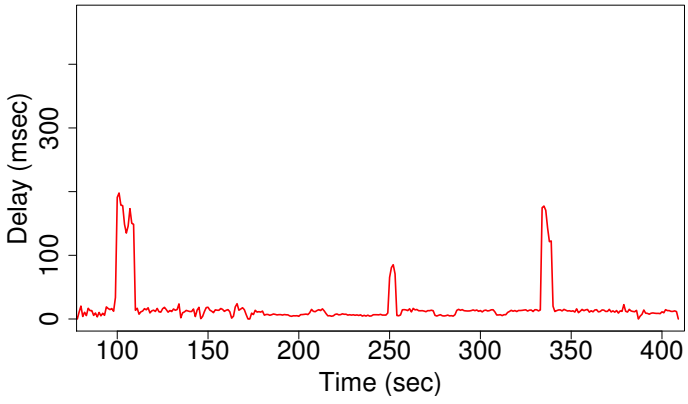
In this section we present results for two cases when the bottleneck link is congested. In the first case, the client experiences short-lived variations in the available bandwidth for only a few seconds. Our objective here is to study the effect of short variations on both the queuing delay and the adaptation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

We repeated the experiment in section 5.4.1.1 with the following changes. We start the experiment with the available bandwidth set to 6 Mbps. The bandwidth of the bottleneck link is set to 3 Mbps at times $t = 95, 245, 330$ for the duration of $10, 4, 6$ seconds respectively. We performed this experiment twice; once using a SABRE client and the other using an On/Off client. Since we are interested in the steady state phase, we do not show results for the initial buffering phase. In figures 32 and 33 we show results for SABRE and On/Off clients respectively.

We can see from figure 32a that SABRE treated the three short variations in the available bandwidth as random drop events and it did not switch to a lower video profile. The behavior of the On/Off client was similar in figure 33a although it switched to lower profiles at time $t = 100$ when the drop in the available bandwidth lasted for 10 seconds. Since the On/Off client uses a moving average to estimate the available bandwidth, it needs to download multiple video segments with the new available bandwidth before it can act accordingly.

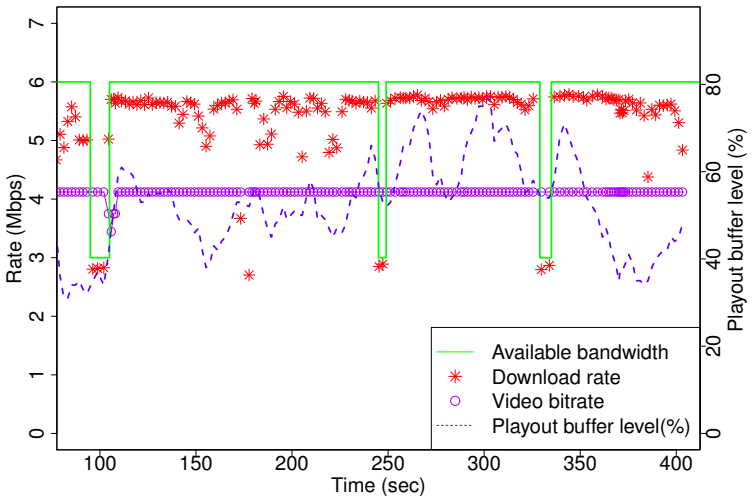


(a) Player adaptation to change in the available bandwidth

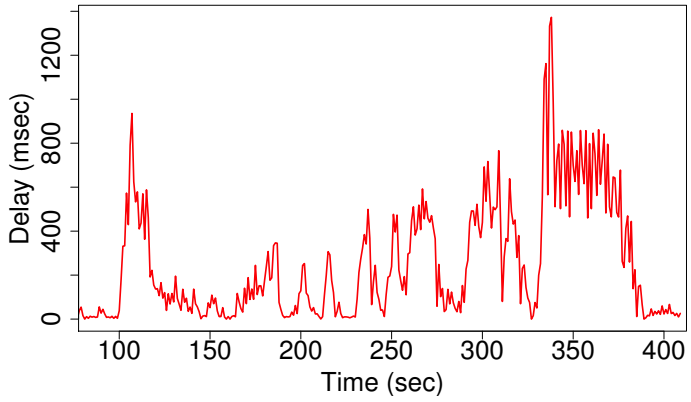


(b) Queuing delay

Figure 32: SABRE player with tail-drop queue and short duration congestion



(a) Player adaptation to change in the available bandwidth



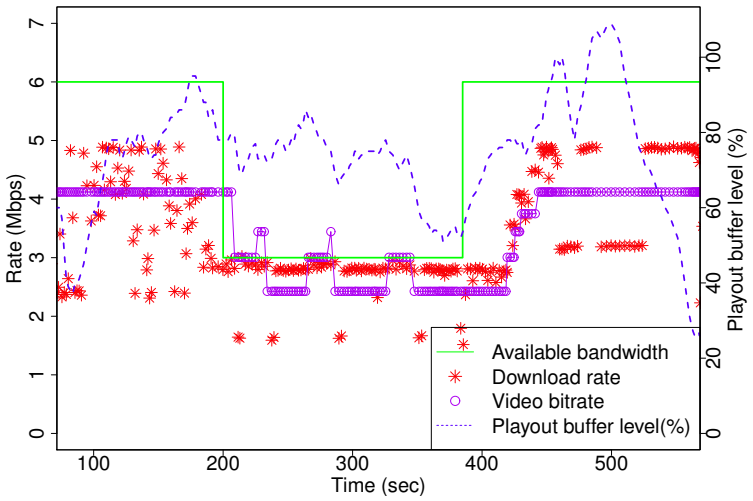
(b) Queuing delay

Figure 33: On/Off player with tail-drop queue and short duration congestion

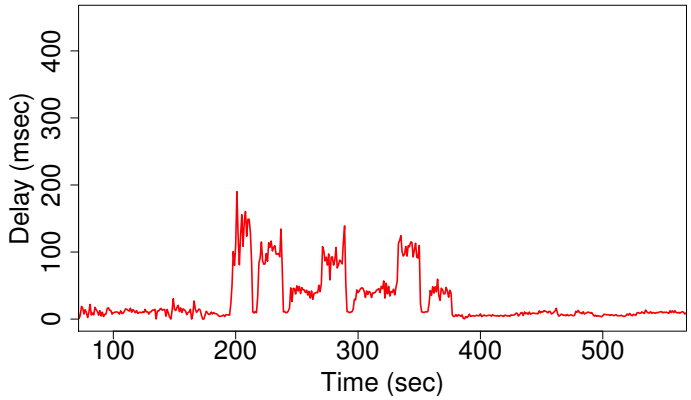
Short-lived variations

In figure 32b we plot queuing delay over time for the SABRE client. We can see that delay is always below $20ms$ except for the three events when the available bandwidth drops to 3 Mbps. In this case, reducing the rate of the *recv* calls prevents large delay spikes from happening, although delay still increased to about $200ms$. This happens in the three events and the lowest effect was at time $t = 245$ when the drop event lasted for only 4 seconds. On the other hand, we can see queuing delay for the On/Off client in figure 33b. It is clear that droping the available bandwidth causes significant increase in delay. This can be seen at times $t = 100, 335$ when queuing delay jumps to $900ms$ and $1300ms$ respectively.

For this case, we repeated the experiment in section 5.4.1.1 with the following changes.



(a) Player adaptation to change in the available bandwidth



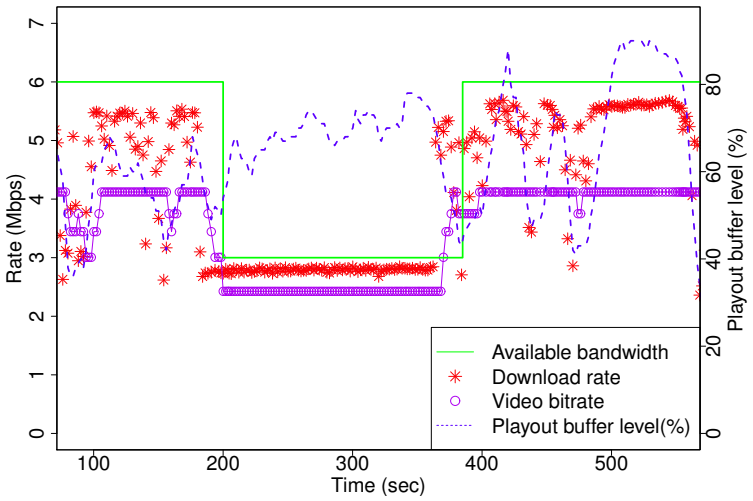
(b) Queuing delay

Figure 34: SABRE player with tail-drop queue and long duration congestion

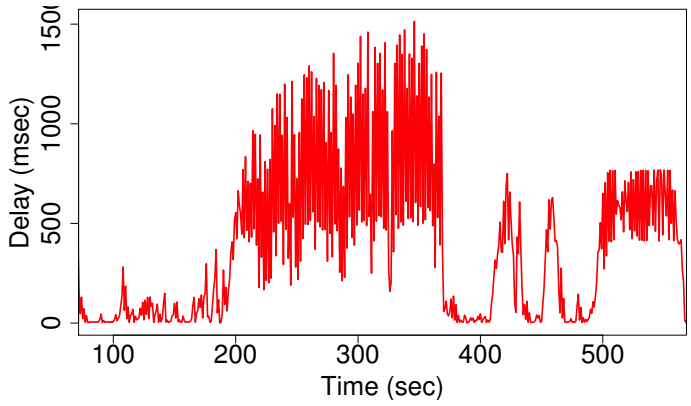
We start the experiment with the available bandwidth set to 6 Mbps from time $t = 0$ to $t = 190sec$. After that, we change the available bandwidth to 3 Mbps from $t = 190$ to $t = 380sec$, then we set it back to 6 Mbps until the end of the experiment. We performed this experiment for both the SABRE client and the On/Off client. We show results for the SABRE and On/Off clients in figures 34 and 35 respectively.

Long-lived variations

In figure 34a we can see that SABRE takes less that 10 seconds to detect a change in the available bandwidth at time $t = 200$. Once change is detected, SABRE applies the multiplicative-decrease policy and down-shifts from $R_5 = 4.1$ Mbps to $R_2 = 3.1$ Mbps. Since the new available bandwidth is 3 Mbps which is very close to R_2, SABRE thinks it can



(a) Player adaptation to change in the available bandwidth



(b) Queuing delay

Figure 35: On/Off player with tail-drop queue and long duration congestion

up-shift to a higher profile. At time $t = 220$ SABRE decides to up-shift to the next profile $R_3 = 3.4$ Mbps. SABRE then detects that the available bandwidth is not enough for the new profile and down-shifts to $R_1 = 2.45$ Mbps at time $t = 230$. After a stabilization period of $wait\_to\_probe = 32$ seconds, SABRE up-shifts to $R_2 = 3.1$ Mbps and then up-shifts again to $R_3 = 3.4$ Mbps. This behavior repeats until the available bandwidth increases to 6 Mbps at time $t = 380$. Since SABRE has to wait for *wait_to_probe* every time before up-shifting to a higher profile, SABRE takes additional time to recover to the highest video profile $R_5 = 4.1$ Mbps. In this case SABRE reached R_5 at time $t = 440$ which means it took an additional 60 seconds after the available bandwidth changed to 6 Mbps.

On the other hand, it took the On/Off player only about 10 seconds after the available

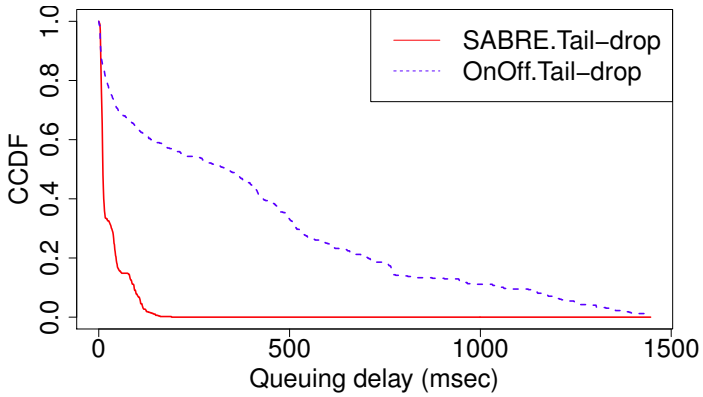


Figure 36: CCDF of queuing delay, On/Off player vs SABRE both using a tail-drop queue and long duration congestion

bandwidth became 6 Mbps to recover to the highest profile R_5 (see figure 35a). However, this comes with a very expensive price in terms of queuing delay. We can see from figure 35b that during the congestion period from $t = 190$ to $t = 380$ queuing delay jumps dramatically over one second while in figure 34b SABRE manages to keep the maximum delay below $200ms$ and about $100ms$ on average.

We also plot the CCDF of queuing delay for both SABRE and the On/Off player in figure 36. It is clear that while SABRE manages to keep queuing delay less that $100ms$ for about 90% of the time, an On/Off player causes queuing delay to exceed $200ms$ about 60% of the time. This delay can be very disturbing to other applications sharing the same bottleneck link with the video flow.

5.5 Two video players

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

In all three experiments we first start one of the two clients, wait for 20 seconds, and then start the second one. Both clients run for 300 seconds. Again, we ignore the first 100

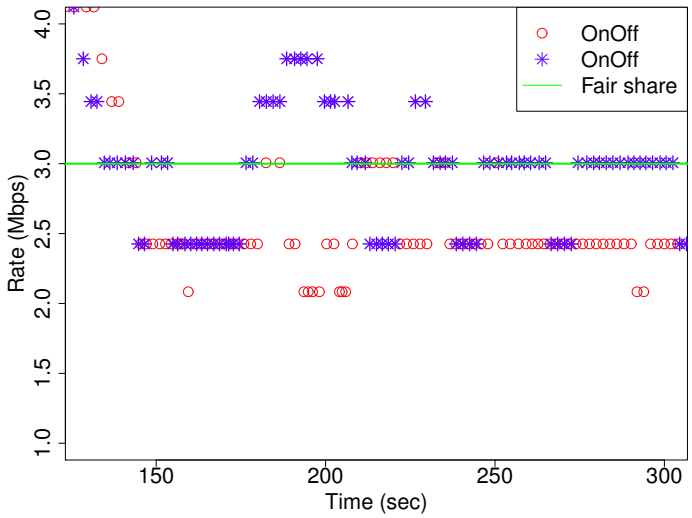


Figure 37: Bitrate adaptation when two On/Off players share a bottleneck link of 6 Mbps

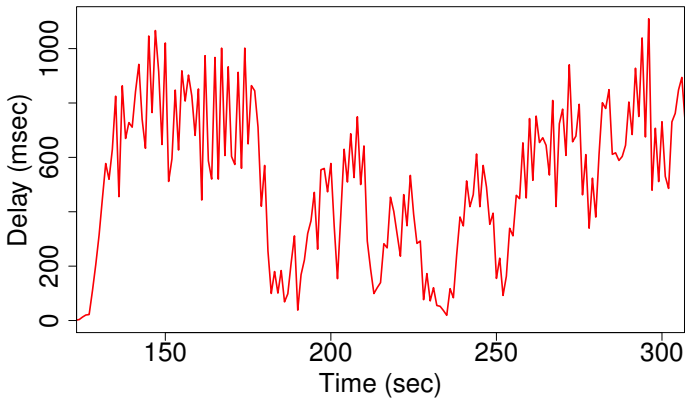


Figure 38: Queuing delay when two On/Off players share a bottleneck link of 6 Mbps

seconds of the experiment since we are only interested in the steady state behavior of the two clients. The bandwidth of the bottleneck link is set to 6 Mbps. Ideally, each client should converge to a video profile that occupies its own fair share of the bandwidth. Since the fair share is 3 Mbps, the ideal video bitrate for each of the two clients should be 2.45 Mbps. Below we present the observed results for all three experiments.

Two On/Off clients. Figure 37 shows the requested video bitrates by the two clients over a period of three minutes. We can observe that sometimes a client can overestimate the available bandwidth. This in turn can lead the client to request a video bitrate that is higher than its fair share. This can be seen at times $t = 180$ and $t = 230$ when one of the

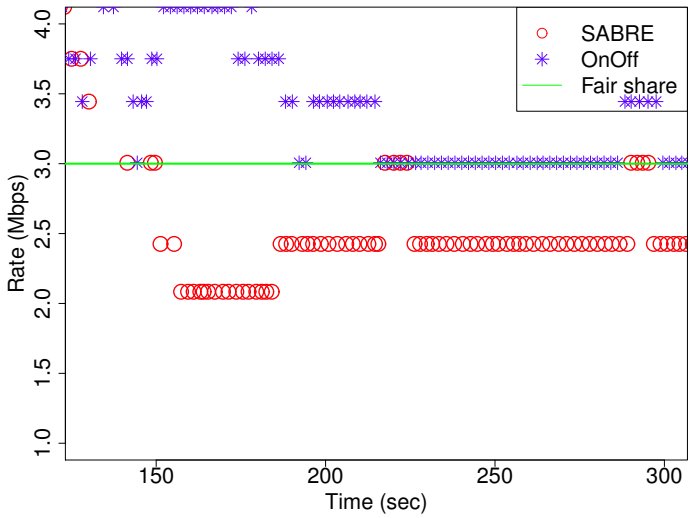


Figure 39: Bitrate adaptation when two players, On/Off and SABRE, share a bottleneck link of 6 Mbps

two clients requests a bitrate of 3.4 Mbps.

This behavior is similar to what was observed in [?]. The reason behind this behavior is the *off* periods of On/Off players. During an `Off` period, the `On` client can overestimate the available bandwidth because it thinks it is the only one using the link. In figure 38 we plot the queuing delay caused by the two On/Off video flows. We can see that adding two flows produces much higher delays than what was observed with a single flow in figure 27b. The delay sometimes reaches one second and is over $500ms$ for about 50% of the time. Having a one way queuing delay close to a second makes it almost impossible to use the bottleneck link for anything else.

An On/Off client and a SABRE client. The interaction between an On/Off client and a SABRE client sharing a bottleneck link is very important to study. The On/Off client always probes the link aggressively for the available bandwidth. On the other hand, SABRE follows a much less aggressive behavior. It is important to study and understand whether one of them can cause performance degradations to the other. In figure 39 we plot the requested video bitrates by both players over time.

We can see that after the SABRE client detects a congestion event it manages to stabilize at a video bitrate of 2.45 Mbps. The client then periodically tries to shift to the next video

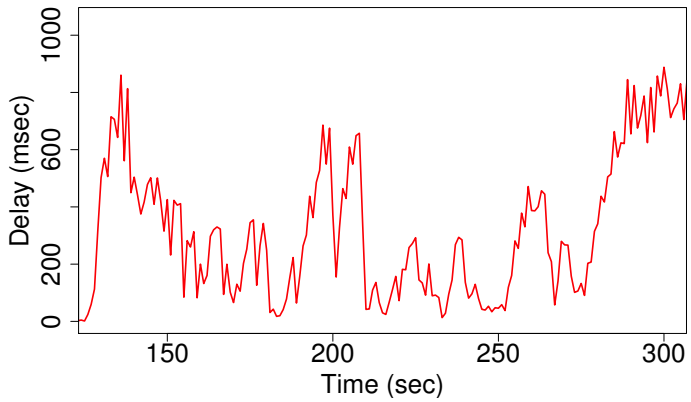


Figure 40: Queuing delay when two players, On/Off and SABRE, share a bottleneck link of 6 Mbps

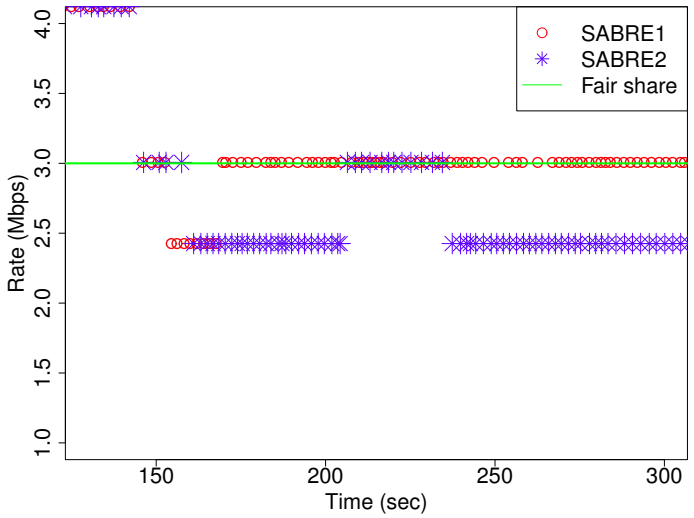


Figure 41: Bitrate adaptation when two SABRE players share a bottleneck link of 6 Mbps

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

In figure 40 we plot the queuing delay caused by the two video flows. We can make two observations from this figure. First, although the delay is high and sometimes reaches $800ms$, the combination of a SABRE player and an On/Off player achieves a lower queuing

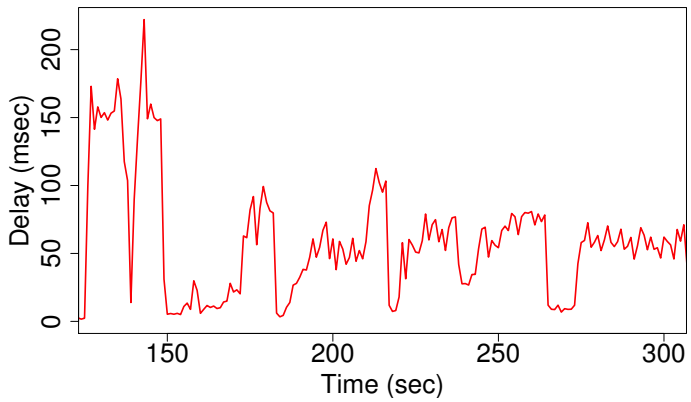


Figure 42: Queuing delay when two SABRE players share a bottleneck link of 6 Mbps

delay than having two On/Off clients. Second, a single On/Off player can cause high queuing delays even if SABRE players are sharing the bottleneck bandwidth with it.

Two SABRE clients. We can see from figure 41 that the two SABRE clients are competing over the video profile $R_2 = 3.1$ Mbps. Since the bottleneck link has a bandwidth of 6 Mbps, it can not accomodate two video flows of 3.1 Mbps each. At time $t = 145$, both clients converged to R_2 but not for a long time. They both detected that the available bandwidth is not enough to accomodate their video profiles and they both down-shifted to $R_1 = 2.45$ Mbps. Later, one of the two clients up-shifts to R_2 and manages to stay there, while the other client periodically up-shifts then down-shifts again. In our experiment, the client that managed to stay at the higher video profile (R_2) had more memory and processing power than the other client. We suspect this is an important factor in deciding which of the two clients will win with the higher video profile.

In order to characterize the buffer bloat effect in the three experiments, we plot the CCDF of queuing delay for the three experiments in figure 43. We can clearly see that two SABRE players are much better for the network than having even a single On/Off player. More specifically, the SABRE players manage to keep queuing delay below $100ms$ for about 95% of the time. On the other hand, whenever one On/Off player gets in the way we get queuing delay over $200ms$ for about 70% of the time.

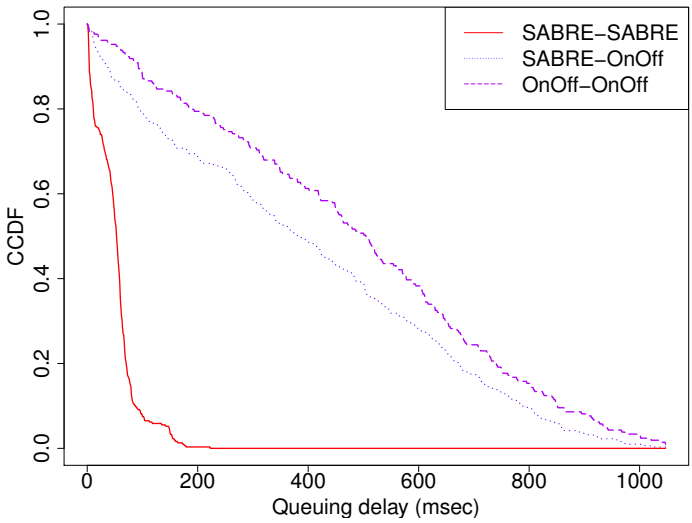


Figure 43: CCDF of queuing delay for two clients sharing a bottleneck link of 6 Mbps. Three cases: two On/Off clients, one On/Off and one SABRE, and two SABRE clients

5.6 Summary

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

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

In addition, we introduce SABRE, a client based technique that can be implemented in the video player to mitigate this problem. We implemented SABRE in the VLC DASH player. Using testbed experiments, we show that SABRE manages to significantly reduce

queuing delays while not affecting the user viewing experience. We also conduct experiments with two clients sharing the home network to study the interaction between SABRE and other traditional players. Our results show that SABRE can coexist with traditional streaming players without having any performance penalties.

CHAPTER VI

ANALYSIS OF ADAPTIVE STREAMING FOR HYBRID CDN/P2P LIVE VIDEO SYSTEMS

6.1 Introduction

Video is widely believed to dominate traffic of the Internet. Although stored video forms the bigger portion of video content on the Internet, live video streaming is growing in volume and importance specially with important events being broadcast over the Internet [7] and new live streaming services becoming free to the public (e.g., the new live service from YouTube [19]).

Content Distribution Networks (CDNs) are currently considered the main pillar of video distribution over the Internet. The purpose of CDNs is to improve user performance in terms of delay and throughput. CDNs accomplish this by deploying multiple nodes, usually called edge servers, distributed geographically in multiple ISPs. Each edge server implements a streaming server, and when a user requests a video stream it is redirected to the closest edge server to start the desired stream. In recent years, CDN-based video streaming has evolved to provide a new adaptive streaming service where a single video can be streamed in multiple qualities at the server and a video player can choose the best quality that fits the condition of the Internet connection of the user. Although much work has been done on adaptive streaming over the years [71, 88], it was not commercially popular until the widespread adoption of *HTTP adaptive streaming* technology which is now being used by many video streaming providers (e.g., Microsoft Smooth Streaming, Netflix, Adobe).

Another important source of video traffic on the Internet is Peer-to-Peer (P2P) networks. Some P2P video systems have recently succeeded in attracting significant numbers of users [58, 59, 122]. According to [42], the volume of P2P TV monthly traffic exceeded 280 peta bytes in 2009. Some adaptive streaming techniques have been proposed in P2P systems such as [99] and [89]. These approaches use layered streaming as opposed to HTTP streaming.

Although layered streaming has existed for a long time, its complicated design and need for high processing power specially at the clients does not make it attractive for major commercial video providers.

Despite the popularity and success of CDN-based systems, some concerns arise about their cost-effective scalability specially when supporting high quality videos to a large population of users. In order to address these issues, some recent work has proposed hybrid streaming systems that combine CDNs and Peer-to-Peer technology [59], [57]. These systems promise to achieve the scalability of P2P networks and the desired low delay and high throughput of CDNs. LiveSky [57] is an operational commercial live streaming system with more than ten million users that adopts the hybrid CDN-P2P approach. Although hybrid systems have a significant potential for providing an attractive video streaming scheme, adaptive streaming has not been extensively explored in such systems.

Designing and operating adaptive hybrid streaming systems is very challenging. By definition, these systems come with two degrees of freedom in their operation; one is the *adaptive* property of the system where users can switch among different streams of different qualities for the same video, and the other one is the *hybrid* operation mode which means users may receive data either from the server or from other peers streaming the same video. That said, two decisions are very critical in the design of any adaptive hybrid streaming system. The first one is the bitrate adaptation strategy which specifies how different bitrates are assigned to different users while maximizing the overall user satisfaction. The second is defining the operational guidelines a system can use to switch between the CDN and the P2P modes while efficiently utilizing both server and peer upload capacity. Another challenging issue is the interaction between these two decisions and understanding how they affect each other.

In this chapter we present an analysis of adaptive streaming in a hybrid live video system with the goal of providing answers to these two design questions and studying the interactions between them. We model a system that adopts the *HTTP adaptive streaming* technology which makes it very easy to integrate into real CDN-based systems. We first present a stochastic fluid model to the hybrid streaming system with a single video bitrate

and we obtain a lower bound on the number of users that should receive the stream from a CDN server in order to be able to support delivering that video stream to other users. This lower bound can be described as the switching point between the CDN and the P2P modes. We then extend this analysis to the adaptive streaming case with multiple video bitrates. We model adaptive streaming as a linear optimization problem to obtain the best bitrate adaptation strategy. In order to compare our results to CDN-based adaptive systems, we also derive results for these systems similar to what we did to the hybrid system. We developed a discrete event simulator and used simulations to validate our analysis. Our results show that adaptive streaming in hybrid systems can significantly improve the ability to satisfy more users with higher video bitrates over adaptive CDN-based systems. We also show that adaptive hybrid systems can lead to significant savings in CDN resources as opposed to CDN systems.

Related work. P2PLive [58] is one of the most famous P2P TV systems in China where users can play tens of live video channels and hundreds of on-demand movies. Multiple adaptive streaming techniques have been proposed in P2P streaming systems. For example, the work in [99] uses layered video encoding to adaptively deliver different layers of the video to clients. Another approach was used in [89] where network coding is used to make SVC more feasible in adaptive streaming. Some recent work has shown the potential benefits of using hybrid video-on-demand systems [59]. Using a nine-month trace from the MSN video service, the work in [59] shows that a hybrid CDN/P2P system could significantly reduce server bandwidth costs. The most relevant work to ours is LiveSky [57]. LiveSky is a hybrid live streaming system, however it does not support adaptive streaming. Some work has been done also in analyzing hybrid streaming systems [123], [78], however, none of them studies adaptive streaming.

The rest of this chapter is organized as follows. In section 6.2 we present a description of the system architecture and the main goals of our analysis throughout the chapter. We present analysis for the single bitrate system in section 6.3 then in section 6.4 we present analysis for the adaptive streaming system with multiple bitrates. We validate our analysis through simulation in section 6.5. Evaluation results of a case study are presented in section

6.6. Finally, we conclude the chapter in section 6.7.

6.2 System description

In this section we briefly describe how video streaming works in CDN-based systems, then we describe the components of the hybrid system and how they interact with each other.

A content distribution network consists of a set of core servers and a set of edge (surrogate) servers [46]. Core servers are responsible for managing the CDN, saving the content being distributed and when needed forwarding content to edge servers to serve requesting clients. Edge servers are the ones that actually serve client requests and they are usually distributed geographically to bring content close to as many users as possible. This helps to increase the CDN's system scalability and to improve response time for user requests.

User requests are forwarded to different edge servers in a CDN in the following manner. When a web client tries to retrieve an object for a web page it first uses DNS to resolve the server name of the URL of the object. If a CDN is used to distribute that web object, a popular way to direct a client to the best edge server is using DNS redirection. In that case, the requesting client is directed to an authoritative DNS name server that belongs to the CDN which in turn redirects the client to an edge server [73]. For a detailed description of how DNS redirection is done, the reader is referred to [113].

Broadcasting live video over CDNs is slightly different than distributing stored content. In the latter case, content is stored at edge servers hard drives and different caching techniques may be used to decide how to replace old (outdated) content with new content depending on client requests. On the other hand, in the live video case, a CDN usually has an entry point to live video where the video is encoded into a single bitrate or multiple bitrates and then the resulting streams (of different bitrates) are forwarded to edge servers when needed [31]. Note that multiple bitrates are used when adaptive streaming is enabled where clients are allowed to switch among streams of different qualities according to network and server conditions.

It is important to mention here that adaptive streaming in this chapter does not mean that clients switch among different streams according to changes in their download rates.

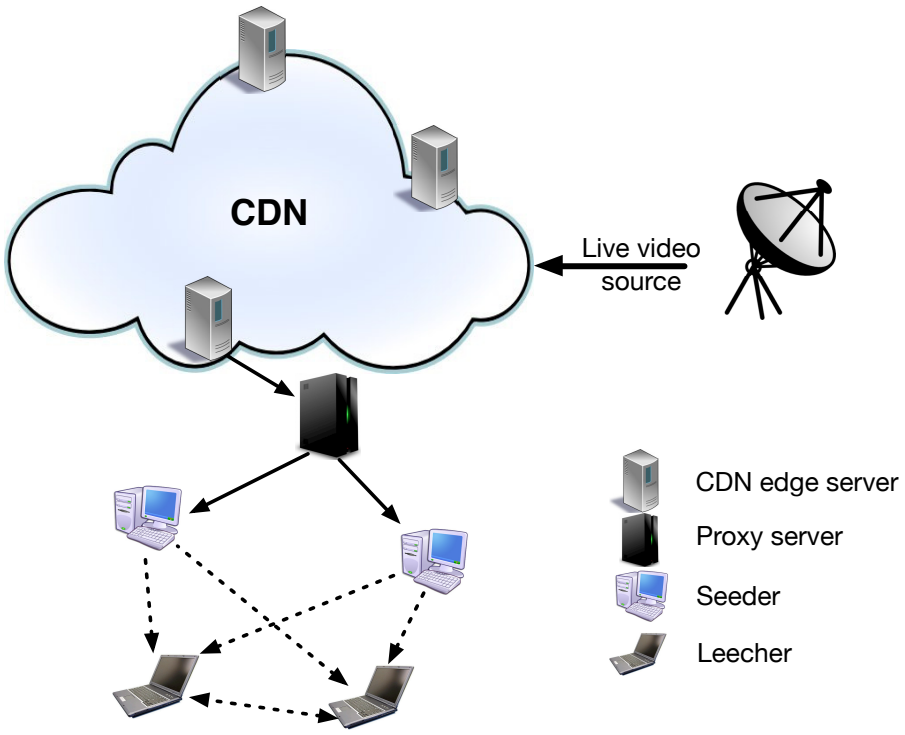


Figure 44: System architecture

Client download rate is assumed to be variable among clients but constant over time which is true for wired connections that are not shared by multiple users. Changes in user download rates over time can be modeled as a composition of two events: departure of a user with the old bitrate, and the arrival of a new user with the new bitrate. Adaptivity in this chapter comes from the fact that a CDN server has a limited capacity and it may not be able to satisfy all client requests. In that case, the server can follow one of two different strategies: 1)Serve clients with their desired bitrates in a first come first serve manner and reject any new clients when the server capacity is fully consumed. 2)Try to accommodate as many clients as possible by considering the possibility of delivering streams of lower bitrates to some clients in order to save some of the server capacity. Based on user arrival rates, video viewing duration, and the distribution of requested different video birates, we will show how the server can compute the best adaptation strategy.

A hybrid CDN/P2P streaming system usually uses the above described infrastructure of CDNs with the addition of new mechanisms for integrating peer coordination into the

system. We propose the system architecture in figure 44 as our hybrid streaming system. We assume that each edge server in the CDN will have a proxy server attached to it. The proxy server can be either a software entity on the same edge server machine, or a separate machine that is connected to the edge server. When a client requests to start a live video stream, the request is directed to the proxy server through the edge server. The proxy server is responsible for the following tasks.

- Compute the best bitrate adaptation strategy.
- Keep a directory service in which it maintains a mapping between all video streams currently being transmitted and clients that are currently connected to the proxy.
- For any new streaming request, the proxy will have to make two decisions. The first one is to allocate a bitrate to the client based on the computed adaptation strategy; this bitrate could be the one originally requested by the client or could be a different, typically lower, bitrate. The second decision is whether to serve that request directly (CDN mode) or to redirect the requesting client to other clients (peers) that are streaming the same video (P2P mode). A client served in the CDN mode is called a *seeder* while a client in the P2P mode will be called a *leecher*.

Although peer selection strategies are not the focus of this chapter, it is important to emphasize here that we assume that peers form random mesh networks. In the next sections we will show how the proxy server can make such decisions in a way to optimize the system performance.

6.3 Single rate system model

In this section we present a model for hybrid streaming using a single video bitrate, we present the adaptive streaming model with multiple bitrates in the next section. The model we develop is a stochastic fluid model similar to the one used in [74]. In our analysis, we do answer the following question to the single bitrate system: when the proxy server receives a new streaming request, should the server treat the incoming client as a *seeder* or as a

leecher. In other words, we aim to find out how many *seeders* will be sufficient to provide a live stream to a certain number of *leechers*.

We analyze the system for the theoretical *unconstrained* case when peers can have an unlimited number of connections with other peers and for the more realistic *constrained* case when peers can only have a limited number of incoming and outgoing connections. Moreover, for these two cases we develop the analysis for systems with *churn* when clients come and go and also for *churnless* systems when the number of clients is fixed. Throughout the rest of this section we assume that the proxy server is providing a video stream with bit rate r bps, and the upload capacity of the proxy is C_{proxy} (bps).

For churn analysis, we assume that users join the system at random points in time, stay in the system for a random period, then leave the system. Previous studies of client behavior in real live streaming systems show that client arrival follows a Poisson process over short time scales [111]. Considering that our analysis can be applied to the system over short time scales, it is reasonable to assume that user arrival follows a Poisson process with rate λ. Users stay in the system for a period of time that follows a general probability distribution with mean $1/\gamma$. Define $N(t)$ as the number of users in the system at time t, then it is clear that $N(t)$ can be represented as the number of customers in a $M/G/\infty$ queue [72].

6.3.1 Unconstrained churnless system

Denote n_l as the number of leechers and n_s as the number of seeders in the system. Also define $u_i^{(l)}$, $u_j^{(s)}$ as the upload rates of leecher i and seeder j respectively. Note that the following two conditions must hold: $C_{proxy} \geq n_s r \quad , \quad \sum_{j=1}^{n_s} u_j^{(s)} \geq r$. The first inequality represents the server capacity constraint and the second one is necessary to guarantee that the set of seeders have the minimum upload capacity to support the video bitrate. In this system, the maximum achievable streaming rate for each client r_{max} is given by

$$r_{max} = \min\{U_s, \frac{U_s + \sum_{i=1}^{n_l} u_i^{(l)}}{n_l}\} \tag{1}$$

where U_s is the total upload rate of all seeders and $U_s = \sum_{j=1}^{n_s} u_j^{(s)}$. The proof of this result can be found in [74].

For analysis purposes, we assume that all leechers have the same upload rate of u_l and all seeders have the same upload rate of u_s. Alternatively, u_l and u_s can be considered as the average upload rates over all leechers and seeders respectively. We also assume that $r > u_l$ which means that the average upload rate of a single leecher is not enough to support sharing the whole stream with other peers. In that case r_{max} can be reduced to $r_{max} = \min\{n_s u_s, \frac{n_s u_s + n_l u_l}{n_l}\}$. When the system is in steady state condition, is it reasonable to assume that $n_s u_s > \frac{n_s u_s + n_l u_l}{n_l}$, then we conclude that a churnless system can support a streaming rate of

$$r \leq \frac{n_s u_s + n_l u_l}{n_l} \tag{2}$$

which gives the lower bound $n_s \geq \frac{n_l(r-u_l)}{u_s}$ on the number of seeders to support bitrate r.

6.3.2 Unconstrained system with churn

As we explained earlier, $N(t)$ is the total number of users in the system at time t (including seeders and leechers) and N follows a Poisson distribution with rate $\rho = \lambda/\gamma$.

We now compute the probability that the system will be able to support a streaming rate r in case of node churn. In order to do that and to simplify the analysis we assume that node churn happens only in leecher nodes. This means that the number of seeders in the system are assumed to be constant and only leechers arrive to and leave the system. This can be done by using the following simple admission policy: 1) the system starts admitting all new arrivals as seeders until n_s clients have arrived. 2) All new arrivals after that point are admitted as leechers. 3) When a seeder leaves the system, one of the leechers is randomly selected by the proxy server and is promoted to become a seeder. Following this policy will always keep the number of seeders to n_s, and then the random variable $N(t)$ will represent the number of leechers in the system at time t. Now, we can compute the probability of supporting a bitrate r as following

$$\begin{aligned} P(\text{support bitrate } r) &= P(r \leq \frac{n_s u_s + N u_l}{N}) \\ &= P(N \leq \frac{n_s u_s}{r - u_l}) = F(\frac{n_s u_s}{r - u_l}) \end{aligned}$$

where $F(w) = \sum_{x=0}^{w} \frac{e^{-\rho}\rho^x}{x!}$. We know that for large ρ we can approximate the Poisson distribution with a Gaussian distribution with mean $\mu = \rho$ and standard deviation $\sigma = \sqrt{\rho}$.

Hence, we can compute the probability of supporting a stream of rate r as

$$\begin{aligned} P(\text{support bitrate } r) &= P(\frac{N-\rho}{\sqrt{\rho}} \leq \frac{\frac{n_s u_s}{r-u_l} - \rho}{\sqrt{\rho}}) \\ &= \Phi(\frac{\frac{n_s u_s}{r-u_l} - \rho}{\sqrt{\rho}}) \end{aligned}$$

where $\Phi(z)$ is the cumulative distribution function of the standard Normal random variable. Now let $\phi_{1-\alpha}$ be a positive real number such that $\Phi(\phi_{1-\alpha}) = 1-\alpha$, then a sufficient condition to guarantee supporting bitrate r with confidence $(1-\alpha)\times 100\%$ is $\left(\frac{n_s u_s}{r-u_l} - \rho\right)/\sqrt{\rho} \geq \phi_{1-\alpha}$ which gives the following lower bound on the number of seeders

$$n_s \geq \frac{(\phi_{1-\alpha}\sqrt{\rho} + \rho)(r-u_l)}{u_s} \tag{3}$$

6.3.3 Constrained churnless system

Now we consider a realistic P2P client configuration where each client has a limited number of inbound and outbound connections. We define these limits as following.

- S_{in} is the maximum number of incoming connections a seeder can accept. Each one of these connections should have a leecher on the other end of the connection. Note that data will only be flowing from the seeder to leechers in these connections.

- Y_{in} is the maximum number of incoming connections a leecher can accept. Again, each one of these connections should have another leecher on the other end of the connection. Data should flow in both directions between leechers.

- Y_{out} is the number connections a leecher can initiate, where connections can be initiated to either seeders or other leechers. We assume that there is no limit on the number of connections a leecher can initiate which means the bottleneck is in the number of connections that actually get established. This number is mainly controlled by the two parameters S_{in}, Y_{in}.

As in [92], we define η as the *efficiency* of the P2P protocol which can be computed as the probability of any leecher finding new content at other leechers when they establish a connection. It was shown in [115] that BitTorrent efficiency can exceed 0.9 if the file

has more than ten pieces. It is clear that η is a function of many parameters of the P2P protocol specially the algorithm used for data exchange among peers (e.g. rarest piece first in BitTorrent protocol). The P2P protocol efficiency means that a leecher has an effective upload rate of ηu_l. Denote d as the average download rate for any leecher in the swarm, then d can be computed as

$$\begin{aligned} d &= \sum_x E[d|\text{leecher is connected to } x \text{ seeders}] \times Pr\{x\} \\ &= \sum_x \left(\frac{x u_s}{S_{in}} + \frac{(Y_{out} - x)\eta u_l}{Y_{in}} \right) \times Pr\{x\} \\ &= \frac{Y_{out}\eta u_l}{Y_{in}} + \left(\frac{u_s}{S_{in}} - \frac{\eta u_l}{Y_{in}} \right) \sum_x x Pr\{x\} \end{aligned} \tag{4}$$

Note that $\sum_x xPr\{x\}$ is the average number of seeders connected to each leecher which can be approximated by the value $n_s S_{in}/n_l$. Note also that when $n_l \geq n_s$, we can calculate an approximate value for the number of connections each leecher can establish by $Y_{out} = (n_s S_{in} + n_l Y_{in})/n_l$. Substituting these two expressions in equation 4, then d can be reduced to

$$d = \frac{n_s u_s + \eta n_l u_l}{n_l} \tag{5}$$

Since d can be considered the average bitrate that can be supported by the system, then the number of seeders sufficient to support that bitrate is $n_s = n_l(r - \eta u_l)/u_s$. The expression in equation 5 is interesting in two aspects. First, the average leecher download rate is not directly related to the constraints of the system, namely the maximum number of uploading connections for both seeders and leechers. Second, comparing the above expression to the maximum bitrate that can be achieved in the unconstrained churnless system in equation 2 we can see that the only difference is η, the P2P protocol efficiency. This is intuitive because the difference between the unconstrained and the constrained systems is in the ability to use the upload capacity of leechers efficiently with a limited number of connections which is represented by the efficiency of the P2P protocol.

6.3.4 Constrained system with churn

Since we only have an estimate of the average bitrate that can be supported by a constrained churnless system, not an upper bound as we had with the unconstrained system, we will

develop our analysis to obtain a confidence interval for the number of seeders that should be sufficient for supporting a bitrate r. Since we know the average bitrate that can be supported in a churnless constrained system from equation 5, we can now compute the probability of supporting a range of bitrates around r in a system with churn as follows

$$\begin{aligned} P(r) &= P(\frac{n_s u_s + \eta N u_l}{N} - \epsilon \leq r \leq \frac{n_s u_s + \eta N u_l}{N} + \epsilon) \\ &= P(\frac{n_s u_s}{r - \eta u_l + \epsilon} \leq N \leq \frac{n_s u_s}{r - \eta u_l - \epsilon}) \end{aligned} \quad (6)$$

As we did earlier, N can be approximated by a Gaussian random variable with mean ρ and variance ρ. In order to compute a $(1-\alpha) \times 100\%$ confidence interval for N we set the following conditions

$$\frac{\frac{n_s u_s}{r-\eta u_l+\epsilon} - \rho}{\sqrt{\rho}} \leq -\phi_{1-\alpha/2} \quad , \quad \frac{\frac{n_s u_s}{r-\eta u_l-\epsilon} - \rho}{\sqrt{\rho}} \geq \phi_{1-\alpha/2}$$

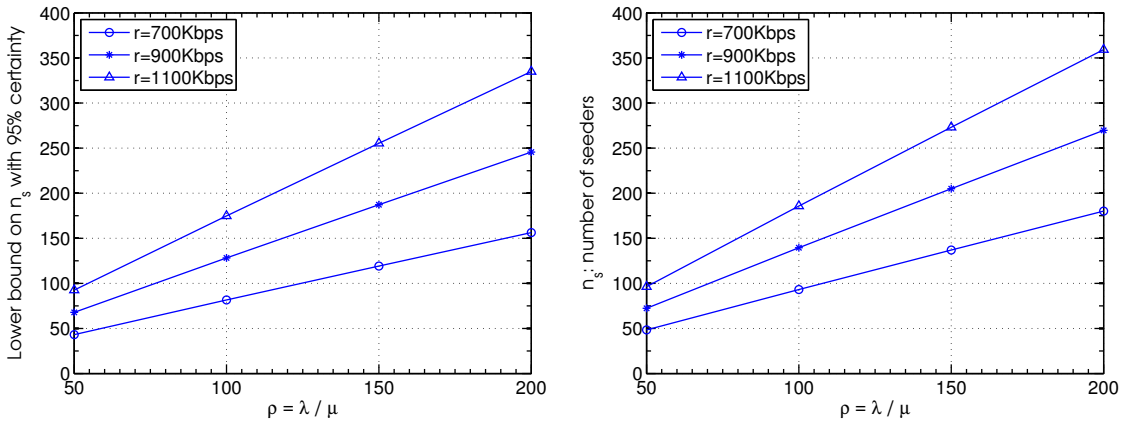
We define $\hat{\phi} = \phi_{1-\alpha/2}$, then we get the following confidence interval for the number of seeders

$$\frac{(\rho + \hat{\phi}\sqrt{\rho})(r - \eta u_l - \epsilon)}{u_s} \leq n_s \leq \frac{(\rho - \hat{\phi}\sqrt{\rho})(r - \eta u_l + \epsilon)}{u_s}$$

It is important to mention here that this inequality generates valid intervals only for ϵ values that satisfy the condition $\epsilon \geq \frac{\phi_{1-\alpha/2}(r-\eta u_l)}{\sqrt{\rho}}$. We can see that ϵ is inversely proportional to ρ which means that the higher client arrival rates and the longer clients stay in the system, the lower ϵ becomes. Lower values of ϵ mean a smaller interval in inequality 6 which yields a higher guarantee that the number of seeders given by the above interval will be sufficient for supporting the bitrate r. In figure 45 we plot the lower bound of the number of seeders against ρ for both constrained and unconstrained systems with node churn. These plots assume a 95% confidence interval on the number of seeders sufficient to support bitrate r. We can observe that the difference in the number of seeder sufficient to support different bitrates is not large.

6.4 Adaptive hybrid live video streaming

In the previous section we assumed that the CDN proxy server has a single bitrate for the live stream, in this section we consider the case when the proxy has multiple bitrates of



(a) Unconstrained system

(b) Constrained system

Figure 45: n_s vs ρ for different video bitrates for systems with churn, $\alpha = 0.05$

the same live video and clients try to get the best stream according to the quality of their Internet connection. Recall that adaptive streaming in this chapter comes from the fact that the CDN proxy server has a limited capacity and it may not be able to satisfy all client requests, note here that by clients we mean only *seeders* because they are the only clients that actually receive data from the proxy. In that case, the proxy will try to accommodate as many clients as possible by considering the possibility of delivering streams of lower bitrates to some clients in order to save proxy capacity.

In this section we answer the following questions about the operation of the adaptive streaming strategy:"which clients should be downgraded to streams of lower bitrates?", "what should these new lower bitrates be?", "how to get an optimal allocation of bitrates to clients while minimizing client downgrading?", and "does the adaptive solution always exist?" In order to answer these questions we formulate the adaptive streaming strategy as a linear optimization problem. We assume that when a client connects to the proxy it has a good estimate of its available bandwidth and then it requests the stream with the best bitrate accordingly. The proxy will then try to give each client the bitrate it requested or if necessary will give the client a lower bitrate, we call the difference between these two bitrates *client dissatisfaction*. The objective of our formulation is to minimize total *client dissatisfaction* over all clients.

6.4.1 Unconstrained case

Denote $r_1, \ldots, r_R$ as the different bitrates provided by the CDN proxy and assume that $r_1 > r_2 > \ldots > r_R$. Also denote n_{l_i} as the number of leechers that request a video stream of bitrate r_i and n_{s_i} as the number of seeders that receive a stream of bitrate r_i where $i = 1, \ldots, R$. Define x_{ij} as the fraction of clients that request bitrate r_i but receive bitrate r_j where $j = i, \ldots, R$ and $\sum_{j=i}^{R} x_{ij} = 1$. When $x_{ii} = 1$, this means that bitrate r_i will be delivered to all clients that requested that rate and none of them will be downgraded to a lower bitrate.

Churnless system. We know from equation 2 for the unconstrained churnless system that the relation between the number of seeders and number of leechers for each bitrate r_i can be written as $n_{s_i} u_s \geq n_{l_i}(r_i - u_l)$. Now the adaptive streaming problem can be formulated as the following linear optimization problem

$$\min \sum_{i=1}^{R} \sum_{j=i}^{R} x_{ij} n_{l_i} (r_i - r_j) \tag{7}$$

subject to: $\sum_{j=i}^{R} x_{ij} = 1$, $0 \leq x_{ij} \leq 1$ for $i = 1, \ldots, R$

$$n_{s_i} u_s \geq \left(n_{l_i} x_{ii} + \sum_{k=1}^{i-1} n_{l_k} x_{ki} - n_{s_i} \right) (r_i - u_l) \tag{8}$$

$$\sum_{i=1}^{R} n_{s_i} r_i \leq C_{proxy} \tag{9}$$

As we mentioned previously, we would like to minimize the total client dissatisfaction which is represented by the *min* objective function above. It is interesting to observe that minimizing client dissatisfaction is equivalent to maximizing *inter-client fairness* defined in [70]. Inter-client fairness is a measure of "utility" acquired by clients in the system, where greater utility means more user satisfaction. Fairness for a single client is defined as the ratio of the delivered bitrate to the actual requested bitrate. Inter-client fairness is defined as the weighted average of client fairness over all clients. To see how we make this observation, we rewrite the objective function as following

$$\sum_{i=1}^{R} \sum_{j=i}^{R} x_{ij} n_{l_i} r_i (1 - \frac{r_j}{r_i}) = \sum_{i=1}^{R} n_{l_i} r_i - \sum_{i=1}^{R} r_i \left(\sum_{j=i}^{R} n_{l_i} x_{ij} \frac{r_j}{r_i} \right)$$

The term $\sum_{i=1}^{R} n_{l_i} r_i$ has no variables and hence could be removed from the objective function. In the latter term, the weighted sum of ratios r_j/r_i can be normalized to get *inter-client fairness* as in [70]. Finally, minimizing this sum with a negative sign is equivalent to maximizing inter-client fairness.

Inequality 8 represents the condition for the number of seeders necessary to support bitrate r_i. Recall that after excluding the seeders themselves there are two sets of leechers that are going to get that bitrate. The first set contains the leechers who actually requested bitrate r_i and were not downgraded to a lower bitrate and these are represented by the term $n_{l_i} x_{ii}$. The second set consists of leechers that requested a higher bitrate and were downgraded to bitrate r_i and this set is represented by the term $\sum_{k=1}^{i-1} n_{l_k} x_{ki}$. Inequality 9 represents the proxy server capacity constraint. Note here that seeders are the only clients that receive data from the proxy and this is why leechers are not included in this condition. This optimization problem is guaranteed to have a solution only if the system can support the lowest bitrate r_R for all clients, which can be interpreted as the following condition

$$\frac{C_{proxy}}{r_R} \geq \frac{r_R - u_l}{u_s} \sum_{i=1}^{R} n_{l_i}$$

Solving this problem will result in values for x_{ij} and n_{s_i} for all i, j. Clearly, n_{s_i} will be the number of seeders that should receive video of bitrate r_i from the proxy. If $n_{s_i} = 0$ for any i it means that bitrate r_i will not be supported by the server. Moreover, $n_{s_i} = 0$ means either no clients requested bitrate r_i or some clients requested r_i but the server decided not to deliver it and downgraded these clients to lower bitrates due to overload and lack of server capacity. Alternatively, $n_{s_i} > 0$ does not necessarily mean that some clients requested bitrate r_i, it could mean that no clients requested rate r_i but the server chose to downgrade some of the clients who requested a higher bitrate to bitrate r_i. The values we get for x_{ij} can be used to randomly choose a fraction of leechers who requested bitrate r_i and deliver bitrate r_j to them.

System with churn. Assume that any arriving client will request a video stream of bitrate r_i with probability θ_i, and define $\lambda_i = \theta_i \lambda$ where λ is the general client arrival rate. We also assume that any client will stay in the system for a random period of time with

average $1/\mu$. Then the number of clients of bitrate r_i at any time in the system becomes a Poisson random variable with an average $\rho_i = \lambda_i/\mu$. In this case we observe that n_{l_i} in equations 7, 8 is a Poisson random variable with mean ρ_i. Using equation 3 we can solve the same optimzation problem after replacing inequality 8 with the following one

$$n_{s_i} u_s \geq (\phi_{1-\alpha}\sqrt{\hat{\rho}_i} + \hat{\rho}_i)(r_i - u_l)$$

where $\hat{\rho}_i = \rho_i x_{ii} + \sum_{k=1}^{i-1} \rho_k x_{ki}$

Solving such a nonlinear optimization problem can be complicated and since the number of seeders n_{s_i} we get from solving this problem is approximate we choose to use a linear approximation of the above inequality. We set $\sqrt{\hat{\rho}} = a + b\hat{\rho}$ and we use curve fitting tools to find values of constants a, b. Our simulation results show that this is a very good approximation and we do not lose the accuracy of our model. In this case the value of x_{ij} is considered as the probability that when a new client requests bitrate r_i it is granted bitrate r_j.

6.4.2 Constrained case

We know from equation 5 that the relation between the number of seeders and number of leechers for bitrate r_i in a constrained churnless system is $n_{s_i} u_s \geq n_{l_i}(r_i - \eta u_l)$. Hence, in order to find the optimal solution for the adaptation strategy for the constrained churnless system we can solve the optimization problem in equation 7 after replacing inequality 8 with the following one

$$n_{s_i} u_s \geq \left(n_{l_i} x_{ii} + \sum_{k=1}^{i-1} n_{l_k} x_{ki} - n_{s_i} \right)(r_i - \eta u_l) \tag{10}$$

Similar to what we did in the previous section, in order to find the optimal adaptive strategy for the constrained system with churn, we can solve the same optimization problem after replacing inequality 8 with the following one

$$n_{s_i} u_s \geq (\phi_{1-\alpha/2}\sqrt{\hat{\rho}_i} + \hat{\rho}_i)(r_i - \eta u_l - \epsilon)$$

where $\hat{\rho}$ is defined as in the previous section.

6.4.3 CDN adaptive live streaming

One of our aims in this chapter is to answer the following question "is a hybrid adaptive system better than a classic CDN adaptive system?", and if so, how much better will that be? By a classic CDN system, we mean the system where clients are served by the closest edge server of the CDN. Moreover, by a CDN adaptive system we mean that edge servers have multiple streams of different bitrates for the same video and clients can request different bitrates according to the quality of their Internet connection. Adaptive streaming here is defined in the same way as it was defined in section 6.4, where edge servers can decide to downgrade some of the clients to lower bitrates in case the server capacity is not sufficient to give each client its desired bitrate.

We will first develop an analysis for the system with a single bitrate by following similar steps to what we did in the hybrid system. Consider a CDN system with a fixed number of users n. Assume the CDN edge server has capacity C_e and provides a video stream of bitrate r. We can clearly see that the maximum bitrate that can be delivered by the server of all users is C_e/n. Now consider a system with churn, we follow the same assumptions of section 6.3 of Poisson arrivals of rate λ and general distribution of duration in the system with an average $1/\gamma$. Then, the number of clients in the system at any time $N(t)$ is a Poisson random variable with mean $\rho = \lambda/\gamma$. Now we can compute the probability that the system will support clients with bitrate r as follows

$$\begin{aligned} P(\text{support } r) &= P(r \leq \frac{C_e}{N}) = P(N \leq \frac{C_e}{r}) \\ &= P(\frac{N-\rho}{\sqrt{\rho}} \leq \frac{C_e/r-\rho}{\sqrt{\rho}}) = \Phi(\frac{C_e/r-\rho}{\sqrt{\rho}}) \end{aligned}$$

And then we can obtain the following condition on the edge server capacity similar to what we did before $C_e \geq r(\rho + \phi_{1-\alpha}\sqrt{\rho})$ which guarantees with confidence $(1-\alpha)\times 100\%$ that edge server capacity will be sufficient for providing bitrate r to arriving clients with rate ρ.

There are many performance metrics that could be used to compare the performance of CDN and hybrid adaptive streaming systems. For example, we could use the total number of clients that could be accommodated with a certain level of service, or we could use the quality of service received by clients in these two systems. We select client *dissatisfaction*

as our performance metric because we believe it is a reasonable quantified measure of the quality of the service received by clients. Similar to what we did in section 6.4.1, we model the adaptive streaming problem of a CDN system as a linear optimization problem with the objective of minimizing total client dissatisfaction.

Assume CDN edge servers have bitrates $r_1, \ldots, r_R$ where $r_1 > r_2 > \ldots > r_R$. Denote n_i as the number of clients that request bitrate r_i at the edge server. Define x_{ij} as the fraction of clients that request bitrate r_i but receive bitrate r_j. The CDN adaptive streaming problem can now be formulated as following

$$\min \sum_{i=1}^{R} \sum_{j=i}^{R} x_{ij} n_i (r_i - r_j) \tag{11}$$

subject to: $\sum_{j=i}^{R} x_{ij} = 1$, $0 \leq x_{ij} \leq 1$ for $i = 1, \ldots, R$

$$\sum_{i=1}^{R} r_i \left(n_i x_{ii} + \sum_{k=1}^{i-1} n_k x_{ki} \right) \leq C_e \tag{12}$$

Equation 11 represents our objective function of minimizing total client dissatisfaction. Edge server capacity constraint is represented by inequality 12. Recall that in a CDN system all clients receive data only from the edge server as compared to inequality 9 in the hybrid system where only seeders receive data from the proxy server. This condition means that total data rate delivered to all clients should not exceed the edge server link capacity. In order to understand inequality 12 remember that there are two sets of clients that receive bitrate r_i; the first one is represented by the term $n_i x_{ii}$ and these are the clients that requested bitrate r_i and were not downgraded to a lower bitrate. The second set is represented by the term $\sum_{k=1}^{i-1} n_k x_{ki}$ and these are the cliets that requested a bitrate higher than r_i but were downgraded by the server and eventually received bitrate r_i. It is important to mention that this optimization problem is guaranteed to have a solution only if the condition $C_e \geq r_R \sum_{i=1}^{R} n_i$ holds, which means that the CDN edge server can support the stream with the minimum bitrate r_R to all of its clients. Solving this problem, we can get the real positive values x_{ij} which can be used by the system to implement an admission control policy where the edge server should downgrade each client requesting bitrate r_i to a lower bitrate r_j with probabilty x_{ij}.

For the system with churn we can solve the same optimization problem after replacing inequality 12 with the inequality $\sum_{i=1}^{R} r_i(\hat{\rho}_i + \phi_{1-\alpha}\sqrt{\hat{\rho}_i}) \leq C_e$. We also use the linear approximation $\sqrt{\hat{\rho}} = a + b\hat{\rho}$ we used before.

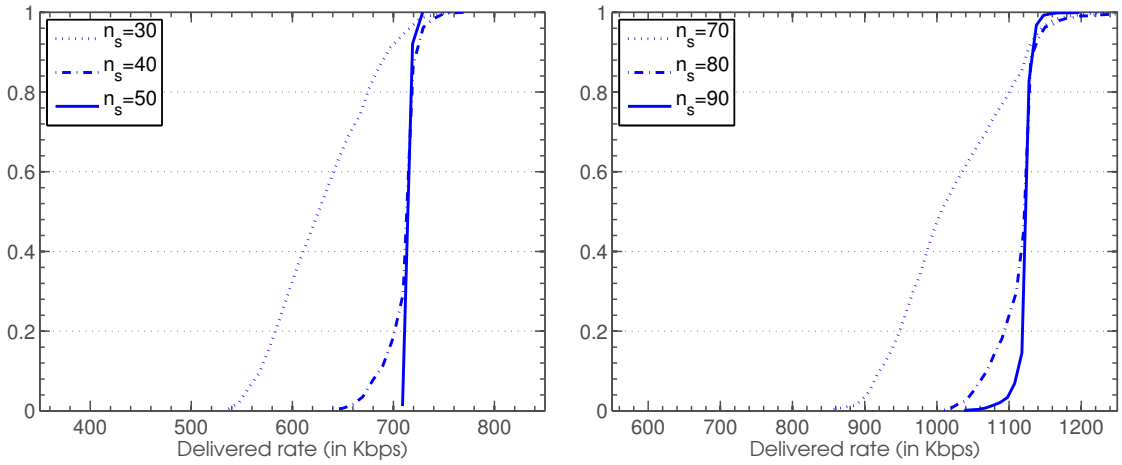
6.5 Analysis validation

6.5.1 Hybrid CDN/P2P streaming

In this section we validate our analytical results through simulation. We validate single bitrate analysis only because adaptive streaming results are based on the same analysis. We wrote a discrete event simulator for the system with a BitTorrent like client. We choose BitTorrent because it is one of most popular P2P clients and because we believe it is possible to integrate our system in real BitTorrent clients. We simulate the basic BitTorrent protocol and ignore some of the complicated details (such as tit-for-tat, peer choking, etc.).

In our simulator we assume that the proxy creates a torrent file for each video file (chunk). When a client connects to the proxy, if the proxy decides to treat the client as a seeder then the client downloads both the torrent and video files for each video chunk. Once a seeder has these two files for a chunk, it starts seeding the torrent file in the BitTorrent like client. On the other hand, if the proxy decides that the current seeders are enough, the requesting client is treated as a leecher and it downloads only torrent files for video chunks as soon as they are created. Once a leecher downloads a torrent file, it starts downloading the corresponding video file from the seeding peers. We assume that the BitTorrent tracker functionality is implemented by the proxy server.

Throughout the rest of this section we assume the following simulation setting. Video streams are split at the proxy server into chunks of ten seconds long. The upload rate of any client is selected randomly as either 350Kbps or 500Kbps. Client arrival is assumed to follow a Poisson process with arrival rates ranging from 100 to 400 clients/hour, and recall that this is a good approximation of client arrivals in real live streaming systems [111]. According to multiple studies of live streaming systems [93, 111], client viewing duration was found to follow a heavy-tailed distribution. We follow the model in [93] and assume

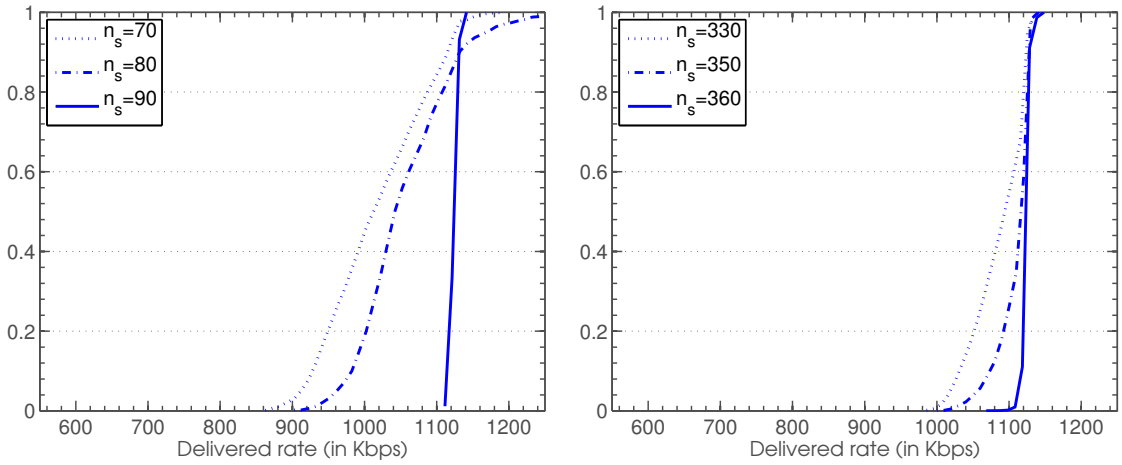


(a) $r = 700Kbps, \rho = 50$ (b) $r = 1100Kbps, \rho = 50$

Figure 46: CDF of average delivered rate for unconstrained system with churn

that client viewing duration is represented by a mixed-exponential distribution. The mixed-exponential probability density function (PDF) is $f(x) = \sum_{i=1}^{n} a_i \lambda_i e^{-\lambda_i x}$. This can easily be described as a set of n exponential distributions, with λ_i as the rate of exponential distribution i and a_i as the probability of selecting the i^{th} distribution. We also use the values of parameters a_i, λ_i that were obtained in [93]. For the constrained case we set $S_{in} = 20, Y_{in} = 10$.

We execute multiple simulation runs with multiple video bitrates ranging from $300Kbps$ to $2.4Mbps$. For each simulation run, we fix the video bitrate, number of seeders, and the number of leechers (churnless) or client arrival rate (churn). Each simulation run is worth ten hours or video streaming and at the end of each run we compute the average delivered bitrate for each leecher then we compute the cumulative distribution function (CDF) of the delivered bitrate for all leechers. In figure 46a we plot the CDF of delivered data rate using different number of seeders when the streamed video bitrate is 700Kbps and the arrival rate is 50 clients/hour. In figure 46b we repeat the same thing when the streamed bitrate is 1100Kbps. Similarly, in figures 47a and 47b we plot the CDF of delivered data rate when the streamed bitrate is 1100Kbps and client arrival rates are 50, 200 clients/hour respectively. In these plots, we can clearly see that solid lines represent the number of seeders that are sufficient to support respective bitrates to leechers, while other lines which represent lower



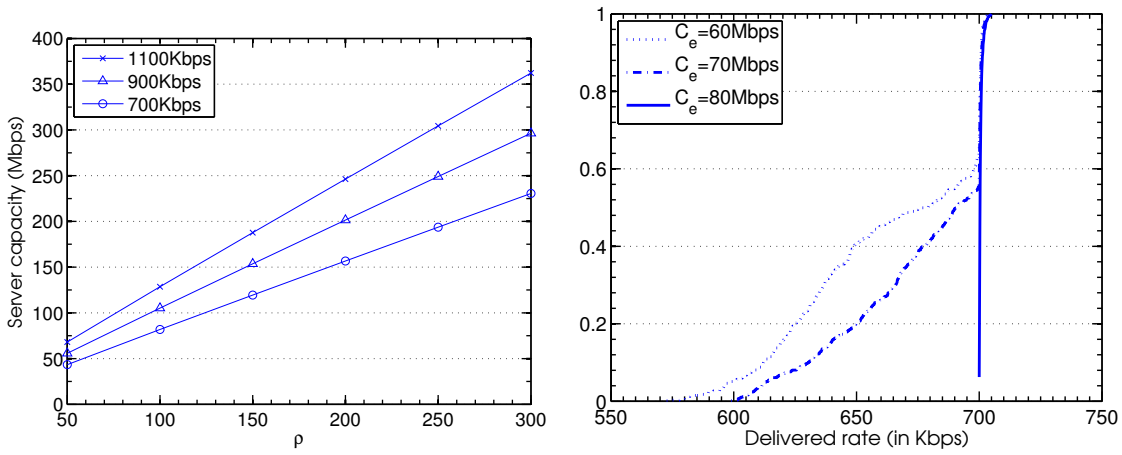
(a) $\rho = 50$ (b) $\rho = 200$

Figure 47: CDF of average delivered rate for constrained system with churn, $r = 1100Kbps$

number of seeders are not sufficient to support respective bitrates. This result matches the analysis we developed earlier in figure 45. For example, in figure 47a we can see that $n_s = 90$ is the minimum number of seeders that are sufficient to support the bitrate 1100Kbps, and this matches the lower bound of n_s we can get from figure 45b for $r = 1100$Kbps and $\rho = 50$.

6.5.2 CDN streaming

We wrote a client/server simulator to validate our analytical results of the single bitrate CDN-based system in section 6.4.3. Similar to the hybrid case, we use a Poisson process to simulate client arrivals and a mixed-exponential distribution to simulate video viewing duration. Our objective in this section is to validate the following condition we developed in section 6.4.3 for single rate systems, $C_e \geq r(\rho + \phi_{1-\alpha}\sqrt{\rho})$. We use this condition to plot the lower bound on server capacity required for supplying bitrate r for different bitrates in figure 48a. We execute multiple simulation runs with different video bitrates and different client behaviors. In figure 48b we plot the CDF of data rates delivered to clients when $\rho = 100$ and the server is streaming a bitrate of 700Kbps for different values of the server capacity, C_e. We can see that $C_e = 80$Mbps is the minimum server capacity sufficient for delivering a video bitrate of 700Kbps which matches the lower bound on C_e in figure 48a.



(a) Server capacity (C_e) vs ρ

(b) $\rho = 100, r = 700$Kbps

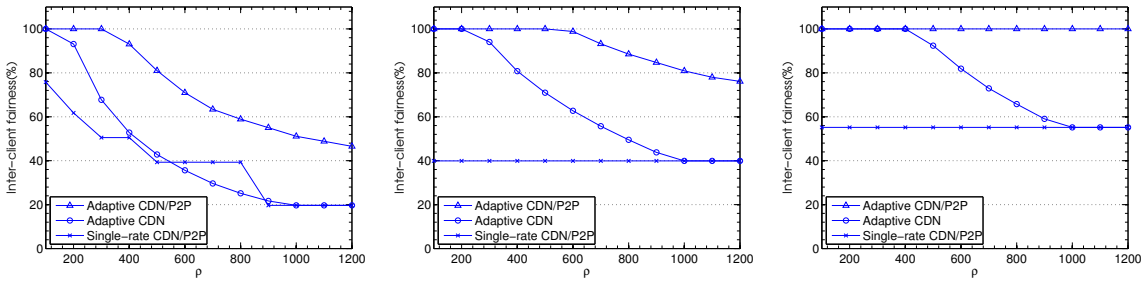
Figure 48: CDN system with churn

6.6 Illustrative case study

In the previous section we developed analysis for general CDN and hybrid live streaming systems. It should be emphasized that our results can be applied to a wide variety of system configurations and parameters. In this section we consider a case study of two systems with typical configurations and we use our analysis to evaluate the performance of adaptive live video streaming in these systems. Our goal in this case study is to measure the improvement in performance (if any) from using hybrid CDN/P2P systems.

We use two evaluation metrics for this purpose. The first one is the *inter-client fairness* described in section 6.4.1. Recall that *inter-client fairness* has a value of 100% when all clients receive the bitrates they originally requested and as clients start to get lower bitrates (due to adaptation) this value becomes lower. The second evaluation metric is *quantifying the savings in CDN server capacity* when we use the hybrid scheme as compared to the CDN scheme.

We assume a CDN system that offers a live video stream encoded in eight different qualities with the minimum bitrate as 350Kbps and the maximum bitrate as 2.4Mbps. We also consider three different profiles of streaming requests, namely *low*, *uniform*, and *high*. Each profile represents a different distribution of client requests over the different available bitrates. In the *low* profile, client requests are mainly focused on the four lowest bitrates



(a) High bitrate bias (b) Uniform distribution over bitrates (c) Low bitrate bias

Figure 49: Inter-client fairness for systems with churn

from 350Kbps to 1.1Mbps. Similarly, in the *high* profile, clients request streams of the four highest bitrates from 1.3Mbsp to 2.4Mbps. On the other hand, all bitrates have the same probability of being requested in the *uniform* profile. In the hybrid system, the average upload rate of a leecher, u_l, is assumed to be 300Kbps and the average upload rate of a seeder, u_s, is assumed to be 500Kbps.

We performed evaluation for systems with and without churn but we show results only to the more important case of systems with churn. We assume the CDN has a fixed server/proxy capacity of 500Mbps then we change client arrival rates and also change request distributions to be one of the profiles mentioned above.

In figure 49 we plot inter-client fairness against ρ, the average number of customers in the system, for both hybrid and CDN systems. In addition, we plot the same metric for a single-rate hybrid streaming system. Although in this case clients request different bitrates, we can apply single-rate hybrid streaming in the following manner: if the system is able to support the lowest bitrate that was requested by some clients then this bitrate is provided for all clients. Otherwise, the system will try all lower bitrates until it finds a bitrate that can be supported to all clients. Note that a client is always capable of receiving a bitrate lower than the bitrate it requested, but the reverse is not true.

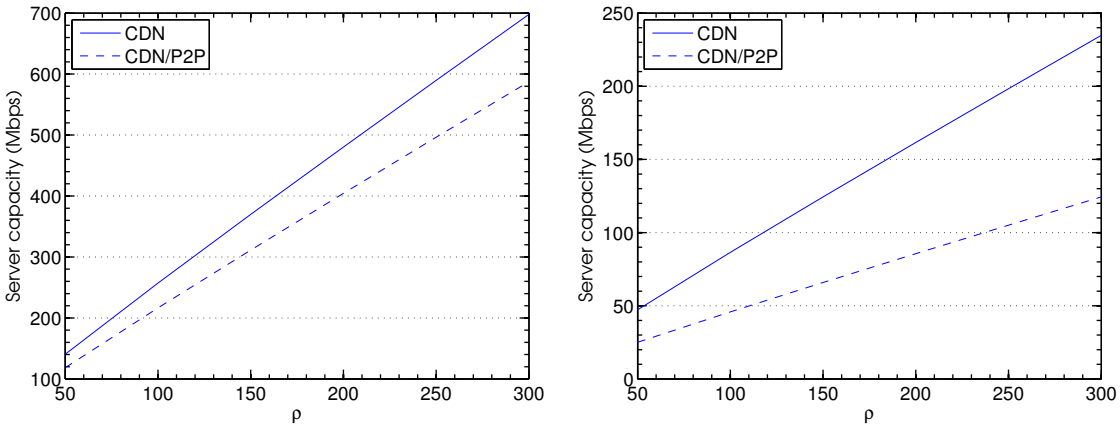
One expected observation for both hybrid adaptive and CDN systems and for all profiles is that inter-client fairness starts as 100% for lower number of customers in the system then it drops when there are more customers in the system. This is because the server(proxy) capacity is sufficient to satisfy client requests when the number of clients in the system

is low, but when this number grows high, the server becomes over-loaded and unable to satisfy all clients with the bitrates they asked for. At this point, the server(proxy) applies the adaptation strategy for the CDN(hybrid) system and decides to downgrade some clients to lower bitrates. The only exception to this observation is the *low* profile in figure 49c where inter-client fairness stays at 100% even when the number of clients in the system increases. This is because in the *low* profile case, the server(proxy) was able to give each client the bitrate it asked for without having to downgrade any client.

On the other hand, the single-rate hybrid system starts with inter-client fairness less than 100% for all profiles. This is because the best bitrate this system can provide is the lowest bitrate requested by some clients, for example in the *high* profile case, the best bitrate that can be provided for all clients is 1.3Mbps. Additionally, since the lowest bitrate (in this system), 350Kbps, is requested by some clients in the *low* and *uniform* profiles, the system has to provide that bitrate to all clients no matter how many clients are in the system, and this is why inter-client fairness remains constant in these two profiles. An important observation is that, as the number of customers in the system increases, the CDN adaptive system performance approaches the performance of the single-rate hybrid system, and in some cases (*high* profile), single-rate systems can even do better. From figure 49 we can see that hybrid systems can improve inter-client fairness from 20% to 40% over CDN systems depending on the distribution of the number of requests to different bitrates and the average number of customers in the system.

In order to measure savings in CDN server capacity from using the hybrid system compared to CDN systems we ask the following question; if we were to achieve inter-client fairness of 100%, how much server capacity do we need for both the hybrid and CDN cases? To do that, we fix all system parameters including client arrival rate and request distribution profile then we compute the server(proxy) capacity that will satisfy all client requests with their desired bitrates for both the CDN and the hybrid cases. Using the analysis developed in sections 6.3.4, 6.4.3 we can calculate savings in CDN server capacity as

$$C_e - C_{proxy} = \sum_{i=1}^{R} (\phi_{1-\alpha}\sqrt{\rho_i} + \rho_i)(\eta u_l + \epsilon)$$



(a) High bitrate bias

(b) Low bitrate bias

Figure 50: Required server capacity for CDN/P2P and CDN systems with churn

It is interesting to see that capacity saving from using hybrid systems is directly proportional to both the average number of clients in the system ρ_i and the average client upload rate u_l. This means that as the number of clients in the system increases, hybrid systems become more effective which can be observed from figures 49, 50.

In figure 50 we plot the computed server(proxy) capacity against the average number of customers in the system. We can see that hybrid systems can save about 21%, 32%, and 100% of CDN server capacity in the high, uniform, and low profiles respectively. The reason capacity savings is less in the *high* profile than other profiles is that peer contribution in hybrid systems become more limited when clients request higher bitrates. This is because we assume clients have asymmetric upload and download rates which means client upload rate is much less than client download rate. For example, a standard cable connection in the United States has a download rate of 3Mbps while the upload rate is usually in the range 380/760Kbps. On the other hand, hybrid systems can be much more effective in other countries where users have more symmetric download/upload rates (e.g. China).

6.7 Summary

In this chapter, we analyze adaptive streaming in a hybrid CDN/P2P live streaming system. Our analysis is driven by the need to develop solutions to two important design questions in hybrid systems. The first question is how to find a way to switch the operation of the

system between the CDN and P2P modes. The second question is how to find the best bitrate adaptation strategy. This strategy basically specifies how bitrates are assigned to different clients when the CDN server can not satisfy each client with the bitrate it requested due to capacity constraints. We believe that these two design decisions mostly control the effectiveness of any hybrid adaptive streaming system.

We develop a stochastic fluid model for a hybrid streaming system with a single video bitrate. We obtain theoretical results that help the CDN server decide when to switch from the CDN to P2P mode. After that, we extend that model to the multiple bitrate case and we develop a linear optimization formulation to get the best bitrate adaptation strategy. Using simulations, we validate our analysis. We use our analysis to evaluate a case study of typical CDN and hybrid systems. Results show that hybrid systems can improve average user satisfaction about 20% to 40% as compared to CDN systems depending on the distribution of client requests to different bitrates. We also find that hybrid systems can achieve significant savings in CDN server capacities as compared to CDN systems; these savings could be from 21% to 100% again depending on the distribution of client requests to different bitrates.

CHAPTER VII

SUMMARY OF CONTRIBUTIONS AND FUTURE WORK

Video streaming is dominating traffic on the Internet and is expected to grow even more in the future as users keep shifting from regular broadcast TV to online streaming services. HTTP adaptive streaming is being adopted by most popular commercial video streaming services. Being able to provide a high quality streaming service for millions of users with heterogeneous streaming devices poses a lot of challenges. We investigate several of these problems and develop a set of tools and techniques to help solve them. In summary, the contributions of this thesis are:

1. We provide a better understanding of video player behavior for mobile platforms in three commercial streaming services (Netflix, YouTube, and Hulu). We identify several patterns that video players use to download video from the server; these patterns affect the player's ability to exploit the available bandwidth. In addition, we discover varying amounts of "redundant" traffic in the presence of bandwidth adaptation across the services, which negatively impacts network resources. We also find these design choices lead to unfairness in bandwidth consumption on shared networks across different platforms. In particular, we find the Android Netflix player is able to take a larger fraction of shared bandwidth when competing with the iOS implementation.

2. We extend existing video QoE metrics to adapt to different screen sizes and resolutions. We then define max-min QoE fairness for a set of adaptive video flows flowing through a network. We develop an algorithm to compute the set of bitrates that should be received by each client to achieve QoE fairness. Furthermore, we design and implement a system on home routers to apply QoE fairness in a home network. Our results show that this system can significantly improve QoE fairness for a set of competing video flows compared to a traditional home router.

3. We show that HTTP adaptive video streaming can be harmful to other applications sharing the same residential network. Our results show that even a single video stream can cause up to one second of queuing delay and it even gets worse when the home link is congested. In addition, we introduce SABRE, a client based technique that can be implemented in the video player to mitigate this problem. We implemented SABRE in the VLC DASH player. Using testbed experiments, we show that SABRE manages to significantly reduce queuing delays while not affecting the user viewing experience. Our results show that SABRE can coexist with traditional streaming players without having any performance penalties.

4. We develop a stochastic fluid model to describe a hybrid CDN/P2P live streaming system. We use the model to compute when the system should switch between the CDN and P2P modes. We then develop a linear optimization formulation for computing the best bitrate assignment for a set of clients when the system is overloaded and can not grant every client its desired bitrate. Using simulations, we validate our analysis. We use our analysis to evaluate a case study of typical CDN and hybrid systems. Results show that hybrid systems can improve average user satisfaction about 20% to 40% as compared to CDN systems depending on the distribution of client requests to different bitrates. We also find that hybrid systems can achieve significant savings in CDN server capacities as compared to CDN systems; these savings could be from 21% to 100% again depending on the distribution of client requests to different bitrates.

7.1 future work

The work done in this thesis can be extended in several directions. We describe some of the potential future work below.

- **Mitigating traffic redundancy of mobile video players.** As explained earlier in chapter 3, mobile video players can download upto $15 - 20\%$ of redundant traffic when switching between bitrates. This redundancy is mainly due to downloading previously downloaded segments and replacing them with higher bitrates to improve

user streaming experience. One way of mitigating this problem is by implementing a video proxy in the ISP. The video proxy is a transparent proxy that receives video requests from clients and forwards them to the video server. The proxy keeps a copy form all previous downloaded video segments within a short time window (few video segments). When a client requests a segment that was downloaded before, the proxy does not forward the request, instead it replies to the client with the stored video segment.

- **Large scale study of traffic redundancy in adaptive video players.** We mentioned in chapter 3 that the high traffic redundancy we observe from video players is mainly because of the design of our controlled experiments. In order to measure redundancy in real large scale networks, streaming traces from large networks (e.g. campus network or ISP) need to be collected and studied to observe the behavior of adaptive video players in these traces.

- **QoE fairness algorithm.** In chapter 4 we developed a centralized algorithm for computing QoE max-min fair bitrates. If any of the inputs of the algorithm change, it has to recompute everything form the beginning. A very useful extension is to have an incremental algorithm that can compute fair bitrates efficiently as clients join and leave the system. Moreover, developing a distributed algorithm for computing QoE fair bitrates without the intervention of the network could be very challenging. Implementing such an algorithm, however, can be facilitated using WebRTC. Using WebRTC, clients can communicate with each other, exchange manifest files and available bitrates if needed.

- **Understanding the multiple control loops affecting the operation of adaptive video players.** As discussed in chapter 4, the operation of adaptive video players can be affected by three control loops: 1) TCP behavior, 2) client bandwidth estimation and bitrate adaptation, and 3) client adaptation to bandwidth allocation enforced by the network. Understanding the different time-scales of the operation

of these loops and the interaction between them is very crucial for controlling the operation of adaptive video players.

- **Adaptive hybrid CDN/P2P systems for stored video.** As mentioned before, hybrid systems can be used to reduce the load on the peering links between ISPs and video streaming providers. However, adaptive streaming for stored video using hybrid systems can be very challenging. First, clients will have to store some video segments on their hard drives so they can be uploaded to other clients later. The ISP will also have to provide a service to allow clients to learn about video segments that can be downloaded from other peers (similar to the Tracker server in BitTorrent). The system still poses the following challenges:

  - Every client has to agree to donate part of his hard drive to store video segments. Since the effectiveness of P2P delivery depends significantly on the number of participating peers, proper incentive mechanisms need to be developed to convince clients to participate. One possibility is to give monthly discounts to clients willing to participate in the P2P system.
  - Given the limited allocated space on the hard drive of every client, the system needs to define smart caching algorithms to decide which video segments to cache. The caching algorithm needs to answer the following questions: 1) which movies should be cached? 2) which segments of these movies should be cached? 3) for each selected segment, which bitrate(s) should be cached?

- **Understanding the economics between ISPs and video providers.** We mentioned earlier the recent disputes between ISPs and video providers about the amount of traffic flowing in the peering links between them. Due to the increasing amounts of traffic flowing from video providers to ISP networks, there have been cases where ISPs ask video providers to pay more money for that traffic. From one side, video providers have the incentive to pay ISPs so that end-users can receive good video quality, otherwise, end-users may unsubscribe from the video service. From the other

side, however, ISPs do not want to lose potential clients who may switch to different ISPs because they are having bad video streaming quality. Understanding this perspective in the relationship between ISPs and video providers is important and can lead us to predict how this relation can develop in the future.

REFERENCES

[1] "Big Buck Bunny." http://www.bigbuckbunny.org.

[2] "Bittorrent." http://www.bittorrent.com.

[3] "DASHJS." http://dashif.org/reference/players/javascript/index.html.

[4] "EMule." http://www.emule.com.

[5] "Host Extensions for IP Multicasting." http://tools.ietf.org/html/rfc1112.

[6] "Http live streaming." https://developer.apple.com/streaming.

[7] "Inauguration day, by the numbers." http://news.cnet.com/8301-13577_3-10145923-36.html.

[8] "Iperf." http://iperf.sourceforge.net.

[9] "Libpcap." http://www.tcpdump.org.

[10] "MPEG-DASH." http://dashif.org/mpeg-dash.

[11] "OpenWrt." http://openwrt.org.

[12] "Pixel density." http://en.wikipedia.org/wiki/Pixel_density.

[13] "Reliable Multicast Research Group." http://irtf.org/concluded/rmrg.

[14] "RTP: A Transport Protocol for Real-Time Applications." http://tools.ietf.org/html/rfc3550.

[15] "TCP Congestion Control." http://tools.ietf.org/html/rfc5681.

[16] "VLC player." http://www.videolan.org/vlc/index.html.

[17] "WebRTC." http://www.webrtc.org.

[18] "Wireshark." http://www.wireshark.org.

[19] "YouTube Live." http://www.youtube.com/live.

[20] "Smooth Streaming Transport Protocol." http://go.microsoft.com/?linkid=9682896, 2009.

[21] "Cisco visual networking index: Forecast and methodology, 2012–2017," 2013.

[22] "Global internet phenomena report." http://bit.ly/1nCPIia, 2013.

[23] A. Banerjea, D. Ferrari, B. Mah, M. Moran, D. Verma, and H. Zhang, "The Tenet Real-time protocol Suite: Design, Implemetation, and Experience," *IEEE/ACM Transactions on Networking*, vol. 4, pp. 1–10, Feb. 1996.

[24] A. Begen, T. Akgul, and M. Baugher, "Watching Video over the Web, Par t 1: Streaming Protocols," *IEEE Internet Computing*, vol. 15, no. 2, pp. 54–63, 2011.

[25] A. Zambelli, "IIS smooth streaming technical overview." `http://download.microsoft.com/download/4/2/4/4247C3AA-7105-4764-A8F9-321CB6C765EB/IIS_Smooth_Streaming_Technical_Overview.pdf`, 2009.

[26] Adhikari, V., Jain, S., Chen, Y., and Zhang, Z.-L., "Vivisecting youtube: An active measurement study," in *INFOCOM*, 2012.

[27] Adhikari, V. K., Guo, Y., Hao, F., Hilt, V., and Zhang, Z.-L., "A tale of three cdns: An active measurement study of hulu and its cdns," in *INFOCOM Workshops*, pp. 7–12, 2012.

[28] Adhikari, V. K., Guo, Y., Hao, F., Varvello, M., Hilt, V., Steiner, M., and Zhang, Z.-L., "Unreeling netflix: Understanding and improving multi-cdn movie delivery.," in *INFOCOM*, 2012.

[29] Adhikari, V. K., Jain, S., and Zhang, Z.-L., "Youtube traffic dynamics and its interplay with a tier-1 isp: An isp perspective," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, 2010.

[30] Adobe, "HTTP Dynamic Streaming on the Adobe Flash Platform." `http://www.adobe.com/products/httpdynamicstreaming/pdfs/httpdynamicstreaming_wp_ue.pdf`, 2010.

[31] Akamai, "http://www.akamai.com/dl/featured-sheets/akamai-media-streaming.pdf."

[32] Akhshabi, S., Anantakrishnan, L., Dovrolis, C., and Begen, A., "What happens when http adaptive streaming players compete for bandwidth?," NOSSDAV '12, pp. 89–94, ACM, 2012.

[33] Akhshabi, S., Anantakrishnan, L., Dovrolis, C., and Begen, A. C., "Server-based traffic shaping for stabilizing oscillating adaptive streaming players," in *NOSSDAV*, pp. 19–24, 2013.

[34] Akhshabi, S., Begen, A. C., and Dovrolis, C., "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over http," in *MMSys '11*, 2011.

[35] Appenzeller, G., Keslassy, I., and McKeown, N., "Sizing router buffers," in *SIGCOMM '04*, (New York, NY, USA), pp. 281–292, ACM, 2004.

[36] Bertsekas, D. P. and Gallager, R. G., *Data Networks*. Prentice Hall, 1992.

[37] Bonfiglio, D., Mellia, M., Meo, M., and Rossi, D., "Detailed analysis of skype traffic," *IEEE Transactions on Multimedia*, vol. 11, pp. 117–127, Jan. 2009.

[38] Cao, Z. and Zegura, E., "Utility max-min: an application-oriented bandwidth allocation scheme," in *INFOCOM '99*, vol. 2, pp. 793–801 vol.2, Mar 1999.

[39] Cha, M., Kwak, H., Rodriguez, P., Ahn, Y.-Y., and Moon, S., "I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, 2007.

[40] Chen, J., Mahindra, R., Khojastepour, M. A., Rangarajan, S., and Chiang, M., "A scheduling framework for adaptive video delivery over cellular networks," MobiCom '13, pp. 389–400, 2013.

[41] Chiu, M. and Jain, R., "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer N etworks," *Computer Networks and ISDN Systems*, 1989.

[42] Cisco, "Cisco visual networking index: Forecast and methodology, 2009–2014," 2010.

[43] Cranley, N., Perry, P., and Murphy, L., "User Perception of Adapting Video Quality," *International Journal of Human-Computer Studies*, vol. 64, Aug. 2006.

[44] D. Tran, K. Hua, and T. Do, "ZIGZAG: An efficient peer-to-peer scheme for media streaming," in *Proceedings of IEEE INFOCOM '03*, 2003.

[45] D. Wu, Y. T. Hou, and Y.-Q. Zhang, "Transporting real-time video over the Internet: Challenges and approaches," in *Proceedings of IEEE*, 2000.

[46] Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B., "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, pp. 50–58, 2002.

[47] E. Akyol, A. Tekalp, and M. Civanlar, "A Flexible Multiple Description Coding Framework for Adaptive Peer-to-Peer Video Streaming," *IEEE Journal of Selected Topics in Signal Processing*, vol. 1, pp. 231–245, Aug. 2005.

[48] Eskicioglu, A. and Fisher, P., "Image quality measures and their performance," *IEEE Transactions on Communications*, vol. 43, pp. 2959–2965, Dec 1995.

[49] Finamore, A., Mellia, M., Munafò, M. M., Torres, R., and Rao, S. G., "Youtube everywhere: Impact of device and infrastructure synergies on user experience," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pp. 345–360, ACM, 2011.

[50] Floyd, S., Handley, M., Padhye, J., and Widmer, J., "Equation-based congestion control for unicast applications," in *Proceedings of the conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '00, (New York, NY, USA), pp. 43–56, ACM, 2000.

[51] Floyd, S. and Jacobson, V., "Random early detection gateways for congestion avoidance," *IEEE/ACM Transactions on Networking*, vol. 1, pp. 397–413, Aug. 1993.

[52] G. Thompson and Y. Chen, "IPTV: Reinventing Television in the Internet Age," *IEEE Internet Computing*, vol. 13, pp. 11–14, May 2009.

[53] G. Tian and Y. Liu, "Towards Agile and Smooth Video Adaption in Dynamic HTTP Streaming," in *Proceedings of the ACM CoNEXT*, 2012.

[54] Gettys, J. and Nichols, K., "Bufferbloat: dark buffers in the internet," *Commun. ACM*, vol. 55, pp. 57–65, Jan. 2012.

[55] Ghobadi, M., Cheng, Y., Jain, A., and Mathis, M., "Trickle: Rate Limiting YouTube Video Streaming," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, 2012.

[56] Gill, P., Arlitt, M., Li, Z., and Mahanti, A., "Youtube Traffic Characterization: A View from the Edge," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, 2007.

[57] H. Yin, X. Liu, T. Zhan, V. Sekar, F. Qiu, C. Lin, H. Zhang, and B. Li, "Design and Deployment of a Hybrid CDN-P2P System for Live Video Streaming: Experience with LiveSky," in *ACM International Conference on Multimedia*, 2009.

[58] Heia, X., Liang, C., Liang, J., Liu, Y., and Ross, K., "Insights into pplive: A measurement study of a largescale p2p iptv system," in *IPTV Workshop, International World Wide Web Conference*, 2006.

[59] Huang, C., Li, J., , and Ross, K., "Can internet video-on-demand be profitable?," in *Sigcomm*, 2007.

[60] Huang, T.-Y., Handigol, N., Heller, B., McKeown, N., and Johari, R., "Confused, timid, and unstable: picking a video streaming rate is hard," IMC '12, 2012.

[61] Huang, T.-Y., Johari, R., and McKeown, N., "Downton abbey without the hiccups: Buffer-based rate adaptation for http video streaming," in *Proceedings of the 2013 ACM SIGCOMM Workshop on Future Human-centric Multimedia Networking*, FhMN '13, pp. 9–14, 2013.

[62] Hyunwoo Nam, Bong Ho Kim, D. C. and Schulzrinne, H. G., "Mobile video is inefficient: A traffic analysis," tech. rep., Columbia University, 2013.

[63] J. Bolot and T. Turletti, "A rate control mechanism for packet video in the Internet," in *Proceedings of IEEE INFOCOM '94*, 1994.

[64] J. Bolot, T. Turletti, and I. Wakeman, "Scalable feedback control for multicast video distribution in the Internet," in *Proceedings of ACM SIGCOMM '94*, 1994.

[65] J. Jannotti, K. Gifford, L. Johnson, F. Kaashoek, and J. OToole, "Overcast: reliable multicasting with an overlay network," in *Proceedings of OSDI*, 2000.

[66] J. Jiang, V. Sekar, and H. Zhang, "Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE," in *Proceedings of the ACM CoNEXT*, 2012.

[67] J. Padhye, J. Kurose, D. T. and Koodli, R., "A Model Based TCP-Friendly Rate Control Protocol," in *Proceedings of NOSSDAV '99*, 1999.

[68] J. Venkataraman and P. Francis, "Multi-tree unstructured peer-to-peer multicast," in *Proceedings of the 5th International workshop on peer-to-peer systems*, 2006.

[69] Jain, R., Chiu, M., and Hawe, W., "A Quantitative Measure of Fairness and Discrimination for Resource Allocation in Shared Computer Systems," *ACM Transactions on Computer Systems*, 1984.

[70] Jiang, T., Ammar, M., and Zegura, E., "Inter-receiver fairness: A novel performance measure for multicast abr sessions," in *SIGMETRICS*, 1998.

[71] Kim, T. and Ammar, M., "Optimal quality adaptation for mpeg-4 fine-grained scalable video," in *Infocom*, 2003.

[72] Kleinrock, L., *Queueing Systems, Vol. 1:Theory.* John Wiley ans Sons, 1975.

[73] Krishnamurthy, B., Wills, C., and Zhang, Y., "On the use and performance of content distribution networks," in *ACM SIGCOMM INTERNET MEASUREMENT WORKSHOP*, 2001.

[74] Kumar, R., Liu, Y., and Ross, K., "Stochastic fluid theory for p2p streaming systems," in *Infocom*, 2007.

[75] L. Wu, R. Sharma, and B. Smith, "Thin Stream: An Architecture for multicasting layered video," in *Proceedings of ACM NOSSDAV '97*, 1997.

[76] Lederer, S., Müller, C., and Timmerer, C., "Dynamic adaptive streaming over http dataset," in *Proceedings of the 3rd Multimedia Systems Conference*, MMSys '12, (New York, NY, USA), pp. 89–94, ACM, 2012.

[77] Liu, C., Shi, L., and Liu, B., "Utility-based bandwidth allocation for triple-play services," in *Fourth European Conference on Universal Multiservice Networks, 2007. ECUMN '07*, pp. 327–336, Feb 2007.

[78] Liu, S., Zhang-Shen, R., Jiang, W., Rexford, J., and Chiang, M., "Performance bounds for peer-assisted live streaming," in *SIGMETRICS*, 2008.

[79] Liu, Y., Li, F., Guo, L., Shen, B., Chen, and Songqing, "A comparative study of android and ios for accessing internet streaming services," in *PAM*, vol. 7799, pp. 104–114, 2013.

[80] Liu, Y., Li, F., Guo, L., Shen, B., and Chen, S., "A comparative study of android and ios for accessing internet streaming services," in *Proceedings of the 14th International Conference on Passive and Active Measurement*, PAM'13, pp. 104–114, 2013.

[81] M. Castro, P. Druschel, M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh, "SplitStream: high-bandwidth multicast in cooperative environments," in *Proceedings of SOSP '03*, 2003.

[82] M. Zhang, L. Zhao, J. Tang, and S. Yang, "A peer-to-peer network for streaming multimedia through the Internet," in *Proceedings of ACM Multimedia*, 2005.

[83] Malik, O., "Verizon: That peering flap (about Netflix) is Cogent fault." `http://bit.ly/1g8bkQI`.

[84] Mansy, A., Ammar, M., Chandrashekar, J., and Sheth, A., "Characterizing client behavior of commercial mobile video streaming services," in *MoVid'14*, 2014.

[85] Masnick, M., "France Telecom Accused Of Holding YouTube Videos Hostage Unless It Gets More Money." `http://bit.ly/MYl2g3`, 2013.

[86] Müller, C. and Timmerer, C., "A test-bed for the dynamic adaptive streaming over http featuring session mobility," in *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, (New York, NY, USA), pp. 271–276, ACM, 2011.

[87] N. Magharei, and R. Rejaie, "Prime: peer-to-peer receiver driven mesh-based streaming," in *Proceedings of IEEE INFOCOM '07*, 2007.

[88] Nelakuditi, S., Harinath, R., Kusmierek, E., and Zhang, Z.-L., "Providing smoother quality layered video stream," in *NOSSDAV*, 2000.

[89] Nguyen, A. T., Li, B., and Eliassen, F., "Chameleon: Adaptive peer-to-peer streaming with network coding," in *Infocom*, 2010.

[90] Parker, J. A., Kenyon, R. V., and Troxel, D., "Comparison of interpolating methods for image resampling," *IEEE Transactions on Medical Imaging*, vol. 2, pp. 31–39, March 1983.

[91] Prasad, R. S., Dovrolis, C., and Thottan, M., "Router buffer sizing for tcp traffic and the role of the output/input capacity ratio," *IEEE/ACM Transactions on Networking*, vol. 17, pp. 1645–1658, Oct. 2009.

[92] Qiu, D. and Srikant, R., "Modeling and performance analysis of bittorrent-like peer-to-peer networks," in *SIGCOMM*, 2004.

[93] Qiu, T., Ge, Z., Lee, S., Wang, J., Xu, J., and Zhao, Q., "Modeling user activities in a large iptv system," in *Internet Measurement Conference (IMC)*, 2009.

[94] R. Bettati, D. Ferrari, A. Gupta, W. Heffner, W. Howe, M. Moran, Q. Nguyen, and R. Yavatkar, "Connection Establishment for Multi-Party Real-time Communication," in *Proceedings of ACM NOSSDAV '95*, 1995.

[95] R. Pantos, "HTTP Live Streaming." IETF Internet draft, November 2010.

[96] R. Rejaie, M. H. and Estrin, D., "An End-to-end Rate-based Congestion Control Mechanism for Realtime Streams in the Internet," in *Proceedings of INFOCOMM '99*, 1999.

[97] Ranjan, P., Abed, E., and La, R., "Nonlinear instabilities in tcp-red," *IEEE/ACM Transactions on Networking*, vol. 12, pp. 1079–1092, Dec. 2004.

[98] Rao, A., Legout, A., Lim, Y.-s., Towsley, D., Barakat, C., and Dabbous, W., "Network characteristics of video streaming traffic," CoNEXT '11, ACM, 2011.

[99] Rejaie, R. and Ortega, A., "Pals: Peer-to-peer adaptive layered streaming," in *NOSSDAV*, 2003.

[100] Rubenstein, D., Kurose, J., and Towsley, D., "The impact of multicast layering on network fairness," SIGCOMM '99, pp. 27–38, 1999.

[101] S. Akhshabi, A. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP," in *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, pp. 157–168, 2011.

[102] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. Begen, "What Happens When HTTP Adaptive Streaming Players Compete for Bandwidth?," NOSSDAV '12, pp. 89–94, ACM, 2012.

[103] S. Cheung, M. Ammar, and X. Li, "On the use of Destination Set Grouping to improve fairness in multicast video distribution," in *Proceedings of IEEE INFOCOM '96*, 1996.

[104] S. McCanne, V. Jacobson, and M. Vetterli, "Receiver-driven layered multicast," in *Proceedings of ACM SIGCOMM '96*, 1996.

[105] Sarkar, S. and Sivarajan, K., "Fairness in cellular mobile networks," *IEEE Transactions on Information Theory*, vol. 48, pp. 2418–2426, Aug 2002.

[106] Sarkar, S. and Tassiulas, L., "Fair allocation of discrete bandwidth layers in multicast networks," tech. rep., University of Maryland, 1999.

[107] Sarkar, S. and Tassiulas, L., "Fair allocation of discrete bandwidth layers in multicast networks," in *INFOCOM*, vol. 3, pp. 1491–1500 vol.3, Mar 2000.

[108] Sarkar, S. and Tassiulas, L., "Fair allocation of utilities in multirate multicast networks: a framework for unifying diverse fairness objectives," *IEEE Transactions on Automatic Control*, vol. 47, pp. 931–944, Jun 2002.

[109] Shenker, S., "Fundamental design issues for the future internet," *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 7, pp. 1176–1188, 1995.

[110] Sisalem, D. and Schulzrinne, H., "The Loss-Delay Based Adjustment Algorithm: A TCP-Friendly Adaption Scheme," in *Proceedings of NOSSDAV '98*, 1998.

[111] Sripanidkulchai, K., Maggs, B., and Zhang, H., "An analysis of live streaming workloads on the internet," in *Internet Measurement Conference (IMC)*, 2004.

[112] Stockhammer, T., "Dynamic adaptive streaming over http –: standards and design principles," in *Proceedings of the second annual ACM conference on Multimedia systems*, MMSys '11, (New York, NY, USA), pp. 133–144, ACM, 2011.

[113] Su, A.-J., Choffnes, D., Kuzmanovic, A., and Bustamante, F., "Drafting behind akamai," in *SIGCOMM*, 2006.

[114] T. Kim and M. Ammar, "A Comparison of Heterogeneous Video Multicast Schemes: Layered Encoding or Stream Replication?," *IEEE Transactions on Multimedia*, vol. 7, pp. 1123–1130, Dec. 2005.

[115] Tewari, S. and Kleinrock, L., "Analytical model for bittorrent-based live video streaming," in *IEEE Consumer Communications and Networking Conference*, 2007.

[116] Torres, R., Finamore, A., Kim, J. R., Mellia, M., Munafo, M. M., and Rao, S., "Dissecting video server selection strategies in the youtube cdn," in *Proceedings of ICDCS '11*, 2011.

[117] V. Padmanabhan, J. Wang and P. Chou, "Resilient peer-to-peer streaming," in *Proceedings of ICNP '03*, 2003.

[118] Wang, Z., Bovik, A., Sheikh, H., and Simoncelli, E., "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, pp. 600–612, April 2004.

[119] X. Li and M. Ammar, "Bandwidth control for replicated-stream multicast video distribution," in *Proceedings of HPDC '96*, 1996.

[120] X. Li, S. Paul, and M. Ammar, "Layered Video Multicast with Retransmission (LVMR): Evaluation of hierarchical rate control," in *Proceedings of IEEE INFOCOM '98*, 1998.

[121] X. Li, S. Paul, P. Pancha, and M. Ammar, "Layered Video Multicast with Retransmission (LVMR): Evaluation of error recovery," in *Proceedings of NOSSDAV '97*, 1997.

[122] X. Zhang, J. Liu, B. Li, and T. Yum, "CoolStreaming/DONet: a data-driven overlay network for peer-to-peer live media streaming," in *Proceedings of INFOCOM '05*, 2005.

[123] Xu, D., Kulkarni, S., Rosenberg, C., and Chai, H.-K., "Analysis of a cdn–p2p hybrid architecture for cost-effective streaming media distribution," *Multimedia Systems*, vol. 11, pp. 383–399, 2006.

[124] Xue Li, Mostafa Ammar, and Sanjoy Paul, "Video Multicast over the Internet," *IEEE Network Magazine*, vol. 13, pp. 46–60, Mar. 1999.

[125] Y. Chu, S. Rao, and H. Zhang, "A case for end system multicast," in *Proceedings of SIGMETRICS*, 2000.

[126] Y. Guo, K. Suh, J. Kurose, and D. Towsley, "P2cast: peer-to-peer patching schemes for VoD service," in *Proceedings of WWW '03*, 2003.

[127] Y. Liu, Y. Guo, and C. Liang, "A survey on peer-to-peer video streaming systems," *Peer-to-Peer Networking and Applications*, vol. 1, pp. 18–28, Jan. 2008.

[128] Zink, M., Suh, K., Gu, Y., and Kurose, J., "Characteristics of YouTube Network Traffic at a Campus Network - Measurements, Models, and Implications," *Computer Networks*, vol. 53, pp. 501–514, Mar. 2009.