2016

Parallel HTTP for Video Streaming in Wireless Networks

Ansari, Mohsen

Ansari, M. (2016). Parallel HTTP for Video Streaming in Wireless Networks (Unpublished master's thesis). University of Calgary, Calgary, AB. doi:10.11575/PRISM/27254
<http://hdl.handle.net/11023/3507>

Downloaded from PRISM Repository, University of Calgary

UNIVERSITY OF CALGARY

Parallel HTTP for Video Streaming in Wireless Networks

by

Mohsen Ansari

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

December, 2016

© Mohsen Ansari 2016

Abstract

To stream video using HTTP, a client device sequentially requests and receives chunks of the video file from the server over a TCP connection. It is well-known that TCP performs poorly in networks with high latency and packet loss such as wireless networks. On mobile devices, in particular, using a single TCP connection for video streaming is not efficient, and thus, the user may not receive the highest video quality possible. In this thesis, we design and analyze a system called *ParS* that uses parallel TCP connections to stream video on mobile devices. Our system uses parallel connections to fetch each chunk of the video file using HTTP range requests. We present measurement results to characterize the performance of ParS under various network conditions in terms of latency, loss rate and bandwidth. Given the limited communication and computational resources of mobile devices, we then focus on determining the minimum number of TCP connections required to achieve high utilization of the wireless bandwidth. We develop a simple model and study its accuracy using ns-3 simulations which confirm the utility of our model for estimating the minimum number of TCP connections required to fully utilize the available bandwidth.

Table of Contents

Abstract	ii
Table of Contents	iii
List of Tables	v
List of Figures	vi
1 Introduction	1
1.1 Motivation	1
1.2 Objectives	2
1.2.1 Recognizing the issues in streaming	3
1.2.2 Proposing our solution	5
1.3 Contributions	6
1.4 Organization	8
2 Background and Related Work	10
2.1 HTTP Protocol	10
2.2 Streaming over HTTP	12
2.2.1 History of Streaming	12
2.2.2 Streaming protocols and systems	14
2.3 Dynamic Adaptive Streaming over HTTP (DASH)	17
2.3.1 Overview	17
2.3.2 Media Presentation	18
2.3.3 DASH Streaming Method	19
2.3.4 rate adaptation	19
2.3.5 DASH vs. Apple HLS vs. MS Smooth Streaming	22
2.4 TCP and HTTP Streaming	23
2.5 Why parallel TCP improves the throughput	24
2.6 Parallel TCP for File Transfer	25
2.7 Parallel TCP for Video Streaming	26
2.7.1 Quality of Experience Factors of HTTP Video Streaming	29
3 Methodology	31
3.1 Overview	31
3.2 Model	31
3.2.1 TCP Modeling	31
3.2.2 Number of TCP Connections	32
3.2.3 Small Transmit Queue	33
3.2.4 Large Transmit Queue	36
3.3 Parallel HTTP Streaming (ParS)	38
3.3.1 Overview	38
3.3.2 System Description	38
3.3.3 System Implementation	39
3.3.4 Simulation Implementation	43
4 Results	48
4.1 Implementation results	48
4.1.1 Testbed description	48

4.1.2	Measurements - weak signal	49
4.1.3	Measurements - strong signal	62
4.1.4	Campus wifi	70
4.2	Simulation results	72
4.2.1	Testbed description	72
4.3	Small Transmit Queue	72
4.3.1	Fluid Approximation	73
4.3.2	Packet Approximation	73
4.4	Large Transmit Queue	75
4.5	Transmit Queue Occupancy	76
5	Conclusion	79
5.1	Thesis summary	79
5.2	Future work	81
A	Appendix	83
A.1	ParS.java	83
A.2	simulation.cc	91
	Bibliography	98

List of Tables

2.1	Comparison of the features among streaming service. Microsoft Smooth Streaming, Apple HTTP LiveStreaming, and DASH	22
4.1	Numbers and the bitrates they represent in bitrate frequency figures for the video dataset [57].	49
4.2	Number of TCP flows from the model.	75
4.3	Average queue occupancy from simulations.	78

List of Figures and Illustrations

1.1	Throughput for single TCP connection. One connection is not able to fully utilize the link and increasing RTT and packet loss decreases the throughput.	3
1.2	Parallel TCP connections with loss probability of 0.1% and RTT of 50 ms. While the aggregate throughput is essentially flat for 3 or more connections, the per-connection throughput decreases	4
1.3	Timeline order of chunks in a video playback. Chunk # 1 is currently playing by the user. The rest need to be downloaded from the server.	6
2.1	Overview of DASH between an HTTP server and a client. All the communications are via HTTP. The client gets the Media Presentation Description file first. Then, requests the segments with a bitrate that is chosen by HTTP Streaming Control module. Then, HTTP Access Client downloads and sends the segments to Media Player.	17
2.2	Communications between a streaming server and a client. The playout begins after client downloads the first segment. The client changes the bitrate of the next segment.	20
2.3	Using parallel connections to download video segments. The method that is used in [10]	26
2.4	Using multiple http streaming queues to download video chunks from server. Used in [7]. Packet number 4 didn't successfully transmitted to the client. Therefore, it is added to two separate queues to increase its chance of delivery.	28
3.1	Network topology used in simulations. Internet link imitates the link between the access router and the server over the Internet. The access link imitates the link between the access router (<i>e.g.</i> , home WiFi router) and the end device.	33
3.2	Traditional Parallel Streaming (TPS) approach for using parallel connections in video streaming. each connection gets a separate chunk.	39
3.3	Using parallel connections to download the same video chunk (ParS). All connections download the same chunk but different parts.	40
3.4	Methods and their responsibilities in ParS.java implementation.	41
3.5	HTTP-GET sample request and response using byte-range feature.	42
3.6	Main function's flowchart	47
4.1	WiFi test bed for weak signal test cases. The server and the client are each located in a different room on 2 nd floor. The router is located on the main floor to have the maximum distance from the client and the server.	50
4.2	Bitrates selected for ParS compared to TPS from 2 parallel connections to 10, in the weak WiFi signal environment. The aggregate frequency of 5 runs are demonstrated. The x-axis numbers represent the bitrates shown in Table 4.1.	56
4.3	Average bitrate for ParS compared to TPS. This is the weighted average of selected bitrates and their frequency from 1 connection to 10 parallel connections. The numbers are derived from the same experiments that are conducted for Figure 4.2.	59

4.4	Time-to-start playback for ParS compared to TPS in weak WiFi signal. The average of 5 runs are represented in the figure. The x-axis shows the number of parallel connections.	60
4.5	WiFi testbed for strong signal test cases	61
4.6	bitrates selected for ParS compared to TPS from 2 parallel connections to 10, in strong WiFi signal environment. The aggregate frequency of 5 runs are demonstrated. The x-axis numbers represent the bitrates shown in Table 4.1.	67
4.7	Average bitrate for ParS compared to TPS with strong WiFi signal. The weighted average of selected bitrates and their frequency from using 1 connection to 10 parallel connections. The numbers are derived from the same experiments that are conducted for Figure 4.6.	69
4.8	Time-to-start playback for ParS compared to TPS in strong WiFi signal. The average of 5 runs are represented in the figure. The x-axis shows the number of parallel connections.	70
4.9	Time-to-start playback for ParS compared to TPS in university campus WiFi. The client is connected to the campus network via WiFi and the server is connected to campus network with Ethernet cable. The average of 5 runs are represented in the figure. The x-axis shows the number of parallel connections.	71
4.10	Small transmit queue: Comparison of the model and simulation when the packet loss probability due to transmission errors is set to zero. The model provides an upperbound, which becomes more accurate as the queue size increases.	73
4.11	Small transmit queue: Trace of TCP window size when queue is one packet.	74
4.12	Small transmit queue: Aggregate throughput increases as the gap between the start time of connections increases.	75
4.13	Large transmit queue: Aggregate throughput under different loss probabilities.	76
4.14	Large transmit queue: When packet loss is 0.02, the link utilization is over 93%. Beyond this, the link utilization drops.	77
4.15	Transmit queue occupancy with different number of flows. The size of the queue is set to 100 packets.	77

Chapter 1

Introduction

1.1 Motivation

Video streaming over HTTP has become extremely popular and is adopted by major online streaming services such as YouTube and Netflix. Internet video streaming now takes the majority of worldwide Internet traffic and its share will continue to grow. It is estimated that, globally, Internet video traffic will be 80 percent of all consumer Internet traffic in 2019, up from 64 percent in 2014 [1].

There are several HTTP-based video streaming services implemented by different organizations. Adobe's HTTP Dynamic Streaming [2], Apple's HTTP Live Streaming [3] and Microsoft's Smooth Streaming [4] are a few examples of such services. Meanwhile, Dynamic Adaptive Streaming over HTTP (DASH) is being developed as an international standard to unify HTTP-based video streaming over the Internet [5].

The DASH standard is composed of two main parts. One part defines the Media Presentation Description (MPD) that is used by the client to discover the URLs for accessing the video content. The other part defines the format of the video content. In a DASH system, the video streaming server sends video content to clients via the HTTP protocol. Before transmission, a video is encoded into different bit rates on the server. The encoded video file at every bit rate is fragmented into small *chunks*, each chunk contains only several seconds (*e.g.*, 10 seconds) of the video. A client sends HTTP requests to the server to download video chunks sequentially. At the time of streaming, a client can dynamically change the target video chunk's bit rate based on its available resources such as available bandwidth and remaining battery [6].

HTTP video streaming systems, including DASH, rely on TCP which has poor performance in networks with high latency and packet loss. In such networks, TCP and consequently the video

streaming application is unable to fully utilize the available bandwidth leading to a lower quality of experience for users [7]. In Fig. 1.1, we have plotted the measured throughput of a single TCP connection¹ over a 10 Mbps link. We increase round-trip-time (RTT) under different packet loss rates and measure the throughput. When RTT and packet loss are set to 50 milliseconds and 0.1%, respectively, TCP can utilize nearly 86% of the available bandwidth. By increasing RTT and packet loss, the throughput drops drastically to a point where, when RTT is set to 150 milliseconds and packet loss is 1%, TCP can only utilize 13% of the bandwidth.

Wireless networks often suffer from high latency and packet loss [8]. Moreover, wireless bandwidth is generally more limited compared to wired bandwidth. This means that HTTP-based video streaming over mobile devices is negatively affected by TCP's inability to fully utilize the wireless bandwidth, which could result in lower video quality and buffering delays during video playback. To improve TCP throughput, the use of parallel connections has been considered. For example, GridFTP [9] is an extension of traditional FTP in which multiple TCP connections are used to transfer data resulting in significant improvements over single connection FTP. This concept has been also applied to video streaming. For instance, Parallel TCP connections have been used in [10] and [7] to improve the quality of video by using multiple connections to download multiple video chunks from the server simultaneously.

1.2 Objectives

With the above points in mind, this work examines the TCP protocol and its performance in order to address the challenges in video streaming on HTTP which relies on TCP for its data transfer. It also reviews the existing streaming applications and analyzes them, then proposes a new method. With the expansion of videos streaming services, and the appearance of different methods of video streaming, the importance of this topic and its issues can not be ignored. As mentioned in this work, we review the different methods of video streaming and point out the common weakness of

¹We use the terms “flow” and “connection” interchangeably.

Figure 1.1: Throughput for single TCP connection. One connection is not able to fully utilize the link and increasing RTT and packet loss decreases the throughput.

all of them (*i.e.*, their inability of utilizing the available bandwidth). There are a number of works in this area to address the issue of under-utilization of bandwidth in video streaming. Most of them try to enhance the throughput of the data transfer connection but don't consider which part of the video is most important to download. In a video playback, assuming that chunk number n is currently playing, it is clear that the next chunk *i.e.*, chunk number $n + 1$, has the top priority to be downloaded from the streaming server. The reason is that, if the next chunk doesn't get downloaded from the server on time, a pause or a *stall* happens. Stall is one of the important factors in quality of experience that can have a negative effect on user's opinion. Continuing occurrences of it, results in frustration of the user and negative effect on quality of experience. Hence, video streaming applications try their best to avoid any stalls happening during a video playback.

1.2.1 Recognizing the issues in streaming

We discussed earlier that the issue of existing video streaming applications is their inability of utilizing the available bandwidth due to their reliance on single HTTP/TCP connection which

Figure 1.2: Parallel TCP connections with loss probability of 0.1% and RTT of 50 ms. While the aggregate throughput is essentially flat for 3 or more connections, the per-connection throughput decreases

doesn't perform well in high latency and packet loss networks. Figure 1.1 depicts that issue clearly. In the figure, when the loss rate is low (0.1%), most of the available bandwidth can be utilized. Therefore, we can see more than 80% of the 10Mbps bandwidth is utilized when RTT is 50ms. But as we increase the loss rate, it can be seen that the utilization drops to 40% and lower. This illustrates that a single TCP connection does perform well in good network conditions but when the loss rate and latency increase, TCP's performance decreases as well.

Most streaming applications such as Apple's Live Streaming, Microsoft's Smooth Streaming, and Dynamic Adaptive Streaming on HTTP (DASH) rely on a single TCP connection. To address and solve this issue, several works have been done. Some of them focused on modifying the rate adaptation algorithm (*i.e.*, the algorithm that chooses the bitrate of the next chunk of a video), in order to avoid stalls in a video playback [11], [12]. Some works proposed UDP-based protocols claiming it doesn't have the under-utilization problem of TCP [13]. And some others proposed the use of multiple simultaneous TCP connections to increase the throughput and utilization [10], [14].

The idea of using multiple TCP connections is better applicable to current network and Internet infrastructure compared to its alternatives because of the following reasons:

- It is easy to implement and doesn't need third party libraries, APIs, run-time environments, etc.
- It does not have the firewall blocking problem like many UDP-based protocols have.
- It does not require complex calculations and heuristic algorithms which may cause processing overhead on the server or the client.

1.2.2 Proposing our solution

So far, we briefly demonstrated the weakness of TCP and HTTP video streaming applications and also mentioned the streaming applications and methods. These topics will be discussed more in depth in Sections 2.4 and 2.2.1 respectively. We also mentioned that among the proposed ideas, using parallel TCP connections is more applicable than others. There are several works already done in this area. We will cover these works in Chapter 2. To the best of our knowledge, the existing works in this area that use parallel TCP in video streaming have a common problem. To address that, consider Figure 1.2. In this figure, we illustrate the aggregate throughput when using parallel connections alongside with the average throughput of each connection. The point of this figure is to show that when using parallel connections to download video chunks, the total throughput may increase but the throughput gained by each individual connection decreases. Now consider Figure 1.3, in this case, if we use parallel connections to download the video chunks 2, 3, 4, and 5, we are dedicating connections to the chunks that have less priority (*e.g.*, chunks 3, 4, and 5). This approach decreases the throughput for downloading the chunks with higher priority (*e.g.*, chunk # 2). This method, not only doesn't improve the quality of experience for the user, but also increases the chance of stalls and lower bit rate of chunks to be downloaded. Therefore, the use of connections to download the video chunk with the highest priority is critical for having a good quality of experience. In Chapter 3 we expand this idea and explain the details of it.

Figure 1.3: Timeline order of chunks in a video playback. Chunk # 1 is currently playing by the user. The rest need to be downloaded from the server.

1.3 Contributions

In this work, we employ parallel TCP connections to improve video streaming quality on mobile devices. However, different from the existing works (as in [10] and [7]), rather than using multiple connections to fetch different video chunks, we use multiple connections to fetch the *same chunk*. We note that when multiple connections are used, the available bandwidth between the client and server is divided among the connections. This means that, while the aggregate throughput increases, the throughput of each individual connection actually *decreases* as depicted in Fig. 1.2. For instance, when there are 5 parallel connections the throughput of each connection on average is less than 2 Mbps.

In wired networks, where the bandwidth between a client and a server is plenty, even the reduced per-connection bandwidth is large enough to sustain video streaming at a good quality. Thus, it makes sense to use multiple connections to download different video chunks. In wireless networks however, the bandwidth is limited. Thus, when using multiple connections, the share of each individual connection may be so low that it cannot sustain video playback at a reasonable quality, resulting in playback stalls. This problem occurs whenever a chunk is needed *as soon as possible* to avoid a playback stall. For example, consider having 3 parallel connections. Then, it takes *3 times longer* to download the first chunk and start the video when using parallel connections to download different chunks compared to downloading the same chunk. In wireless networks, when the available bandwidth fluctuates significantly over time, this situation (*i.e.*, needing chunks

quickly to avoid stalls) would be even more likely to happen compared to wired networks.

The work presented in this thesis has two parts. In the first part, we focus on using multiple connections for video streaming as described above. We design and evaluate *Parallel Streaming* (ParS) that employs parallel HTTP for video streaming in low bandwidth and high loss networks. In the second part, we turn our attention to determining the relationship between the aggregate throughput and the number of parallel TCP connections. A critical problem in all systems that use parallel TCP connections, is to decide how many connections are needed to achieve the best video playback quality. We show that as the number of connections increases, the aggregate throughput also increases up to a certain point. Beyond this point, increasing the number of connections actually results in lower throughput due to increased competition among the connections and increased processing and computation overhead. To this end, we develop a simple model to estimate the number of connections required to achieve high utilization of the available bandwidth in different scenarios.

Our contributions in this work can be summarized as follows:

- We develop a protocol to use parallel TCP connections to download each chunk of a video file from the server using HTTP range requests.
- We develop a prototype system based on our protocol called Parallel Streaming (ParS) and deploy it on a testbed. We then conduct measurement experiments to analyze the impact of RTT, packet loss, bandwidth and chunk size on the throughput performance of our protocol.
- We develop a simple model to determine the number of parallel connections required to fully utilize a network link.
- We measure and compare the Quality of Experience factors of our method (ParS) with Traditional Parallel Streaming (TPS) in different WiFi network environments. TPS is explained in 3.3.

- We conduct ns-3 simulations to study the accuracy and utility of our model that is described in 3.2.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 reviews the background information and related works in the area of video streaming and HTTP streaming. HTTP protocol's history is mentioned and the changes among the 3 versions of HTTP(*i.e.*, HTTP/0.9, HTTP/1.0, and HTTP/1.1) is reviewed. Also, the relation between HTTP and its underlying protocol (TCP) is explained. Then, on streaming over HTTP, the history of streaming is mentioned. The emersion of first live streaming event and then the appearance HTTP-based adaptive streaming is reviewed. Further on streaming protocols, the streaming applications and protocols such as Microsoft Media Server [15], Apple HTTP Live Streaming [3], Adobe HTTP Dynamic Streaming [2], Real Time Streaming Protocol [16], Real-time Transport Protocol [17], and Real Time Messaging Protocol [18] is reviewed. Afterwards, the Dynamic Adaptive Streaming over HTTP (DASH) is demonstrated and the details of it such as media presentation, communication between a server and a client is discussed. Furthermore, the weakness and problems of TCP and TCP congestion control is illustrated and the most common methods to overcome these weaknesses examined. Finally, the applications of parallel TCP in video streaming is studied. Chapter 3 presents our model for determining how many parallel connections are required to fully utilize a network link. First, it studies the models for calculating the total TCP throughput and it clarifies how they are different from our work. Then, the model is presented for small transmit queue 3.2.3 and large transmit queue 3.2.4 cases. Also, it explains our approach for parallel HTTP streaming (ParS) 3.3. In the explanation of ParS, the system is described. Then, the implementation of it is reviewed. After that, the simulation which is implemented to evaluate the model is reviewed. Chapter 4 is dedicated to presenting our approach and includes the measurement results obtained from experiments on a test bed. It describes the testbeds that are used for our experiments. The experiments are done in three different environ-

ments: weak WiFi signal, strong WiFi signal, and university campus WiFi. In each of the testbeds, situation of the client and the server is mentioned and then, the quality of experience is depicted and examined. A summary of our ns-3 simulation results is also presented in Section 4.2. That section, describes the test bed and the parameters set for the simulation. Then, the corresponding results for small and large transmit queues are demonstrated and discussed. Finally, Section 5 concludes our work and Appendix A includes the source codes for ParS implementation and model simulations.

Chapter 2

Background and Related Work

In this chapter, we review some of the background information for video streaming over HTTP, explain the challenges in this area, and review the standards in video streaming and proposed solutions to the issues mentioned in Chapter 1. In Section 2.1, we explain the HTTP protocol, its use, and its features. Section 2.2, focuses on how video streaming works over HTTP and describes existing protocols and systems in this area. Section 2.3, describes Dynamic Adaptive Streaming over HTTP (DASH) system, discusses its standard, and compares it with its alternatives. At the end of this chapter, we will review the limitations and problems of using TCP protocol and present the solutions to address the issues of video streaming over single HTTP/TCP connection.

2.1 HTTP Protocol

In this section, we will review the history of HTTP protocol, its versions and differences among them and then we study the behavior and methodology of it, explaining the communication steps between a user sending requests and a server receiving the requests and sending back responses.

The Hypertext Transfer Protocol (HTTP) is a protocol in the application layer that is used for distributed and collaborative information systems. It is the primary protocol known for communication among World Wide Web users. HTTP has been in use to transfer multimedia (i.e., text, hypertext, images, audio, and video). The first version of HTTP(HTTP/0.9), was a simple protocol used for transferring raw data across the Internet [19]. It only supported GET request types and the response was a message in Hypertext Mark-up Language (HTML). HTTP/1.0 was officially introduced and recognized in 1996. This version supported multiple request types such as POST, HEAD, PUT and DELETE. It also supported multiple response headers depending on the condition of a server, its availability, and type of the requested file. In the introduced response headers,

content type of the response message can be specified, so that the receiving end (e.g., web browser) can open the response file with proper extension/plugin [20]. For example, a browser with the ability to play video files on its own can recognize the response message and start the video playback if the content-type in the response header, is specified as a video file. If a client wants to download a video file that is located at *http://www.example.com/someVideoDirectory/video.mp4*, here is how HTTP communication will be between the server and the client:

1. The client makes a TCP connection to the server which is *www.example.com* on port 80 (default port number for HTTP).
2. The client sends an HTTP request message to the server using the TCP connection.
The message will look like this:

GET /someVideoDirectory/video.mp4 HTTP/1.0

Host: *www.example.com*
3. The server receives the request and if the file exist,s retrieves the file (/someVideDirectory/video.mp4). Then, the server encapsulates the file in an HTTP response and sends the response to the user via the established TCP connection.
4. The server tells its TCP layer to close the TCP connection. The TCP connection however, doesn't get terminated until after the client receives the response message.
5. The client receives the message and TCP connection terminates. The received message is an HTML file therefore, the client extracts the video file from the response message.

HTTP/1.1 made further improvements to previous versions and added more response header fields to enhance the communication between a requesting user and a responding server. One of the features that was added in this version of HTTP was Range field in the request message's header. The Range field enables the user to request a specific part of a file rather than requesting the whole

file [19]. This particular feature is important to us because we make use of it for requesting a portion of a desired file by sending multiple parallel requests to a server. We will describe this method in details in Chapter 3.

2.2 Streaming over HTTP

In the following sections, we will review a brief history of streaming over the Internet and over HTTP. We'll also examine the growth of streaming from the first streaming event until it became a standard, which many pioneering streaming services make use of. Then, we will introduce the streaming solutions, systems and protocols. Moreover, we compare the existing streaming solutions; Following this, we will study the DASH standard.

2.2.1 History of Streaming

Streaming has been in use for over two decades now. The first known live streaming event goes back to September 5, 1995, when ESPN SportsZone streamed a live radio broadcast of a baseball game between the Seattle Mariners and the New York Yankees to thousands of its subscribers worldwide. A startup company named Progressive Networks developed the technology. A few years later, the company changed its name to RealNetworks. At this time, the future of media streaming over the Internet appeared to be exciting. However, it had to overcome many pragmatic problems such as how to successfully stream watchable video over 56kbps dial-up modems. The startup company, RealNetworks found itself in a legal conflict with Microsoft over the new technology and at the end; Microsoft emerged from this as the winner (thanks to its Microsoft Media technologies). In the early 2000s, Macromedia eroded Microsoft's market share by its Flash Player, and was later acquired by Adobe Systems [21].

By the mid-2000s, the majority of Internet traffic was HTTP-based. Content delivery networks (CDNs) were increasingly being used to deliver popular content to a large number of audiences. However, media streaming was struggling to keep up with the demand [21]. Then, in 2007, a

company named Move Networks introduced a technology that changed the industry: HTTP-based adaptive streaming. Move Networks used the HTTP protocol to deliver media in small file chunks while the player application monitored the download rate and requested chunks with varying quality (bitrate) in response to changing network conditions. The new technology allowed streaming media to be distributed with far less buffering and connectivity issues for customers because instead of making the user wait for the next video segment to download, it was adjusting the bitrate of the video based on measured parameters of the network (e.g., download speed). Other HTTP-based streaming solutions soon followed: Microsoft launched Microsoft Smooth Streaming [4] in 2008; the same year Netflix developed its own technology in this area. In 2009, Apple made HTTP Live Streaming (HLS) [3], which was designed for delivery to iOS devices, and Adobe joined in 2010 with HTTP Dynamic Streaming (HDS) [2]. HTTP-based adaptive streaming quickly became used for major sporting events (Vancouver and London Olympics, Wimbledon, Roland Garros, Felix Baumgartner’s jump, etc) and premium video streaming on-demand services (Netflix, LoveFilm, Amazon Instant Video, etc.) [21].

In 2009, a partnership project among seven telecommunications standard development organizations known as 3GPP (The 3rd Generation Partnership Project) [22], began to establish a standard for adaptive streaming. More than 50 companies were involved (Microsoft, Netflix, and Apple, included) and the effort was coordinated with other industry organizations such as 3GPP, DECE, OIPF and W3C. By April 2012, the standard was published as “Dynamic Adaptive Streaming over HTTP”—colloquially known as MPEG-DASH [5]. When streaming with DASH, the server stores multiple copies of video chunks with different bitrates and stores the information about those files in an XML format file. The user first requests the XML and then starts to download the first chunk. After that, the streaming begins and the control of the quality of video playback is by user. The user can request high or low resolution video chunks based on the measured network parameters. Changing the quality of the video playback dynamically, is called rate adaptation. We will describe DASH in details further in this chapter.

2.2.2 Streaming protocols and systems

In the previous sections, we mentioned streaming protocols such as Microsoft Smooth Streaming, HLS and HDS. In this section, we will examine such protocols besides earlier versions of streaming protocols such as, Microsoft Media Server (MMS) [15], Real Time Streaming Protocol (RTSP) [16], Real-time Transport Protocol (RTP) [17] and Real Time Messaging Protocol (RTMP) [18].

Microsoft Media Server [15]

The MMS Protocol is designed for transferring real-time multimedia (audio and video). The goal of MMS protocol is to help ease the cases in which multimedia is being transferred and rendered simultaneously, for example, a video displayed and audio played. In this scenario, the system that initiates the connection is referred to as the client, and the system that responds to the connection is referred to as the server. The data flows from the server towards the client. There are two types of information transfer between the server and the client: request and control messages and multimedia data. The client can send request messages to the server via a TCP connection, requesting the server to perform control actions such as starting and stopping the flow of multimedia. The multimedia data can be transferred over the same TCP connection or as a separate flow of UDP packets. If UDP is used for transferring the multimedia data, the client can send an MMS protocol message to the server requesting it to resend the UDP packet. This is useful if the client does not receive a UDP packet because of unreliable UDP. In newer versions of MMS, the client can send messages to the server, requesting to change the stream that is being transmitted. For example, the client may request a lower bit-rate stream of the same video that was being transmitted by the server [15].

Apple HTTP Live Streaming [3]

Apple originally developed HTTP Live Streaming to allow content providers to send live or prerecorded audio and video to Apple's iPhone and/or iPod Touch using an ordinary Web server. They

later included the same playback functionality in their QuickTime desktop media player. The general view of how HLS works is that, a multimedia presentation is specified by a Universal Resource Identifier (URI) to a play-list file, which is an ordered list of media URIs and informational tags. The URIs and their associated tags specify a series of media segments. To play the stream, the client first obtains the play-list file and then obtains and plays each media segment in the play-list. It reloads the play-list file to discover additional segments [23]. Conceptually, HTTP Live Streaming consists of three parts: the server component, the distribution component, and the client software. The server component is responsible for taking input streams of media and encoding them digitally, encapsulating them in a format suitable for delivery, and preparing the encapsulated media for distribution. The distribution component consists of standard web servers. They are responsible for accepting client requests and delivering prepared media and associated resources to the client. The client software is responsible for determining the appropriate media to request, downloading those resources, and then reassembling them so that the media can be presented to the user in a continuous stream [24].

Adobe HTTP Dynamic Streaming [2]

HDS is Adobe's method for adaptive bitrate streaming for Flash Video. This method of streaming enables on-demand and live adaptive bitrate video delivery of MP4 media over regular HTTP connections. Adaptive bitrate works by detecting a client's bandwidth capacity and adjusts the quality of the video stream between multiple bitrates and/or resolutions in real-time. Before the development of HDS, Adobe only offered progressive downloading over HTTP connections. So users could play a progressive download file, such as a YouTube clip, and could see the video downloading so they would know how far they can skip ahead. With dynamic streaming, users will be able to jump ahead to any point and start watching almost immediately.

Real Time Streaming Protocol

The Real Time Streaming Protocol (RTSP) is an application layer protocol designed to give users the control over the transfer of streaming data with real-time properties. It provides a framework

that enables media servers to control the streaming of real-time data (e.g., audio and video) whether the data source is stored (*e.g.*, movie, animation) or it is live feed (*e.g.*, sport event, news). RTSP is intended to manage multiple data transfer sessions which means the user of this protocol has the option to choose the under-lying transfer protocol such as UDP, multicast UDP, and TCP. The main features of RTSP are: Retrieving media from server, Inviting a media server to a virtual conference (a media server can be invited to an existing conference either to record the media or to play back a media), Adding media to an existing presentation (useful for live presentations because the server can tell its users about additional media becoming available) [16].

Real-time Transport Protocol

RTP is an end-to-end protocol for transmitting real-time data (e.g., audio and video). It can work on both multicast and unicast networks. RTP does not guarantee quality for service of streaming however, data transmission in RTP is enhanced by a control protocol (RTCP) which enables large-scale monitoring of the data delivery. RTP and RTCP are designed in such a way that are independent of the underlying layers (i.e., network and transport layers). The responsibilities of RTP and RTCP are explained as following: RTP is responsible for transferring the data that has real-time properties. RTCP monitors the transferring data to measure the quality of service and conveys information about the ongoing sessions [17].

Real Time Messaging Protocol

Before the development of HDS, Adobe was using the Real-Time Messaging Protocol (RTMP) for transmission of audio, video, and data between Adobe Flash Platform technologies, including Adobe Flash Player and Adobe AIR. RTMP is now available as an open specification to create products that enable delivery of video, audio, and data in the open AMF, SWF, FLV, and F4V formats compatible with Adobe Flash Player [18].

Figure 2.1: Overview of DASH between an HTTP server and a client. All the communications are via HTTP. The client gets the Media Presentation Description file first. Then, requests the segments with a bitrate that is chosen by HTTP Streaming Control module. Then, HTTP Access Client downloads and sends the segments to Media Player.

2.3 Dynamic Adaptive Streaming over HTTP (DASH)

In this section we examine the DASH standard which was made by 3GPP as a form of standardized streaming solution in addition to the already existing RTP/RTSP-based streaming solutions and the HTTP-based progressive download solutions.

2.3.1 Overview

The main structure of DASH streaming between an HTTP server and a user is shown in Figure 2.1.

The DASH standard provides the specification of the following components:

- a definition of a Media Presentation, where Media Presentation is defined as a structured collection of data that is provided for the DASH client through Media Presen-

tation Description file,

- a definition of the formats of a segment where, a segment is defined as an internal data unit of a media that can be uniquely addressed by an HTTP-URL
- a definition of the delivery protocol used for transmission of segments, namely HTTP/1.1,
- an informative description of how a DASH client can establish a streaming service for its user(s).

Figure 2.2 illustrates the overall scenario and communications between a streaming server and its client. The streaming begins with the client receiving the Media Presentation Description (MPD) file. This file contains the URLs to segments of the video with different bitrates. Then, the client requests the desired segment with a bitrate specified by its rate adaptation policy. The rate adaptation policy and the algorithm for selecting the next segment to download are flexible and depend on the implementation of DASH on the client side.

2.3.2 Media Presentation

A Media Presentation is a structured collection of encoded data of some media content (e.g., a movie or a program). The data is accessible to the DASH Client to provide a streaming service to the user. As shown in Figure 2.1:

- A Media Presentation is a sequence of one or more consecutive periods (e.g., Intro, movie, commercial, rest of the movie)
- Each period contains one or more representation from the same media content (e.g., different bitrates of a video)
- Each representation contains one or more segments

- Segments contain the media and metadata to decode and present the included media content.

A Media Presentation is described in a Media Presentation Description (MPD), and MPDs may be updated during the lifetime of a Media Presentation. In particular, the MPD describes accessible Segments and their timing. The MPD is a well-formatted XML document and the 3GPP Adaptive HTTP Streaming specification defines an XML schema to define MPDs [25]. Please refer to MPEG-DASH [5] description document for in detail description of DASH specifications.

2.3.3 DASH Streaming Method

DASH specification provides a universal and flexible solution for Adaptive HTTP Streaming. The solution is based on existing technologies. The technologies such as codecs, encapsulation formats, multimedia protection and transfer protocols. DASH focuses on the specifications of the interface between a standard HTTP server and a HTTP streaming client [25]. To summarize, we can say DASH specifies syntax and semantics of the MPD file, format of segments, and the delivery protocol for segments. Figure 2.2 demonstrates the communication messages between a DASH streaming server and a client. Right after the MPD is delivered to the client, client sends a request to get the first segment of the video. Then, playback of the video begins when the first segment is received completely. The client sends requests in order to get the next segments for the video playback while it can decide whether to change the bitrate of the segments or keep the same bitrate. In this case, client sends a request for 192 kbps bitrate for segment number one, then sends a request for 384 kbps bitrate for segment number two. The method that decides which bitrate to get at the client side is called rate adaptation which will be discussed in the next section.

2.3.4 rate adaptation

There are numerous methods and algorithms proposed for rate adaptation, each using a different parameter to estimate throughput and next segment's bitrate. In this section, we are going to

Figure 2.2: Communications between a streaming server and a client. The playout begins after client downloads the first segment. The client changes the bitrate of the next segment.

examine some major methods and ideas on rate adaptation and discuss the effect of this on overall quality of experience. Mostly, rate adaptation methods are based on an estimation of the available bandwidth. Liu et al., [11] Introduces a method in which the authors define switch-up and switch-down actions. Switch-up is increasing the bitrate of a video and in the introduced method, switch-up happens when the ratio of Media Segment Duration to the Segment Fetch time is more than a certain amount. Basically, it means that the duration of a video segment should be more than the time to download the next segment. This way, the application makes sure of having enough time to download the next segment without interrupting the video playback. The switch-down action happens when the same ratio is less than a certain constant threshold that is set to the application. Depending on the estimated bandwidth some methods choose to either upgrade or downgrade the bitrate of the next segment in a video. However, rate adaptation doesn't necessarily has to depend on the bandwidth. Huang et al., [12] introduces a method that chooses a video rate based only on the current buffer occupancy, and avoids estimating bandwidth. The authors justify the avoidance of bandwidth estimation by stating the fact that often a network environment isn't stable, same as the the dynamics of TCP's congestion control, competing flows, and the loads on the server. Their method examines the buffer for the amount of video playback time it has and the current video rate it is playing. If the buffer is full then it upgrades the video rate and if the buffer is empty it downgrades the video rate. Any other case the video rate stays the same. The work in [26] separates audio and video streams using two TCP connections and finds the best bitrate for each separately. The author, models the problem of finding the best bitrate for either video or audio, as an optimization problem considering each bitrate has a quality and tries to maximize the overall quality of all the downloaded segments. The constraint for their model is that each bitrate can not be higher than a certain amount which is calculated based on throughput estimation. Claeys et al., [27] uses Q-Learning to find the optimized bitrate on client's side. Q-Learning methods usually have a slow reaction to sudden changes which makes the adjustment of bitrate in highly dynamic networks difficult and not suitable. To address this issue, the authors proposed "Frequency

adjusted Q-Learning” which makes a quicker adaptation to the changes in their feedback function, compared to regular Q-Learning methods.

2.3.5 DASH vs. Apple HLS vs. MS Smooth Streaming

Feature	MSS	Apple HLS	DASH
On-demand & Live	yes	yes	yes
Live DVR	yes	no	yes
Delivery Protocol	HTTP	HTTP	HTTP
Scalability via HTTP Edge cache	yes	yes	yes
Origin Server	MS IIS	HTTP	HTTP
Stateless Server Connection	yes	yes	yes
Media Container	MP4	MP2 TS	3GP/MP4
DRM Support for Live and VOD	PlayReady	no	OMA DRM
Supported Video Codecs	Agnostic	H.264 BL	Agnostic
Default Segment Duration	2 sec	10 sec	flexible
End-to-End Latency	>1.5 sec	3 sec	flexible
File Type on Server	contiguous	fragmented	both
3GPP Adaptation	no	no	in work
Specification	proprietary	proprietary	standard

Table 2.1: Comparison of the features among streaming service. Microsoft Smooth Streaming, Apple HTTP Live Streaming, and DASH

In Table 2.1, On-demand & live feature is the ability to stream a live event (*e.g.*, live Basketball match) and stream a previously recorded video (*e.g.*, a movie). Live DVR feature includes, Pause, Seek, Fast Forward, Fast Rewind, Go To Live, Instant Replay, and Slow Motion functionality during a live streaming event. Delivery Protocol is the protocol used to transfer data from a source to its destination. DRM support is the Digital Rights Management which is a systematic approach to copyright protection.

Looking at the comparison of the available features between Microsoft Smooth Streaming, Apple HTTP Live Streaming and DASH, it is clear that the DASH standard’s goal is to maintain the features of already existing streaming solutions and be flexible in many features such as, Segment Duration, Video Codecs, File Type on Server, etc.

2.4 TCP and HTTP Streaming

Transmission Control Protocol (TCP) was mainly designed for wired networks. In wired networks congestion is the main cause for packet loss. Because, random bit error rate (BER) is either zero or negligible. In a transmission, TCP packets are associated with a sequence number, and the packets that are received in-order are considered successful. Thus, they are acknowledged to the sender (ACKs) by sequence numbers of the next expected packet. To eradicate packet delivery failures, TCP has flow control and congestion control algorithms based on the sliding window and additive increase multiplicative decrease (AIMD). There are different versions of TCP congestion control. TCP Vegas [28], TCP Westwood+ [29] and, TCP New Reno [30] are the most known congestion control algorithms. Grieco et al. [31] compares these congestion control methods and demonstrates the superior performance of TCP New Reno. TCP New Reno is also the most widely adopted congestion control method [32]. It consists of four transmission phases: slow start, congestion avoidance, fast recovery, and fast retransmit. TCP has two variables, the slow start threshold (ssthresh), and the congestion window size (cwnd) which is initialized to one maximum segment size (MSS). At the start of a TCP connection, the sending side goes to slow start phase, which means cwnd increases by 1 MSS for every received ACK. The sender enters the congestion avoidance phase when cwnd reaches ssthresh. TCP New Reno employs a sliding-window-based flow control mechanism that allows the sender to increase the transmission window linearly by one after reception of every in-order packet received successfully by the receiver. In the occurrence of packet loss, TCP's fast retransmission and recovery gets activated, that is, the sender reduces its cwnd, causing a lower transmission rate to relieve the link congestion.

Wireless links are affected by many uncontrollable quality-affecting factors such as weather conditions, urban obstacles, large moving objects, and mobility of wireless end devices. As a result, wireless links face much higher BERs than wired links. Besides that, limitations of signal coverage and user mobility cause temporal disconnections that can stall the TCP transmission for a long period, and cause poor performance. A standard TCP like New Reno cannot handle the high

BER and frequent disconnections effectively. The reason is that, all packet losses are seen to be caused by network congestion which is mostly not the case in wireless networks. Random packet losses caused by the high BER of the wireless link would mistakenly trigger the TCP sender to reduce its sending window unnecessarily. The fast retransmit and recovery algorithms can recover fairly quickly if such losses occur once within an RTT. However, in wireless networks, noises and other factors usually cause BERs to occur in short bursts, thus the probability of multiple random packet losses within one RTT is higher [32]. Other than high BER problems, different types of wireless networks pose their own challenges. Wireless ad hoc networks have frequent route changes and network partitions because of high node mobility of the network. In satellite networks, long propagation delay makes the long-range transmission inefficient [32]. One of the main solutions for the aforementioned problems of TCP in wireless environments is to use parallel TCP connections to deliver the desired data. In the next Section, we will explain the reason why parallel TCP connections increase the aggregate throughput compared to a single TCP connection. This solution has been in use in different areas such as file transfer and video streaming which we will cover in the next sections.

2.5 Why parallel TCP improves the throughput

TCP is designed in such a way that its behavior is unnecessarily conservative for data-intensive applications on high bandwidth networks [33]. TCP probes the available bandwidth of the connection by continuously increasing the window size until a packet is lost, then it cuts the window in half and starts increasing the window size again. The higher the bandwidth-delay product, the longer this window size increment will take, and less of the available bandwidth will be used during its duration. When the window of the bottleneck link is large enough to keep the buffer full during this ramp up, TCP's throughput does not degrade. However this requires large buffers on all in-between routers. Furthermore, when there is random loss during the data transfer, it is shown [34] that the link utilization is proportional to $q(ut)^2$, where q is the probability of loss and ut is the

bandwidth-delay product. In order to improve the throughput, parallel connections can be used. This technique is implemented by dividing the data to be transferred into N portions and transferring each portion with a separate TCP connection. The effect of N parallel connections is to reduce the bandwidth-delay product experienced by a single connection because they all share the same bandwidth (u). Random packet losses will usually occur in one connection at a time, therefore their effect on the aggregate throughput will be reduced by a factor of N . As a result, when competing with connections over a congested link, each of the parallel connections will be less likely to be selected for having their packets dropped, and thus the aggregate amount of potential bandwidth which must go through premature congestion avoidance or slow start is reduced [33]. One might argue that HTTP/1.1 pipelining with persistent connection in which the sender can send multiple objects at once, can have the same advantage. But the difference between the two methods is that in HTTP/1.1 pipelining all the data transfer occurs within a single connection. Meaning that, there is still a single TCP window and in the event of packet drop the window size will be reduced by half. Therefore, all the pipelined objects to be transferred are affected by it and the TCP's under-utilization remains a problem in this case.

2.6 Parallel TCP for File Transfer

One of the main applications in the area of parallel TCP connections is in file transfer applications. GridFTP [9] uses parallel TCP to improve the performance of File Transfer Protocol (FTP). Many download manager software such as Internet Download Manager (IDM) [35] and Download Accelerator Plus (DAP) [36] use this technique for increasing the download speed.

GridFTP is one of the main applications to use parallel TCP to improve performance of FTP. GridFTP has five main features that makes it stand out among alternative protocols. One of the main features of GridFTP is parallel data transfer. Using multiple TCP streams in parallel between a single source and destination can improve the total throughput relative to that achieved by a single stream. GridFTP supports such parallel streams using FTP command extensions and data

Figure 2.3: Using parallel connections to download video segments. The method that is used in [10]

channel extensions. Note that data striping and parallel streams may be used alongside (*i.e.*, you may have multiple TCP streams open between each of the multiple servers participating in a striped transfer). Striped transfer is a protocol extension that is defined by GridFTP to support the transfer for partitioned data among multiple servers, because the data intended for transfer may not be located on a single server. Distributed data centers or servers may have striped data or interleaved data across multiple servers.

2.7 Parallel TCP for Video Streaming

In the area of HTTP streaming, the authors in [10] used parallel HTTP connections and suggested to gradually increase the number of connections based on the network conditions, number of parallel connections, and the portion of a segment being downloaded. The main idea in their work is, while downloading a segment, decide whether to make a parallel connection and begin downloading the next segment or not. Figure 2.3 illustrates how the client downloads segments in parallel. Their approach is based on initiating a separate connection for each video segment. The client adapts bitrates (ra), requests a segment ($req\#$), and receives a segment ($rec\#$) one after another. t_0 denotes the current playback timestamp at the time of requesting segment number 1. The available time to download segment number 1 is labeled as $\hat{ft}$, t_{ns} denotes the start timestamp of the

next segment and m_{min} denotes minimum buffered media time. The decision of requesting a new segment (k) in parallel is made by $\phi_k > \mu_k$.

ϕ_k is the ratio of the duration of k^{th} segment that is received, over the total duration of k^{th} segment. μ_k is a threshold for ϕ_k for making a new parallel connection, where the client has already k parallel connections. It is defined as $\mu_k = Ak + C$, which is a linear function and it means the threshold increases with increasing the number of parallel connections.

The work in [37] uses content centric networking (CCN) and multiple network interfaces typically available on a mobile device to implement parallel streaming. It uses multiple connections to download the same video content. However, at any point in time, it uses the connection which delivers data at the highest rate. CCN routers automatically find the best (fastest) link for retrieving specific content. This is because, CCN is agnostic to the used network link and can switch among multiple available links during a data transfer. This characteristic of CCN enables this work to switch between two different CCN servers which both have the same data, and then change the bandwidth of each of the links on a timely period. Their measurements show that DASH over multi-link CCN has a higher *average bitrate* than regular DASH. This work proposes its model for mobile devices, where mobile data connection and WiFi are fetching video chunks and the mobility of user causes bandwidth fluctuations for WiFi and data connection. Therefore, using this method can balance the fluctuations and increases the quality of the video that is viewed by the user. One of the main problems with this method is power consumption. When data connection and WiFi are both connected and being used, the mobile device drains its battery on a faster pace. Also, note that this work uses parallel links to download the same chunk and decides to download from the link that has a higher available bandwidth at the time of request. While, our work uses parallel connections, where each connection downloads a portion of a chunk.

The work in [7] uses multiple connections to reduce throughput fluctuations in networks with high packet loss and RTT. It uses 10 parallel connections to overcome the poor performance of TCP in such networks, though the main focus is to maintain fairness among connections with favorable

and poor network conditions in terms of packet loss and RTT. As depicted in Figure 2.4, video

Figure 2.4: Using multiple http streaming queues to download video chunks from server. Used in [7]. Packet number 4 didn't successfully transmitted to the client. Therefore, it is added to two separate queues to increase its chance of delivery.

chunks are numbered in order of appearance. Each stream fetches the chunks in the same order as located in the queue (round robin). When the transfer of a chunk gets interrupted and reaches the specified timeout, it will be requested by two other queues to increase the probability of a successful transmission. Two queues requesting the same chunk does not affect the friendliness because the back-off in time of congestion is done implicitly by TCP. This work also uses multiple connections to download different video chunks. As opposed to our work, this one also doesn't consider the competing parallel streams. For example, in the beginning of streaming, 3 streams download chunks number 1, 2, and 3 respectively and if the available bandwidth is limited, the second and third stream are restricting the bandwidth for the first chunk. This may cause stall in video playback and it isn't considered in this work.

The authors in [14] use multiple TCP connections to enable users to play different parts of the video without having to wait for buffering of the entire video. As an example, if we have a 60 seconds long video, and we have 3 parallel connections, first connection starts to fetch the video chunks from the beginning, the second connection starts from second 20 and the third connection starts from second 40. This way, if user wants to jump to second 20 of the video the waiting time for video buffering will be zero. Finally, Chaurasia et al., [38] uses multiple connections to download different layers of the video with scalable video coding. Each layer adds more to the

quality of the video and the first layer is the base. Layers that come after that, are enhancement layers. Even though this work uses parallel connections to transfer the video, it does not use the parallel connections to transfer video chunks. Parallel connections are used to transfer encoded video layers. In the experiments in this work, 2 parallel connections are used. One for transferring the base layer and the other for transferring the enhancement layer.

2.7.1 Quality of Experience Factors of HTTP Video Streaming

For HTTP video streaming, [39], [40] show in their results that initial delay and stalling (the number of stalling events and length of a single stalling event) are the key influence factors of QoE. The work in [41] also compares the adaptive and non-adaptive video streaming and states that quality adaptation can reduce stalling by 80% when the available bandwidth drops, and makes a better utilization of bandwidth when bandwidth increases. Also the work in [42] confirm that it is better to control the quality (video bitrate) than to suffer from uncontrolled stalling events. In the following, we will review the three aforementioned QoE factors.

- Initial delay: In a multimedia streaming service, a certain amount of data must be transferred before video playback can begin. Initial delay in such services is defined as the time spent from user requesting the video until the beginning of video playback. In most applications the video playback is delayed (*i.e.*, higher initial delay) more than technically required in order to fill the playout buffer. Filling the playout buffer is an efficient tool used to deal with short term throughput fluctuations [43].
- Stalling event: Stalling event is simply the stopping of video playback due to playout buffer run out. This event happens when the throughput of video streaming application is lower than the video bitrate. Eventually, insufficient rate of data arrival in the buffer makes an interruption in video playback and a stalling event occurs. Stalling of video playback continues until the buffer receives a certain amount of data. Here again, the amount of buffering playtime can be more than necessary

to make a tradeoff between the length of the interruption and the risk of recurring stalling event.

- Average bitrate: To have adaptation in video streaming, the client should control which data rate is suitable for the current conditions, because the client can measure its current network conditions at the edge of the network. On the server side, the video is split into small segments and each of them is available in different quality levels(different bitrate levels) [43]. The average bitrate is a parameter which indicates the average of selected bitrates of the transferred video segments. The higher value for this parameter shows the quality of the video that is received at the user end is better.

Initial delay and stalling are perceived by human sensory system in different ways. Initial delay is expected by user, as opposed to stalling that invokes a sudden interruption in video playback. Hence, stalling events are perceived much worse than initial delays as they happen unexpectedly without the anticipation of the user [44]. The authors in [45] show that even though video interruptions due to rebufferings are experienced as disturbing, users accept a (limited) number of these rebufferings in a mobile context. In [46] the authors study the two cases of frame skipping and frame freezing (stalling event). In their experiments they illustrate two findings. First, increasing the duration of stalling decreases the quality. Second, one long stalling event is preferred to frequent short ones. In [47] the authors study both frame rate reduction and stalling events. They compare the two factors and show that stalling is perceived worse than frame rate reduction by users. Furthermore, they compare the time period of stalling events and find that stalling at irregular intervals is worse than periodic ones. The authors of [39] propose a mapping model for regular stalling patterns to mean opinion score (MOS). They find an exponential correlation between stalling parameters and MOS. They also find that users tolerate only one stalling event per clip at most, if its duration is in order of a few seconds.

Chapter 3

Methodology

3.1 Overview

In this chapter, we review the solution to the problems mentioned in Chapter 2. First, we introduce a model in order to find the number of parallel TCP connections needed to fully utilize a channel. In this section, we cover two different scenarios, one where the transmit queue between a server and a client is small and it causes packet loss due to buffer overflow. We also show two different models for when there is a small number of parallel connections (packet approximation) and when there is large number of parallel connections (fluid approximation). Next, we go through our solution and describe the details of our implementation. First, we explain the implementation of our system which uses parallel connections for the same video chunk (ParS). We describe the structure of the system, responsibility of each component and then we describe the whole system using a flowchart. We show each step of the program from starting the playback till ending. In addition to that, we review our simulations that we used to validate our model.

3.2 Model

3.2.1 TCP Modeling

A notable work on characterizing the effect of parallel connections on aggregate TCP throughput is [48], which provides informative measurements for an upper bound of aggregate bandwidth using parallel connections with different value for Maximum Segment Size (MSS). Also Lee [33] explains the reason why using parallel connection improves the aggregate throughput. Selecting the number of parallel connections is also suggested based on loss rate, round trip time, and MSS. However, it lacks the calculation and formula for finding out the number of connections based on

the mentioned parameters.

The work in [49] provides a closed form expression for the throughput of parallel TCP connections assuming the congestion windows have no synchronization or are weakly synchronized, there is no transmission losses, and no buffering at the bottleneck router. Under these assumptions, it is shown that the effect of parallel connections on the aggregate throughput diminishes inversely proportional to the number of connections. Assuming at any congestion event, only one connection is signaled for congestion, then the equation to determine the total throughput for the mentioned work is [49]

$$\frac{X}{c} = 1 - \frac{1}{1 + \frac{1+\beta}{1-\beta}N} \quad (3.1)$$

Having N parallel connections, X is the total throughput, c denotes the link capacity, and at the time of congestion a connection's rate decreases to a fraction β . In the asymptotic regime of a large number of connections, [50] and [51] study the throughput behavior of parallel connections and show a linear relation between the aggregate throughput and the number of connections. None of the models developed in these works are applicable to our scenario, where the number of connections is limited and the size of the buffer at the router is arbitrary.

3.2.2 Number of TCP Connections

So far, we have shown that it is feasible to use parallel TCP connections to download each chunk of a video file, and that this approach results in higher throughput and lower chunk download time. An important question then is how many parallel connections are required under different network conditions (*e.g.*, RTT, loss rate, *etc.*). Increasing the number of TCP connections adds complexity to the client application. In particular, mobile devices are usually less powerful compared to desktop machines in terms of computational power. Most web browsers on mobile devices actually limit the number of parallel connections per-server for this reason (*e.g.*, Google Chrome limits the number of parallel connections to 6). Therefore, we like to have as few parallel connections as possible. In general, it is difficult to find the precise number of connections required since it

Figure 3.1: Network topology used in simulations. Internet link imitates the link between the access router and the server over the Internet. The access link imitates the link between the access router (*e.g.*, home WiFi router) and the end device.

depends on dynamic network conditions, the interplay between TCP congestion control and buffer management policies at routers, and specific TCP implementation details such as RTO estimation and so on.

In this section, we develop a simple model to estimate the number of parallel connections required to fully utilize the access link (*i.e.*, bottleneck link) of the end-to-end path. We also present ns-3 simulation results to verify the accuracy of our model in various network scenarios. Since our focus is on wireless networks, we consider a scenario where the bottleneck link is at the edge of the network (*i.e.*, the wireless access link is the bottleneck) as depicted in Fig. 3.1. We consider two cases in the following sections depending on the size of the transmit queue of the router that is attached to the bottleneck link. Note that in order to forward packets on its outgoing link, the router first copies the packets to the transmit queue associated with that link. Once the outgoing link becomes available then the packet at the head of the queue is transmitted.

3.2.3 Small Transmit Queue

In this case, congestion losses due to buffer¹ overflow at the router dominate the packet loss process seen by TCP. We consider two models to approximate the aggregate throughput of parallel connections in this case.

¹We use the terms “buffer” and “queue” interchangeably.

Fluid Approximation.

Assume that there is only one flow between the client and the server. A commonly used approximation for the TCP throughput is given by $X = \frac{3}{4}C$, where C denotes the capacity of the bottleneck access link [52]. Clearly, a single flow is not able to fully utilize the bottleneck link in this case. To determine the effect of multiple flows, we make the following (optimistic) approximation about the aggregate throughput (thus, the result is an upper bound). We assume that the first flow utilizes 3/4 of the link bandwidth and the additional flows utilize the remaining 1/4 of the bandwidth. Thus, the aggregate throughput with N parallel connections, denoted by X_N , is given by,

$$X_N \approx X_{N-1} + \frac{3}{4}(C - X_{N-1}) \quad (3.2)$$

$$= \frac{3}{4}C + \frac{1}{4}X_{N-1}, \quad (3.3)$$

which yields,

$$X_N \approx (1 - \frac{1}{4^N})C. \quad (3.4)$$

As can be seen from the above equation, while using multiple flows improves the aggregate throughput, only a few flows are sufficient to fully utilize the bottleneck link.

The fluid approximation is valid for large number of flows (*i.e.*, the asymptotic fluid regime). For small number of flows however, it is inaccurate. Specifically, if the size of the transmit queue is 0, in reality, the achieved throughput of a single flow will be 0, and not 75% of C as predicted by this model. However, in practice, routers do have non-negligible buffers. In fact, most edge routers (such as the WiFi APs) suffer from the bufferbloat problem [53] due to excessive buffering at the router. Thus, the fluid approximation can be applied, and later, using ns-3 simulations we show that it is a reasonable approximation in some scenarios.

Packet Approximation.

Consider a router with transmit queue of size 1 packet. The time to transmit a packet over the bottleneck link is given by L/C , where L denotes the length of the packet. For the sake of argument,

assume that the system operates in a discrete time manner where time is divided into time slots of length L/C (long enough to transmit one full packet). In each time slot, zero or more packets arrive at the router but at most one is transmitted and the rest are dropped. Let

$$K = \frac{R}{L/C}, \quad (3.5)$$

denote the number of time slots in an RTT where R indicate RTT in the above equation. Consider the following two cases:

1. *One Flow Between Client and Server*

In this case, the server transmits one packet in the first RTT, receives the corresponding ACK and increases its window to two packets. In the second RTT, it sends two packets back-to-back. The first packet arrives at the router and is immediately scheduled for transmission. The second packet arrives while the first one is being transmitted (recall that the bottleneck access link is slower than the Internet link) and hence is dropped. After some time, the server receives an ACK for the first packet and increases its windows size to 2.5 packets. However, it never receives an ACK for the second packet which results in a time-out. It follows from this argument that the flow throughput is bounded by:

$$X = \frac{L}{R + L/C + RTO}. \quad (3.6)$$

In this equation, RTO denotes the retransmission time-out which typically has a minimum value of 1 second in practice [54] (but 200 ms in ns-3).

2. *Multiple Flows Between Client and Server*

In this case, the aggregate throughput is clearly a function of the degree of synchronization among the flows. If they are fully synchronized, then the aggregate throughput is equal to the throughput achieved by a single flow. If they are fully desynchronized, then the bottleneck link could be fully utilized if there are sufficient

number of flows (at least K flows). In this case, the aggregate throughput increases linearly with the number of flows until it reaches the bottleneck link capacity C . In the following, we consider the case when N flows are randomly synchronized. For ease of exposition, ignore RTO and packet transmission time L/C . With “random synchronization”, we assume that each flow transmits in a time slot that is uniformly chosen at random in one RTT, which has K time slots. The probability that a given time slot remains idle (no flow transmits during that time slot) is given by $(1 - \frac{1}{K})^N$. Thus, the expected aggregate throughput is given by:

$$X_N = \left(1 - \left(1 - \frac{1}{K}\right)^N\right)C. \quad (3.7)$$

Since K is constant, as N increases, the aggregate throughput approaches C quickly, which is consistent with the fluid model’s prediction.

3.2.4 Large Transmit Queue

In this case, even one connection could fully utilize the bottleneck link if a sufficiently large transmit queue is installed on the router (in contrast to what is predicted by some existing models). Given a large queue, the packet loss process seen by TCP is dominated by *random transmission losses*. For a TCP connection with round trip time R and transmission loss probability p , the average throughput (in units of “segments”) is given by [52]:

$$X = \frac{\phi}{R\sqrt{p}}, \quad (3.8)$$

where, ϕ is a constant that depends on the specific congestion control algorithm used. For TCP New Reno, we have $\phi \approx 0.3$. Our goal is to fully utilize the access link. Let Q denote the average queue length that is required to fully utilize the link. Also, let τ denote the two-way propagation delay between the client and server. We have, $R = \tau + Q/C$, and thus,

$$(1 - p)X = C, \quad (3.9)$$

After substitution for X , it yields,

$$(1-p) \frac{\phi}{(\tau + Q/C)\sqrt{p}} = C. \quad (3.10)$$

Therefore,

$$\begin{aligned} \frac{\phi(1-p)}{C\sqrt{p}} &= (\tau + Q/C) \\ \Rightarrow Q &= \frac{\phi(1-p)}{\sqrt{p}} - \tau C. \end{aligned} \quad (3.11)$$

In order to have $Q \geq 0$, the following condition should be satisfied,

$$\begin{aligned} \frac{\phi(1-p)}{\sqrt{p}} &\geq \tau C \\ \Rightarrow p &\leq \left(\frac{\phi}{\tau C} \right)^2. \end{aligned} \quad (3.12)$$

In other words, if the packet loss probability p is very small, then even a single flow can fully utilize the bottleneck link. However, if the packet loss probability p is large (specifically, $p > (\phi/(\tau C))^2$) then, one flow cannot fully utilize the link. In this case, multiple flows are required to achieve an aggregate throughput of C . The number of flows can be computed as follows:

$$X_N = (1-p)NX = C. \quad (3.13)$$

Thus, we obtain that,

$$\frac{N\phi(1-p)}{R\sqrt{p}} = C, \quad (3.14)$$

and, consequently,

$$Q = \frac{N\phi(1-p)}{\sqrt{p}} - \tau C. \quad (3.15)$$

Given that $Q > 0$, we have,

$$\frac{N\phi(1-p)}{\sqrt{p}} \geq \tau C, \quad (3.16)$$

which yields the following relation for the number of flows,

$$N \geq \frac{(\tau C)\sqrt{p}}{\phi(1-p)}. \quad (3.17)$$

3.3 Parallel HTTP Streaming (ParS)

3.3.1 Overview

In this section, we will review the practical solution for the aforementioned problems for TCP in 2.4. Section 3.2 demonstrated the idea of parallel connections and the number of connections to use depending on the structure of the network and queue size. Here, we will describe our implemented application which is used to run our measurements in wired and wireless networks. Also, we will explain our simulation implementations in ns-3 [55].

3.3.2 System Description

The idea of using parallel connections to improve the throughput and overcome TCP's poor performance is not a new concept. It also has been used in video streaming. We covered some of those works in Section 2.7. But the problem was that all these aforementioned works use each parallel connection for a different video chunk. Figure 3.2 demonstrates how each parallel connection downloads a different chunk. We call this method Traditional Parallel Streaming (TPS). TPS can be useful when there is high bandwidth available in the network and the packet loss rate is low. But in wireless networks, where the packet loss rate is usually high and the available bandwidth is low, TPS not only doesn't improve the performance, it also reduces the available bandwidth for the connection that is downloading the next video chunk that is about to be played. This problem is shown in Figure 1.2. This causes multiple playback interruptions (stalls) and lower video bitrate for the next video chunk. As a result, it reduces the quality of experience for the user in wireless networks.

In order to avoid the aforementioned problem in wireless networks, we use parallel connections to download the same video chunk (the next chunk that hasn't been downloaded and is about to be played). Figure 3.3 demonstrates the HTTP requests that are sent to the server requesting the same video chunk. In this case, there are two parallel connections and the first one requests the first half of the chunk using Range feature in HTTP-GET request. The next one requests the second half of

Figure 3.2: Traditional Parallel Streaming (TPS) approach for using parallel connections in video streaming. each connection gets a separate chunk.

the video chunk. The Range feature has been discussed in Section 2.1.

3.3.3 System Implementation

In this section we will examine the implementation details of our system (ParS), which was used in our measurements over wired and wireless networks. It is written in JAVA and requires Java Runtime Environment (JRE) to run. The program has only one Java class called ParS which contains 4 methods: Main, getFileSize, sendGet, and rateAdapt. The collaboration and responsibilities of each of these methods are demonstrated in Figure 3.4. The written code is included in Appendix A.1.

Figure 3.3: Using parallel connections to download the same video chunk (ParS). All connections download the same chunk but different parts.

rateAdapt

This method takes the download rate (bits per second) as input and finds the closest video bit rate of segments according to the MPD file. We run the experiments using a dataset that, each segment in it has 15 seconds duration and there are 20 different bit rates for each segment. The smallest video bit rate for the segments is 45 kbps and the largest one is 3.7 Mbps. Depending on the download rate that is given to this method, it will decide which bit rate to download for the next segment of the video playback.

getFileSize

As mentioned in the description of ParS in Section 3.3, in order to send parallel HTTP requests to using HTTP-Range, the client needs to know the exact file size for the next segment so that the

Figure 3.4: Methods and their responsibilities in ParS.java implementation.

byte-range of each request can be calculated. Thus, to get the size of the next segment, we send an HTTP-HEAD request, to obtain the information about the next segment including its size. This method gets the bit rate and the file name of next segment, then it forms an HTTP-HEAD request and sends it to the server. After receiving the HTTP-response, it extracts the file size from it and returns the file size. This way, before requesting any segments, the application knows its size and can form the proper parallel HTTP-GET requests and their byte range. Figure 3.5 shows a sample HTTP request where the Range feature is used to download a part of a .mp4 file between the byte ranges of 1000 and 2000. Also, the HTTP response for this request is “206 Partial Content” instead of the typical “200 Ok” response that is received by HTTP requests.

sendGet

This method has three main responsibilities, forming a proper HTTP-GET request, receiving the response, and storing the response (*i.e.*, video chunk). For the first part, the method takes the file name, bit rate, start range, and end range. The bit rate indicates the bit ratio of the next segment to

```
GET /BigBuckBunny_320x180.mp4
    Connection: keep-alive
    Accept-Language: en-GB,en-US,en
    Host: localhost:8080
    Range: bytes=1000-2000
    Accept: */*
206 Partial Content
    Content-Type: video/mp4
    Connection: keep-alive
    Last-Modified: Wed,14 Dec 2011 15:50:59 GMT
    ETag: A023EF02BD589BC472A2D6774EAE3C58
    Transfer-Encoding:
    Content-Length: 1000
    Accept-Ranges: bytes
    Server: Brisket/1.0.1
    Date: Wed,14 Dec 2011 16:11:25 GMT
```

Figure 3.5: HTTP-GET sample request and response using byte-range feature.

download. Start and end range indicate the byte range of the segment to download, and file name is the name of the segment stored on server. The client knows the file names and their bit rates, from the MPD file downloaded at the beginning of the streaming process. The start and end range numbers are also within the limits of the file size, which is derived from `getFileSize` method.

Main

As it is illustrated in Figure 3.6, the main function makes use of all three aforementioned functions to complete the process of parallel HTTP Streaming (ParS). At first, it takes the number of parallel connections (*connumber*) as an input. Then, it downloads the MPD file and stores the information from it. The number of segments, path to the segments, file names of the segments, and available bit rates of the segments are all derived from the MPD file. After receiving the necessary infor-

mation, the method, starts the main loop to download all the segments. Inside the loop for each segment, the main method creates the file name of the segment, then finds the bit rate by using `rateAdapt` function and gets the segment size by using `getFileSize` function. The byte-range of each request depends on the number of parallel connections and size of the segment. Dividing these two number (considering the remainder for last request) gives us the byte-range. After getting all the information to form the parallel connections, an array of Java Threads with the size of the number of parallel connections is created. Furthermore, in a loop that repeats itself *connumner* times, the start range, end range and port number of each connection is formed and a new Thread is created in which the `sendGet` method is called. The newly created Thread is added to the array of Threads and after adding all the Threads, they will start. Each thread calls the `sendGet` method with a specific start and end range for the next segment. After all the segments are received by the client, the main method calculates the download rate by starting a time stamp before sending requests and stopping it after all the threads are done. This way, the time spent for downloading all the subsegments (a specific range of a segment) is calculated and dividing the file size by this number gives us the download rate. The download rate is further used for rate adaptation of the next segment. In the beginning of the streaming, first segment is always downloaded with the lowest bitrate. Eventually, when all the segments are downloaded, the Quality of Experience influence factors such as, time to start playback, number of stalls, duration of stalls and the bitrate and download rate of each segment is measured and saved into a log file.

3.3.4 Simulation Implementation

Evaluation of a TCP model requires a highly controlled network environment, where user is able to modify network parameters. Therefore, we decided to run ns-3 [55] simulations in order to test the validity and accuracy of our model which is described in section 3.2. The topology used in the simulation is shown in Figure 3.1. In order to simulate an applicable scenario, we put two end devices as client and server. The server plays the role of a web server or a streaming server over the Internet. The client is the end user with a hand-held device or any other device that is able to

communicate with a server over the internet via HTTP/TCP. Internet link is the link between the server and the router, which has a high bandwidth and a high latency. Access link is link between the router and the end user's device which, has a low bandwidth and a low latency (compared to the Internet link). The router in between has a queue with limited capacity on the access link side (outgoing packets from the server to the client). The queue size has been configured to different values in order to study its impact on the overall throughput of data transfer between the server and client. The simulation code is written in C++ and it is included in Appendix A.2.

The ns-3 simulations are written in C++ language. They use ns3 namespace and ns-3's libraries for different applications (*e.g.*, point to point module, WiFi module, ipv4, and ipv6 module). Our simulation code consists of seven sections: Initializing variables, getting input arguments, creating the topology, creating the applications, setting up monitoring utilities, running the simulation, and making the log files. Each of these sections will be described below.

Initialize variables

In this part, we set the default variable values for when the user doesn't specify them in input arguments. The initialized variables are simulation time, number of parallel TCP flows, packet loss probability, queue size for the router, and TCP packet size.

Getting input arguments

This section defines which variables can be modified by user in the form of input arguments. Packet loss probability, queue size of the router and delay between the router and the client, and number of parallel TCP flows are the variables that can be controlled by input arguments.

Creating the topology

In ns-3, every entity in the network is a node. We created three nodes, two for server and client, one for the router in between. After creating the nodes, we set up an error model for packet loss on the links. It is set as uniformly distributed with the probability defined by input arguments. Furthermore, we define the links between the nodes and specify the delay and the bandwidth for

them and then install the links on the nodes. The Internet link is set to have 100 Mbps bandwidth and 20 ms delay. Queue size for the router in between is also set in this section. Queue size for incoming packets from server is set to 1000 packets (large queue size) and the queue size for outgoing packets to the client is defined by input arguments. Finally, the IP addresses and IPv4 routing is installed on the nodes and the links.

Creating the applications

In this part, we make the applications for source and sink nodes. Since each application uses a port and a destination IP address, we specified all the sink applications to use the same port. But on the source side each application uses a different port number. This way, applications are using different sockets and we can have multiple streams of packets going from the same node and reach the same destination, each having a separate TCP connection. This whole section's code is inside a loop whose number of iterations is equal to the number of parallel connections. The number of parallel connections, as mentioned, is specified by the input arguments.

Setting up monitoring utilities

In this section, we initialize and enable the monitoring tools in ns-3. There are several utilities that can be enabled such as netAnim (for animation of network topology and data flow), flow monitor (for measuring the amount of data transferred on each connection by source and destination IP and port), and pcap tracing (for making dump files of the packets transferred among the nodes which can be processed using Wireshark). We used flow monitor tool for measuring the throughput of each TCP connection and the total throughput. For other measurement, we made our own log files keeping track of the variables and changes in them. For example, for tracking the number of packets in the queue and TCP window size, during the simulation, we used a time-based event to call back a function on a timely basis. In the function, we saved the values of TCP window and number of packets in the queue.

Running the simulation

In this part, we simply set the duration that we want the simulation to run. Then we set the time-based events and specify their functions to callback. Then, finally we start the simulation.

Making the log files

Some logging information needs to be saved during the simulation because after the simulation is done, that information will be gone as well. The number of TCP packets in the queue is one of them. But other types of logging information such as throughput of each connection and the total throughput need to be calculated after the simulation is done. In this section, we use the flow monitor utility to get the number of bytes that is transferred from server to the client and divide that by the duration of simulation. Then, save this information along with other values of the simulation to the log files.

Figure 3.6: Main function's flowchart

Chapter 4

Results

In this chapter we will examine the results of the simulations and experiments that have been conducted to validate the proposed model in chapter 3 and study the performance of ParS. To demonstrate ParS system's efficiency and advantage over Traditional Parallel Streaming (TPS). We have implemented both systems and tested them under various WiFi network conditions in order to cover different network conditions and emulate a real wireless network. In the experiments, we measured QoE factors such as *time to start playback*, number of playback stalls, and bitrate of each video chunk and then calculated the average bitrate of the whole video. Our results demonstrate the superiority of ParS system over TPS in average bitrate of video playback. Also, it maintains its fast time to start playback despite the increment in number of parallel connections. In simulation results section, we illustrate the validity of the proposed model in Section 3.2. The calculations for both cases of small transmit queue and large transmit queue are studied and evaluated in this chapter.

4.1 Implementation results

4.1.1 Testbed description

The client and server systems in sections 4.1.2 and 4.1.3, are laptop computers connected via a wireless router the testbed for both cases are depicted in 4.1 and 4.5. The client system is a Core i7 3.18Hz machine with 8GB of RAM and runs Microsoft Windows 10. It has Java Runtime Environment (JRE) installed to run the java codes written for ParS and TPS. The server system is a Core i5 2.66Hz machine with 4GB of RAM and also runs Microsoft Windows 10. Apache webserver is used on the server system and the video file used is BigBuckBunny [56]. The DASH dataset for the video is taken from [57]. To have a strong wireless signal between the server and

Number	Bitrate (bps)
1	45351
2	88563
3	127052
4	176031
5	215976
6	252582
7	315047
8	365887
9	500530
10	565835
11	769540
12	982168
13	1165885
14	1419205
15	2061491
16	2372196
17	2863389
18	3222578
19	3464259
20	3748236

Table 4.1: Numbers and the bitrates they represent in bitrate frequency figures for the video dataset [57].

the wireless router, and also between the client and wireless router, the transmission power of the wireless router is increased to 100%. Both server and client systems are placed within a short distance of the wireless router without any blocking objects in between. For the case of weak wireless signal, the transmission power of wireless router is set to 20%. Then, the client and server systems, each are put in different rooms from each other and the wireless router. This way, the physical distance of the router and the client/server is increased. Also, the obstacles in between (the walls) would weaken the signals propagating between the systems and the wireless router.

4.1.2 Measurements - weak signal

In this section, we will review and discuss the measurement results derived from weak signal testbed. As mentioned earlier, in this testbed, we reduced the router's transmission power to 20%.

Figure 4.1: WiFi test bed for weak signal test cases. The server and the client are each located in a different room on 2nd floor. The router is located on the main floor to have the maximum distance from the client and the server.

Also the server system is covered with aluminium foil to weaken the WiFi signals received from the router. The figures in 4.2 are the frequency of bitrates selected by the application. Note that the rate adaptation makes the application choose the bitrate that is closest to the download rate. Each figure shows the results of 5 runs using a certain number of parallel connections. Starting from 2 parallel connections and ending with 10. The figure for 1 connection isn't included because both methods (ParS and TPS) behave exactly the same in this case. Both methods in this scenario, use a single connection to download a video chunk, therefore comparing them doesn't have any purpose. In each figure, the blue bars show the frequency of selected bitrates with ParS method and yellow bars show the frequency of selected bitrates with TPS that use parallel connections to download different chunks.

(a) 2 parallel connections

(b) 3 parallel connections

(c) 4 parallel connections

(d) 5 parallel connections

(e) 6 parallel connections

(f) 7 parallel connections

(g) 8 parallel connections

(h) 9 parallel connections

(i) 10 parallel connections

Figure 4.2: Bitrates selected for ParS compared to TPS from 2 parallel connections to 10, in the weak WiFi signal environment. The aggregate frequency of 5 runs are demonstrated. The x-axis numbers represent the bitrates shown in Table 4.1.

Prior to discussing the behavior of the two methods and examination of the figures, we should note that the bitrates are numbered from 1 to 20 on X-axis, 1 being the lowest bitrate and 20 being the highest. The numbers are representing the bitrates but they are not the actual video bitrates on the server. The range of these numerical numbers are chosen to increase readability and comprehensiveness of the figures. The actual bitrates are listed in Table 4.1.

By examining the figures in 4.2, we can observe the following characteristics:

- In all cases the highest bitrate has the most frequency and that is because the bandwidth in the testbed was sufficient enough for the highest bitrate. Also, the network didn't have other nodes or streams to compete for bandwidth. Therefore, the only thing that was lowering the bandwidth was the weak WiFi signal and the fluctuations that it causes.
- As mentioned earlier, the first video chunk is always downloaded with the lowest bitrate. This is to prevent time-to-start playback to be high in low bandwidth networks. In ParS, only the first video chunk gets downloaded with the lowest bitrate but in TPS, "the first set of chunks" get downloaded with the lowest bitrate. The reason for that is, changes in bitrate selection happen after downloading every n chunks where n is the number of parallel connections and each connection downloads a separate chunk. Thus, in the beginning, if there are 5 parallel connections, the first 5 chunks are downloaded with the lowest bitrate and then the rate adaptation measures the download rate and changes the selected bitrate.
- In TPS, increasing the number of parallel connections, increases the frequency of higher bitrates and decreases the frequency of lower ones. This has two reasons. First, having parallel connections utilizes more of the available bandwidth therefore, more chunks with higher bitrate are selected. Second, the frequency of bitrate-change decreases as the number of parallel connections increases. For example, in the case of having 2 parallel connections, rate adaptation adapts the bitrate after

downloading every 2 chunks. But in the case of having 10 parallel connections, bitrate adaptation happens after every 10 chunks getting downloaded. This means the client application has less chance to change its bitrate during video playback therefore the rate of change in selected bitrates is slower. It is important to note that this scenario does not happen in ParS since, rate adaptation can change the bitrate after downloading every chunk regardless of the number of parallel connections.

- In all cases, ParS selects more video chunks with higher bitrates than TPS. Looking at the figures, we can see that ParS is choosing more of the higher bitrate chunks compared to TPS which means, the quality of the video that ParS is playing is better. Considering the fact that both methods are running in the same environment and the same test bed, in WiFi networks with weak signal, we can see that ParS is outperforming TPS in terms of the quality of the video playback.

We will demonstrate the average bitrate by both methods when using different number of parallel connections. Later, we compare the time-to-start playback of both methods.

Figure. 4.3 shows the average bitrate for different number of parallel connections. The average bitrate is derived by calculating the weighted average of each bitrate and its frequency. For n parallel connections the average bitrate is calculated by the following statement:

$$avgBitrate_n = \frac{\sum_{i=1}^k Freq(bitrate) * bitrate}{k * r} \quad (4.1)$$

where k is the number of different bitrates, and r represents the number of times we have run the program. When there is a single connection, the average bitrates for both methods are close with a slight difference. That is because in case of a single connection, both methods behave similarly as mentioned earlier. By increasing the number of parallel connections, a clear decrease is visible in parallel connections for separate chunks while, the average bitrate for ParS has a slight fluctuation but it maintains an average bitrate above 6 Mbps. Also, in case of having parallel connections (2, 3, ..., 10 connections), ParS always has a higher average bitrate than TPS. The

Figure 4.3: Average bitrate for ParS compared to TPS. This is the weighted average of selected bitrates and their frequency from 1 connection to 10 parallel connections. The numbers are derived from the same experiments that are conducted for Figure 4.2.

Figure 4.4: Time-to-start playback for ParS compared to TPS in weak WiFi signal. The average of 5 runs are represented in the figure. The x-axis shows the number of parallel connections.

figure clearly demonstrates the advantage of using ParS over TPS in terms of video quality during a video playback.

Time-to-start playback as discussed in Chapter 2 is one of the quality of experience influence factors in video streaming. Figure 4.4 shows this factor for ParS and TPS under weak WiFi signal. We ran the experiment 5 times and got the average time-to-start for the figures 4.4, 4.8, and 4.9. The difference between the two methods which makes ParS to have lower time-to-start playback is that ParS downloads only one chunk at a time. This approach has a strong advantage over TPS's approach during playback of a video. As it is demonstrated in the figure, by increasing the number of parallel connections the time to start playing the video increases up to 1834 milliseconds for TPS. This value can get higher for higher number of parallel connections because as we increase the number of connections there will be more video chunks to download before the playback of the video can start. This problem doesn't exist for ParS system since it uses all connections to download one video chunk and after downloading the first chunk the video playback can start. That makes time to start playback of the ParS to stay under 400 milliseconds regardless of how

Figure 4.5: WiFi testbed for strong signal test cases

many connections are being used. This advantage of having a quick time to start playback gives ParS another advantage over TPS in terms of QoE.

So far we reviewed two of the most important quality of experience factors (bitrates for each chunk and time to start playback) in an environment where the WiFi signal is weak and thus the bandwidth is limited. In next sections, we will review the same QoE factors under strong WiFi signals and also in a university campus environment to evaluate the performance of ParS in a real network environment.

4.1.3 Measurements - strong signal

In this section, we will review the results of our measurements under strong WiFi signal and coverage. Unlike section 4.1.2, the transmission power of the router is set to maximum. Also, server and client are both in the same room as the router. This is to prevent having any physical object in between the systems and the router and eliminate the chance of receiving weakened signals. Figure 4.5 demonstrates the structure of the test bed and the location of client and server systems. As it is mentioned, both client and server are in the same room as the WiFi router. Similar to measurements in weak WiFi signal, first the frequency of bitrates selected is compared between ParS and TPS. There is a total number of 9 figures each using a different number of parallel connections starting from 2 parallel connections to 10 parallel connections. In each figure, both programs (ParS and TPS) are run 5 times and the total frequency of each selected bitrate is shown in the figures. The numbers on the x-axis represent a specific video bitrate which is demonstrated in Table 4.1. The lowest bitrate is *45kbps* which is represented by number 1 on the figures and the highest bitrate is *3.7Mbps* which is represented by number 20 on the figures. After comparing the frequency of bitrates we examine the average bitrates for both methods to get a better understanding of the distribution of selected bitrates in each run with both methods. Finally, we compare the time to start playback for the two methods which is another factor for defining the quality of experience for the user.

(a) 2 parallel connections

(b) 3 parallel connections

(c) 4 parallel connections

(d) 5 parallel connections

(e) 6 parallel connections

(f) 7 parallel connections

(g) 8 parallel connections

(h) 9 parallel connections

(i) 10 parallel connections

Figure 4.6: bitrates selected for ParS compared to TPS from 2 parallel connections to 10, in strong WiFi signal environment. The aggregate frequency of 5 runs are demonstrated. The x-axis numbers represent the bitrates shown in Table 4.1.

Reviewing the above figures 4.6, gives us the following conclusions about the two methods in a strong WiFi signal environment:

- By increasing the number of parallel connections the frequency of the lowest bitrate constantly increases with TPS. That is because when the number of parallel connections increases, the number of chunks to download before the video playback begins increases too. Also, the first set of chunks always get downloaded with the lowest bitrate to prevent long stalls and playback start times at the beginning of playing a video. This affects both bitrate selection frequency and average bitrate in general, and time to start playback which we go through later in this section.
- In all figures the number of chunks selected with the highest bitrate for ParS is more than TPS. Thus, we can assume users would have a better experience watching a video using ParS method than using TPS.
- Increasing the number of parallel connections doesn't eliminate the selection of lower bitrates. As it is shown in figure 4.6, selection of lower bitrates can not be eliminated. This is because of the unpredictable behavior of the wireless networks and fluctuations in available bandwidth. Packet drops even in strong signal WiFi environments are inevitable and can affect the available bandwidth and therefore the bitrate selection of the video. Figure 4.6d shows that a few number of packets are chosen with bitrates less than the maximum.

In this part, we compare the average bitrate of ParS and TPS in WiFi networks with strong signal. Figure 4.7 depicts the average bitrate with multiple parallel connections. The average bitrates are calculated for n parallel connections using 4.1.

The comparison of time-to-start playback in Section 4.1.2 showed that ParS consistently maintains a low time-to-start when playing a video. Figure 4.8 also depicts a similar behavior in which, ParS's time-to-start playback is consistent and less than 200 ms. But the TPS increases its time to

Figure 4.7: Average bitrate for ParS compared to TPS with strong WiFi signal. The weighted average of selected bitrates and their frequency from using 1 connection to 10 parallel connections. The numbers are derived from the same experiments that are conducted for Figure 4.6.

Figure 4.8: Time-to-start playback for ParS compared to TPS in strong WiFi signal. The average of 5 runs are represented in the figure. The x-axis shows the number of parallel connections.

start playback as the number of parallel connections increases.

4.1.4 Campus wifi

Besides running the experiments on a small network with weak and strong signal, we ran the experiments on University of Calgary's campus WiFi. The server and the client systems are located on campus but in different buildings and under different sub-networks.

We ran both methods in busy hours of the day when there are lots of connected devices to the network. But because of the high bandwidth availability, both ParS and TPS behave similarly. The selected bitrate is always the maximum possible right after the first video chunk is downloaded. The only difference is in the number of chunks each method downloads till it chooses the maximum bitrate. Because the rate adaptation for ParS happens after downloading every video chunk but for TPS, it depends on the number of connections. Therefore, in general, the number of chunks with maximum bitrate is still higher for ParS because of the frequency of its rate adaptation.

Since, the two methods didn't have a clear advantage over the other in terms of bitrate selection,

we only compare their time to start playback here.

Figure 4.9: Time-to-start playback for ParS compared to TPS in university campus WiFi. The client is connected to the campus network via WiFi and the server is connected to campus network with Ethernet cable. The average of 5 runs are represented in the figure. The x-axis shows the number of parallel connections.

As depicted in Figure 4.9, time-to-start playback for ParS still maintains a low number while it increases for TPS. But because of the high bandwidth on campus' WiFi, the increase of time to start playback for TPS on 10 parallel connections reaches nearly 100 ms which in reality isn't a long time for a user to wait till the video playback starts. But, it is a relatively long time compared to ParS's time to start playback where it is less than 50 ms despite the number of connections.

Reviewing the measurement results in 3 different network environments, we realized the efficiency and advantage of ParS to TPS. We reviewed the bitrate selection and time to start playback for our measurements. Further in this chapter, simulation results are being utilized to evaluate the accuracy of the model introduced in Section 3.2 for finding the number of parallel connections necessary to utilize the available bandwidth.

4.2 Simulation results

To have full control over network parameters such as the transmit queue size of the router, we use ns-3 to simulate the network topology depicted in Fig. 3.1 and study the accuracy of our model presented in Section 3.2. We note that ns-3 is widely used in the literature on TCP and is believed to simulate TCP behavior accurately. With this setup, we can conveniently change the size of the router's transmit queue to arbitrary values to simulate *small queue* and *large queue* scenarios.

4.2.1 Testbed description

The network topology in both small queue and large queue scenarios is the same topology depicted in Fig. 3.1. The access link's bandwidth is set to 10 Mbps and its delay is set to 20 ms. The Internet link's bandwidth is set to 100 Mbps and it has a delay of 5 ms. Thus RTT between the client and server is 50 ms. This setup makes the bottleneck link to be at the edge of the network. It is set in such a way that it recreates the common setup of a wireless network where the access link (the link between user's device and access point) has high latency and a relatively low bandwidth. The transmit queue is set on the access link side of the router where packets from the server are being sent to the client. By default, the packet loss probability in small and large queue scenarios is set to zero and 0.1%, respectively, unless otherwise specified. The TCP segment size in all simulations is set to 1448 bytes.

Each simulation run starts with a given number of parallel TCP connections and lasts for 100 seconds. During the simulation, the server continuously sends data to the client. If the packet loss is set to a number other than zero, then simulations are repeated for 10 times, each time with a different seed. The throughput is then obtained by calculating the average throughput over the 10 simulation runs.

4.3 Small Transmit Queue

We consider both fluid and packet models, as described earlier.

Figure 4.10: Small transmit queue: Comparison of the model and simulation when the packet loss probability due to transmission errors is set to zero. The model provides an upperbound, which becomes more accurate as the queue size increases.

4.3.1 Fluid Approximation

We set the transmit queue size to 1, 5 and 10 packets and compare our fluid model with ns-3 simulations. The results are presented in Fig. 4.10. As explained in the previous section, the fluid model is not an accurate approximation when the queue size is very small. This is clearly visible in the figure for queue size of 1, where the model is quite far from the simulation. However, as soon as we increase the queue size slightly, the model becomes more accurate. In particular, the model always provides an upperbound on the aggregate throughput as expected. Moreover, the model is able to predict the behavior of the aggregate throughput with respect to the number of flows reasonably well especially for large number of flows.

4.3.2 Packet Approximation

Recall that when only one flow passes through the router, the TCP window size is bounded by 2.5 packets. Fig. 4.11 depicts the time series of the TCP window size for one flow when the

Figure 4.11: Small transmit queue: Trace of TCP window size when queue is one packet.

router transmit queue is set to 1 packet. It can be seen from the figure that once the connection stabilizes, the window size is range bound between 1 and 2.5 packets. Using our model, the expected throughput in this case is 0.046 Mbps. From simulations, we obtain the throughput of 0.05 Mbps, which is remarkably close to the value predicted by the model.

When there are multiple flows, as discussed in Section 3.2, the degree of synchronization among the flows is critical in determining the aggregate throughput. In real word scenarios, the synchronization degree is dynamic and varies over time. Since it is difficult to simulate scenarios in which flows are either completely synchronized or desynchronized, we simulate two scenarios where: 1) all connections are started at the same time, or 2) connections are started with a delay from each other in a staggered manner. The results are presented in Fig. 4.12. As expected, as the degree of synchronizing among flows decreases, the aggregate throughput increases. It would be interesting to develop a throughput model for multiple flows that could consider the synchronization degree among the flows as a parameter.

Figure 4.12: Small transmit queue: Aggregate throughput increases as the gap between the start time of connections increases.

4.4 Large Transmit Queue

In this case, we set the size of the transmit queue at the router to 100 packets. Table 4.2 presents the number of flows required to fully utilize the access link as computed using the model presented in Section 3.2. According to the table, for instance, at least 8 flows are required to fully utilize the link when packet loss probability is set to 1%.

Table 4.2: Number of TCP flows from the model.

Loss probability	Number of flows
1%	8
0.1%	2
0.01%	1

Using the same parameters as the ones specified for the model, we have computed the aggregate throughput achieved for different loss probabilities and number of flows. The results are presented in Fig. 4.13. It can be seen that for 1% loss probability, using 8 flows can achieve over 90% link

Figure 4.13: Large transmit queue: Aggregate throughput under different loss probabilities.

utilization. Similarly, for 0.1% and 0.01% loss probabilities, using 2 and 1 flows can achieve high link utilization of over 95%.

Recall that our model also provides an upper bound on the loss probability that allows full link utilization. To verify this, we have conducted a simulation study with one TCP connection. From the model, we obtain that the loss probability should be less than 0.019 to allow full utilization. The simulation results are presented in Fig. 4.14. The figure shows that when the packet loss probability is 0.02, the link utilization reaches over 93%.

4.5 Transmit Queue Occupancy

The transmit queue of the router plays a critical role in determining the utilization of the bottleneck link. A higher occupancy for the queue indicates a higher link utilization in general. To show the effect of multiple flows on the transmit queue occupancy, we run a simulation experiment with transmit queue size set to 100 packets.

Figure 4.14: Large transmit queue: When packet loss is 0.02, the link utilization is over 93%. Beyond this, the link utilization drops.

Figure 4.15: Transmit queue occupancy with different number of flows. The size of the queue is set to 100 packets.

Transmit queue traces over a period of 10 seconds are presented in Fig. 4.15. It can be seen that, while the queue occupancy fluctuates over time, having multiple flows generally results in higher occupancy rates. In Table 4.3, we show the average number of packets in the queue (*i.e.*, the average queue occupancy) for different number of flows along with the achieved aggregate throughput. As expected, increasing the number of parallel flows results in increased average queue occupancy. Consequently, as the queue occupancy increases so does the aggregate throughput.

Table 4.3: Average queue occupancy from simulations.

No. flows	Tput (Mbps)	Avg. packets in queue
1	8.839	22.62
2	9.623	64.49
4	9.785	74.64
8	9.837	76.45

Chapter 5

Conclusion

In this work, we introduced ParS, a video streaming system that uses parallel TCP connections in conjunction with HTTP range requests to speedup downloading video chunks from the server and enhance user's experience in video streaming. We implemented a prototype of the system on a testbed and conducted various measurement experiments to study its performance. We then developed a model to estimate the minimum number of parallel connections required to achieve high utilization of the available bandwidth in the network. Simulation results conducted using ns-3 demonstrated the accuracy and utility of our model. We also measured two quality of experience factors in WiFi testbed and compared our method with traditional streaming methods. Based on our results we could conclude the following:

- Using parallel TCP connections instead of single TCP, increases the total throughput and is used as a common method to overcome TCP's under-utilization problem.
- In video streaming, dedicating the parallel connections to download the same chunk(ParS) works more efficiently than downloading separate chunks(TPS) in terms of *time to start playback*, *bitrate frequency* of delivered video chunks, and *average bitrate* for all the delivered chunks.
- The aforementioned are QoE factors in video streaming therefore we can say that ParS method of video streaming has a higher QoE than TPS.

5.1 Thesis summary

In Chapter 1, we explained the motivation of our research, and the importance of video streaming. Further on, we reviewed a summary of the background information around HTTP video streaming

and explained the problem in current streaming solutions. We demonstrated the scenarios in which TCP is unable to utilize the available bandwidth and reviewed the existing methods to overcome this problem. We then proposed a basic version of our method to address the issue of under-utilization of TCP. Finally, we mentioned our contributions in this research and explained our plan to overcome the under-utilization problem in streaming.

In Chapter 2, we reviewed the necessary background information for video streaming in HTTP and explained the challenges in this area. Then, described the HTTP protocol, its structure, and behavior. After that, we focused on streaming over HTTP. Started from the history of streaming and then, streaming protocols and systems were reviewed. Chapter 2 also elaborated how Dynamic Adaptive Streaming over HTTP (DASH) works. Furthermore, the problems of TCP were discussed, more in depth and details than Chapter 1. Eventually, the existing literature in the area of parallel TCP for video streaming was reviewed and the quality of experience factors were described.

We demonstrated the methodology in Chapter 3. This chapter had two main parts: The mathematical model in which we calculated the number of connections needed to utilize the channel and the solution in which, we represented our parallel streaming system (ParS) in detail and described its differences from other streaming methods. In the model part, we considered small transmit queue and large transmit queue scenarios and presented formulations for both. In the solution part, we described ParS and how it will untangle the problem of TCP that was mentioned earlier in Chapter 2. Further in this chapter (Chapter 3), we pointed out the implementation side of the system, demonstrating both measurement implementations and simulation implementations.

Finally, Chapter 4, exhibits the measurements and simulation results of our system. At first, the testbed in which the measurements took place is described. Then, the results of those measurements are presented, and the comparison of our system with traditional parallel streaming methods are discussed. In measurement results section in this chapter, three scenarios (weak WiFi coverage, strong WiFi coverage, and WiFi coverage on university campus) were considered. Then in the

simulation results section, after describing the testbed, for both cases of small and large transmit queues, the results are presented and their accuracy is discussed and evaluated.

5.2 Future work

There are several areas that this research can be expanded. In the following, we suggest some of those areas that can be achieved in near future:

- **Smoothness and frequency of bitrate change:** The number of times the quality of video fluctuates during a video playback can affect user's experience. Too much change in bitrate selection can have a negative effect on user's opinion about quality of the video playback. Therefore, having less fluctuation on rate adaptation algorithm can be considered as a further step in this work.
- **Improving the rate adaptation algorithm:** In this work we implemented a simple rate adaptation method which chooses closest bitrate to the calculated download rate from downloading previous video chunks. It also, always starts with the lowest bitrate possible which means the first few seconds of video playback will be with low quality. This rate adaptation can be improved with heuristic methods to increase the accuracy of bandwidth estimation and as mentioned earlier, to increase the smoothness of video playback by reducing the fluctuation in bitrate selections.
- **Conducting measurements over 3G and LTE networks:** So far in our work, we tested our model and system on simulations and WiFi testbeds. 3G and LTE networks usually have higher delays and lower bandwidth. Thus, testing our system on these network environments will give insightful feedback on the performance of our streaming system. Also, the number of users using handheld devices to stream video is increasing therefore, proving the efficiency of our system on mobile networks will be necessary in near future.

- **Testing the system on inconsistent connections** As mentioned, the number of streaming users on handheld devices is increasing. Usually, these devices are mobile, and their network coverage can not always be guaranteed. So one of the problems that these type of users may encounter is disconnecting from network multiple times during a video playback. Having a method to address and overcome this issue can be an interesting and useful next step for our work.
- **Conducting user studies to get Mean Opinion Score (MOS):** In Chapter 2 we discussed QoE influence factors. One of those factors that hasn't been covered in this research is MOS. This factor is mostly has been measured and used for mapping Quality of Service factors to Quality of Experience [58]. But considering MOS in our work can also give a better insight of outperforming or underperforming in our system compared to traditional parallel streaming methods.

Appendix A

Appendix

A.1 ParS.java

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.net.HttpURLConnection;
import java.net.InetAddress;
import java.net.Socket;
import java.net.URL;
import java.util.ArrayList;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

public class ParS {
    static int segmentduration = 15;
    long playbacktime = 0;
    static long buffertime = 0;
```

```

static long stallduration = 0;
static long timetostartplayback = 0;
static int numberofstalls = 0;
static String host = "136.159.7.38";
static long totalbitrate = 0;
static String path = "/www-itec.uni-klu.ac.at/ftp/datasets/DASHDataset2014/
    BigBuckBunny/15sec/bunny_";
static String mpdpath = "/www-itec.uni-klu.ac.at/ftp/datasets/
    DASHDataset2014/BigBuckBunny/15sec/BigBuckBunny_15s_simple_2014_05_09.
    mpd";

public static void main(String[] args) throws IOException {
    int connumber = Integer.parseInt(args[1]); // number of parallel
        connections

    System.out.println("number of parallel connections: " + connumber);
    // initializations
    int numsegments = 40; // number of segments Max: 596
    new File("./verynewresults/"+args[0]+"/").mkdirs();
    File log = new File("./verynewresults/"+args[0]+"/"+connumber + "_parallel
        connections_" + "bitrate-frequency.log");
    File log2 = new File("./verynewresults/"+args[0]+"/"+connumber + "
        _parallel connections_" + "results.log");
    PrintWriter writer = new PrintWriter(log, "UTF-8");
    PrintWriter writer2 = new PrintWriter(log2, "UTF-8");
    long bitrate = 0;
    // requesting segments
    for (int i = 1; i <= numsegments; i++) {
        // get time stamp before getting segment
        long transmissionstart = System.currentTimeMillis();
        String filename = "BigBuckBunny_15s" + i + ".m4s";
        String adaptedrate = rateadapt(bitrate) + "";

```

```

int segmentsize = getfilesize(adaptedrate , filename);
int subsegmentsize = segmentsize / connumber;
ArrayList<Thread> ths = new ArrayList<Thread>(connumber);
for (int j = 0; j < connumber; j++) {
    int startrange = j * subsegmentsize;
    int endrange = (j + 1) * subsegmentsize;
    int port = 5200 + j;
    Thread th = new Thread() {
        public void run() {
            try {
                sendGet(startrange , endrange , filename , adaptedrate , port);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    };
    ths.add(th);
    th.start();
}
for(int ii=0 ; ii< ths.size() ; ii++){
    try {
        ths.get(ii).join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
// calculating transmission rate
// get time stamp after getting segment
long transmissionend = System.currentTimeMillis();
long transmissionduration = transmissionend - transmissionstart;
// first segment (playback hasn't started yet)
if(i==1){

```

```

        timetostartplayback = transmissionduration;
        writer2.println("time to start playback(ms): " + timetostartplayback);
        writer.println("downloadRate  adaptedRate");
    }
    else if(buffertime < transmissionduration){
        numberofstalls ++;
        stallduration += transmissionduration - buffertime;
        buffertime = 0;
    }
    buffertime += segmentduration * 1 * 1000;  //*1000 to make in
        milliseconds
    buffertime -= transmissionduration;
    bitrate = segmentsize / transmissionduration;
    bitrate *= 8000; // byte per milisec -> bit per sec
    // outputs
    System.out.println("Segment number: " + i);
    System.out.println("Transmission Rate: " +((float) bitrate /1000000) + "
        Mbps");
    System.out.println("buffer time: " + buffertime);
    // write log file
    writer.println(bitrate + " " + adaptedrate);
    totalbitrate += bitrate;
}
writer2.println("average bitrate: " + (float)totalbitrate / (numsegments *
    1000000) ); //Mbps
writer2.println("Number of stalls: " + numberofstalls);
writer2.print("duration of stalls: " + stallduration);
writer2.close();
writer.close();
}

```

private static int getfilesize(String rate , String filename) **throws**

```

        IOException {
    Socket s;
    try {
        s = new Socket(InetAddress.getByName(host), 80);
        PrintWriter pw = new PrintWriter(s.getOutputStream());
        pw.println("HEAD " + path + rate + "bps/" + filename + " HTTP/1.1");
        pw.println("Host: " + host);
        pw.println("");
        pw.flush();
        BufferedReader br = new BufferedReader(new InputStreamReader(s.
            getInputStream()));
        String t;
        while ((t = br.readLine()) != null) {
            String cl = "Content-Length: ";
            if (t.contains(cl)) {
                String filesize = t.substring(cl.length());
                int filesizebytes = Integer.parseInt(filesize);
                return filesizebytes;
            }
        }
        br.close();
        s.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return -1;
}

private static void sendGet(int startrange, int endrange, String filename,
    String rate, int portnumber)
    throws IOException {
    try {
        HttpURLConnection s;

```

```

    //save file
    URL url = new URL("http://" + host + path + rate + "bps/" + filename);
    s = (HttpURLConnection) url.openConnection();
    //set other headers
    s.setRequestProperty ("Range", "bytes=" + startrange + "-" + endrange);
    //connect
    s.connect();
    BufferedInputStream in = new BufferedInputStream(s.getInputStream());
    OutputStream out = new BufferedOutputStream(new FileOutputStream(new
        File("./files/" + filename)));
    byte[] buf = new byte[256];
    int n = 0;
    while ((n = in.read(buf)) >= 0) {
        out.write(buf, 0, n);
    }
    out.flush();
    out.close();
} catch (IOException e) {
    e.printStackTrace();
}
}

private static int rateadapt(long bitrate) {
    int [] rates = mprates();
    if (bitrate == 0)
        return rates[0];
    else {
        for (int i = 0; i < rates.length; i++) {
            if (bitrate == rates[i])
                return rates[i];
            else if (bitrate < rates[i])
                return i == 0 ? rates[0] : rates[i - 1];
        }
    }
}

```



```

    }
    return rates[rates.length - 1];
}

private static int[] mprates(){
    //downloading the MPD file
    try {
        HttpURLConnection s;
        //save file
        URL url = new URL("http://" + host + mpdpath);
        s = (HttpURLConnection) url.openConnection();
        //connect
        s.connect();
        BufferedInputStream in = new BufferedInputStream(s.getInputStream());
        OutputStream out = new BufferedOutputStream(new FileOutputStream(new
            File("./files/" + "rates.xml")));
        byte[] buf = new byte[256];
        int n = 0;
        while ((n = in.read(buf)) >= 0) {
            out.write(buf, 0, n);
        }
        out.flush();
        out.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
    //parsing the xml file
    try{
        File fXmlFile = new File("./files/rates.xml");
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(fXmlFile);
        doc.getDocumentElement().normalize();
    }
}

```

```

NodeList nList = doc.getElementsByTagName("Representation");
int [] rates = new int [ nList.getLength() ];
for(int i=0 ; i<nList.getLength(); i++){
    Node nNode = nList.item(i);
    if (nNode.getNodeType() == Node.ELEMENT_NODE) {
        Element eElement = (Element) nNode;
        rates[i] = Integer.parseInt(eElement.getAttribute("bandwidth"));
    }
}
return rates;
} catch (Exception e){
    e.printStackTrace();
}
return null;
}
}

```

A.2 simulation.cc

```
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>          // std::setw
#include "ns3/netanim-module.h"
#include "ns3/core-module.h"
#include "ns3/applications-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/flow-monitor-helper.h"
#include "ns3/error-model.h"
#include "ns3/netanim-module.h"
#include "ns3/queue.h"
#include "ns3/rng-seed-manager.h"
#include "ns3/ipv4-flow-classifier.h"
using namespace ns3;
NS_LOG_COMPONENT_DEFINE ("bottleneckrouter");
////////// variables //////////
int simulation_time = 20;
int tcpflows = 1; //number of tcp flows default = 1
int maxBytes = 0; //setting up max number of bytes to send default = 0, 0 is
    unlimited
double lossprob = 0.01; //loss probability default = 0.01 - 1%
int queue_size = 1000; // Queue size at node 1 (router) default 1000 packets
int seed = 1; // Seed number for random number generator default = 1
std::string delay = "5ms"; //default value for delay between middle node and
    receiving node
int tcp_app_data_unit_size = 1460;
```

```

//////////queue tracer function//////////
void queuetracer(NetDeviceContainer device){
    double simtime = Simulator::Now ().GetSeconds ();
    Ptr<NetDevice> toDev = device.Get(0);
    PointerValue txQueue;
    toDev->GetAttribute ("TxQueue", txQueue);
    uint32_t k = txQueue.Get<DropTailQueue> ()->GetNPackets ();
    std::ostringstream q;
    std::ostringstream f;
    std::ostringstream l;
    std::ostringstream s;
    l << lossprob;
    f << tcpflows;
    q << queue_size;
    s << seed;
    std::ofstream file;
    std::string faddress = "../qtraces/q"+q.str()+"-lossp"+l.str()+"-flow"+f.str
        ()+"-seed"+s.str()+".txt";
    std::ostringstream fadd;
    fadd << faddress;
    file.open(faddress.c_str(),std::ios_base::app);
    file<<simtime <<" "<<k<<std::endl;
}

int main(int argc, char *argv[]){
    ////////// logging and command line arguments //////////
    //enable logging module
    LogComponentEnable("bottleneckrouter", LOG_LEVEL_LOGIC);
    //enable commandline argument passing
    CommandLine cmd;
    cmd.AddValue ("tcpflows", "number of tcp flows", tcpflows);
    cmd.AddValue ("lossp", "loss probability between 0.0 and 1.0", lossprob);
    cmd.AddValue ("seed", "seed number for random number generator", seed);

```

```

cmd.AddValue ("delay", "delay between switch and receiver node", delay);
cmd.AddValue ("queue", "queue size of the node in between", queue_size);

cmd.Parse (argc , argv);

////////// global configurations //////////

    //setting up seed number for random number generator
Config::SetGlobal ("RngSeed", IntegerValue(seed));

    //setting up tcp configurations
Config::SetDefault ("ns3::TcpL4Protocol::SocketType", TypeIdValue (
    TcpNewReno::GetTypeId ());
Config::SetDefault ("ns3::TcpSocket::SegmentSize", UIntegerValue (1448));
////////// creating topology //////////

    //creating 2 node containers one for left side one for right side
NodeContainer n0n1;
n0n1.Create(2);
NodeContainer n1n2;
n1n2.Add(n0n1.Get(1));
n1n2.Create(1);

    //setting up error model for simulating loss rate
Ptr<UniformRandomVariable> uv = CreateObject<UniformRandomVariable> ();
uv->SetStream (100);
RateErrorModel error_model;
error_model.SetRandomVariable (uv);
error_model.SetUnit (RateErrorModel::ERROR_UNIT_PACKET);
error_model.SetRate (lossp);    //error rate

    //defining the 2 point to point links between the nodes
PointToPointHelper leftlink;
leftlink.SetDeviceAttribute ("DataRate", StringValue ("100Mbps"));
leftlink.SetChannelAttribute ("Delay", StringValue ("20ms"));
PointToPointHelper rightlink;
rightlink.SetDeviceAttribute ("DataRate", StringValue ("10Mbps"));
rightlink.SetChannelAttribute ("Delay", StringValue ("5ms"));

    //assigning the error model to the link

```

```

rightlink.SetDeviceAttribute ("ReceiveErrorModel", PointerValue(&error_model
));
//installing devices for nodes 0, 1
NetDeviceContainer devices01;
devices01 = leftlink.Install(n0n1);
//installing devices for nodes 1, 2
NetDeviceContainer devices12;
devices12 = rightlink.Install(n1n2);
//setting up queue sizes for nodes
//std::cout<<queue_size<<"\n";
Ptr<DropTailQueue> queue = DynamicCast<DropTailQueue> (DynamicCast<
    PointToPointNetDevice> (devices01.Get (1))->GetQueue ()); //middle node
queue -> SetAttribute("MaxPackets", UIntegerValue(1000));
DynamicCast<PointToPointNetDevice> (devices01.Get (1))->SetQueue (queue);
Ptr<DropTailQueue> endqueue = DynamicCast<DropTailQueue> (DynamicCast<
    PointToPointNetDevice> (devices12.Get (0))->GetQueue ()); //middle right
side node
endqueue -> SetAttribute("MaxPackets", UIntegerValue(queue_size));
DynamicCast<PointToPointNetDevice> (devices12.Get (0))->SetQueue (endqueue);
// installing IP stack
InternetStackHelper stack;
stack.InstallAll();
//defining addresses
Ipv4AddressHelper address;
//defining address for nodes 0, 1
address.SetBase ("10.1.1.0", "255.255.255.0");
Ipv4InterfaceContainer interface01 = address.Assign(devices01);
//defining addresses for nodes 1, 2
address.SetBase ("10.1.2.0", "255.255.255.0");
Ipv4InterfaceContainer interface12 = address.Assign(devices12);
//applying global routing
Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

```

```

////////// creating application //////////
for (int port=10; port < 10+tcpflows; port++){
    //sink application
    PacketSinkHelper sink ("ns3::TcpSocketFactory", InetSocketAddress (
        IPv4Address::GetAny (), port));
    ApplicationContainer sinkApps = sink.Install (n1n2.Get (1));
    sinkApps.Start (Seconds (0.0));
    sinkApps.Stop (Seconds (simulation_time));
    //source(s) application
    ApplicationContainer sourceApps;
    //loop for creating multiple applications (flows) from source
    BulkSendHelper sources("ns3::TcpSocketFactory", InetSocketAddress (
        interface12.GetAddress (1), port));
    //Set the amount of data to send in bytes. Zero is unlimited.
    //sources.SetAttribute ("SendSize", UIntegerValue (tcp_app_data_unit_size)
    );
    sources.SetAttribute ("MaxBytes", UIntegerValue (maxBytes));
    sourceApps = sources.Install (n0n1.Get (0));
    double start = 0.0+(port - 10)/10;
    //double start = 1.0;
    sourceApps.Start (Seconds (start));
    sourceApps.Stop (Seconds (simulation_time));
}

////////// utilities //////////
    //using netAnim
    /*AnimationInterface::SetConstantPosition (n0n1.Get(0), 10.0, 20.0);
    AnimationInterface::SetConstantPosition (n0n1.Get (1), 20.0, 20.0);
    AnimationInterface::SetConstantPosition (n1n2.Get (1), 30.0, 20.0);*/
    //AnimationInterface anim ("myCode1_anim.xml");
    //using flow monitor
    Ptr<FlowMonitor> flowMonitor;
    FlowMonitorHelper flowHelper;

```

```

flowMonitor = flowHelper.InstallAll();

    //using pcap tracing
//rightlink.EnablePcapAll("rightlink");
    //leftlink.EnablePcapAll("leftlink");
std::ostream losspsr;
    std::ostream tcpflowsstr;
std::ostream qsizestr;
    losspsr << lossps;
    tcpflowsstr << tcpflows;
qsizestr << queue_size;
    //run the simulation
Simulator::Stop(Seconds(simulation_time));
for(double i=0 ; i<simulation_time ; i+= 0.1)
    Simulator::Schedule(Seconds(i), &queuetracer , devices12);
Simulator::Run();
//////////making results file//////////
std::ofstream file;
file.open("results-correctTP.txt", std::ios_base::app);
file <<std::left<<std::setw(18)<<queue_size <<","<< std::setw(18)<< lossps <<
    ","<< std::setw(18) << tcpflows <<","<< std::setw(18) << seed <<","<<
    std::setw(18);
    //calculating the throughput
Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier> (
    flowHelper.GetClassifier ());
std::map<FlowId, FlowMonitor::FlowStats> stats = flowMonitor->GetFlowStats
    ();
double total_bytes = 0;
for (std::map<FlowId, FlowMonitor::FlowStats>::const_iterator i = stats.
    begin (); i != stats.end (); ++i){
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow (i->first);
    if ((t.sourceAddress=="10.1.1.1" && t.destinationAddress == "10.1.2.2"
        )){

```



```

        double flow_bytes = i->second.rxBytes;
        total_bytes += flow_bytes;
        std::cout << "   throughput: " << (flow_bytes*8)/(1000000*simulation_time
            ) << "\n";
    }
}

double total_throughput = total_bytes * 8;
total_throughput /= simulation_time;
total_throughput /= 1000;
total_throughput /= 1000;
std::cout << "   Throughput: " << total_throughput << " Mbps\n";
file << total_throughput << ",\n";
file.close();
// ////////////////////////////////// end test code //////////////////////////////////
    //saving flow monitor results before destroying simulator
    flowMonitor->SerializeToXmlFile("bottleneck/small-buffer/flowmonitor-p: "+
        losspsstr.str()+"-n: "+tcpflowsstr.str()+" q: "+qsizestr.str()+".xml",
        true, true);
    Simulator::Destroy();
    return 0;
}

```

Bibliography

- [1] “Cisco visual networking index: Forecast and methodology.” [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.html
- [2] “Adobe dynamic streaming.” [Online]. Available: <http://www.adobe.com/products/hds-dynamic-streaming.html>
- [3] “Apple HLS.” [Online]. Available: <https://developer.apple.com/streaming/>
- [4] “Microsoft smooth streaming.” [Online]. Available: <https://www.microsoft.com/silverlight/smoothstreaming/>
- [5] “Dynamic adaptive streaming over HTTP (DASH) part 1: Media presentation description and segment formats,” online, 2014, iSO/IEC 23009-1:2014.
- [6] G. Tian and Y. Liu, “On adaptive HTTP streaming to mobile devices,” in *IEEE PV*, 2013.
- [7] R. Kuschnig, I. Kofler, and H. Hellwagner, “Improving internet video streaming performance by parallel tcp-based request-response streams,” in *IEEE CNCC*, 2010.
- [8] H. Balakrishnan, S. Seshan, E. Amir, and R. H. Katz, “Improving tcp/ip performance over wireless networks,” in *ACM MobiCom*, 1995.
- [9] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, and S. Tuecke, “Gridftp: Protocol extensions to ftp for the grid,” *Global Grid ForumGFD-RP*, vol. 20, 2003.
- [10] C. Liu, I. Bouazizi, and M. Gabbouj, “Parallel adaptive HTTP media streaming,” in *IEEE ICCCN*, 2011.
- [11] —, “Rate adaptation for adaptive http streaming,” ser. MMSys 2011. ACM, 2011, pp. 169–174.

- [12] T.-Y. Huang, R. Johari, and N. McKeown, “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*. ACM, 2013.
- [13] I. F. Akyildiz, T. Melodia, and K. R. Chowdury, “Wireless multimedia sensor networks: A survey,” *IEEE Wireless Communications*, vol. 14, no. 6, pp. 32–39, 2007.
- [14] S. Smachet, K. Sangkul, and J. Y. Tham, “Enabling parallel streaming of multiple video sections by segment scheduling,” in *ACM MoMM*, 2015.
- [15] “Media streaming server protocols overview.” [Online]. Available: <https://msdn.microsoft.com/en-us/library/hh871697.aspx>
- [16] H. Schulzrinne, A. Rao, and R. Lanphier, “Real time streaming protocol,” Tech. Rep. [Online]. Available: <https://tools.ietf.org/html/rfc2326>
- [17] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “Rtp: A transport protocol for real-time applications,” Tech. Rep. [Online]. Available: <https://tools.ietf.org/html/rfc3550>
- [18] “Real-time messaging protocol (rtmp).” [Online]. Available: <https://www.adobe.com/devnet/rtmp.html>
- [19] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol – http/1.1,” Tech. Rep. [Online]. Available: <https://tools.ietf.org/html/rfc2616>
- [20] —, “Hypertext transfer protocol – http/1.0,” Tech. Rep. [Online]. Available: <https://tools.ietf.org/html/rfc1945>
- [21] “A history of media streaming and the future of connected tv.” [Online]. Available: <http://www.theguardian.com/media-network/media-network-blog/2013/mar/01/history-streaming-future-connected-tv>

- [22] “About 3gpp.” [Online]. Available: <http://www.3gpp.org/about-3gpp>
- [23] E. R. Pantos and W. May, “Internet draft - http live streaming,” Tech. Rep. [Online]. Available: <http://tools.ietf.org/html/draft-pantos-http-live-streaming-01>
- [24] A. Fecheyr-Lippens, “A review of http live streaming,” Tech. Rep. [Online]. Available: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.466.6043&rep=rep1&type=pdf>
- [25] T. Stockhammer, “Dynamic adaptive streaming over http –: Standards and design principles,” in *Proceedings of the Second Annual ACM Conference on Multimedia Systems*, ser. MMSys ’11. ACM, 2011.
- [26] T. C. Thang, Q. D. Ho, J. W. Kang, and A. T. Pham, “Adaptive streaming of audiovisual content using mpeg dash,” *IEEE Transactions on Consumer Electronics*, vol. 58, no. 1, pp. 78–85, 2012.
- [27] M. Claeys, S. Latré, J. Famaey, T. Wu, W. Van Leekwijck, and F. De Turck, “Design and optimisation of a (fa) Q-learning-based HTTP adaptive streaming client,” *Connection Science*, vol. 26, no. 1, pp. 25–43, 2014.
- [28] L. S. Brakmo and L. L. Peterson, “Tcp vegas: end to end congestion avoidance on a global internet,” *IEEE Journal on Selected Areas in Communications*, vol. 13, no. 8, pp. 1465–1480, 1995.
- [29] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, “Tcp westwood: Bandwidth estimation for enhanced transport over wireless links,” in *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom ’01. ACM, 2001.
- [30] A. G. T. Henderson, S. Floyd, “The newreno modification to tcp’s fast recovery algorithm,” Tech. Rep. [Online]. Available: <https://tools.ietf.org/html/rfc6582>

- [31] L. A. Grieco and S. Mascolo, "Performance evaluation and comparison of westwood+, new reno, and vegas tcp congestion control," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 25–38, Apr. 2004.
- [32] Y. Tian, K. Xu, and N. Ansari, "Tcp in wireless environments: problems and solutions," *IEEE Communications Magazine*, vol. 43, no. 3, pp. S27–S32, 2005.
- [33] J. Lee, D. Gunter, B. Tierney, B. Allcock, J. Bester, J. Bresnahan, and S. Tuecke, "Applied techniques for high bandwidth data transfers across wide area networks," *Lawrence Berkeley National Laboratory*, 2001.
- [34] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.
- [35] "Internet download manager." [Online]. Available: <http://www.internetdownloadmanager.com>
- [36] "Download accelerator plus." [Online]. Available: <http://www.speedbit.com>
- [37] S. Lederer, C. Mueller, B. Rainer, C. Timmerer, and H. Hellwagner, "Adaptive streaming over content centric networks in mobile networks using multiple links," in *IEEE ICC*, 2013.
- [38] A. K. Chaurasia and A. K. Jagannatham, "dynamic parallel tcp for scalable video streaming over mimo wireless networks," in *IFIP WMNC*, 2013.
- [39] T. Hofeld, M. Seufert, M. Hirth, T. Zinner, P. Tran-Gia, and R. Schatz, "Quantification of youtube qoe via crowdsourcing," in *Multimedia (ISM), 2011 IEEE International Symposium on*, 2011.
- [40] R. K. Mok, E. W. Chan, X. Luo, and R. K. Chang, "Inferring the qoe of http video streaming from user-viewing activities," in *Proceedings of the First ACM SIGCOMM Workshop on Measurements Up the Stack*, ser. W-MUST '11. ACM, 2011.

- [41] J. Yao, S. S. Kanhere, I. Hossain, and M. Hassan, "Empirical evaluation of http adaptive streaming under vehicular mobility," in *International Conference on Research in Networking*. Springer, 2011, pp. 92–105.
- [42] T. Zinner, T. Hossfeld, T. N. Minhas, and M. Fiedler, "Controlled vs. uncontrolled degradations of qoe: The provisioning-delivery hysteresis in case of video," in *EuroITV 2010 Workshop: Quality of Experience for Multimedia Content Sharing*, 2010.
- [43] M. Seufert *et al.*, "A survey on quality of experience of HTTP adaptive streaming," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 1, 2015.
- [44] A. Raake and S. Egger, *Quality and Quality of Experience*. Cham: Springer International Publishing, 2014, pp. 11–33.
- [45] T. D. Pessemier, K. D. Moor, W. Joseph, L. D. Marez, and L. Martens, "Quantifying the influence of rebuffering interruptions on the user's quality of experience during mobile video watching," *IEEE Transactions on Broadcasting*, vol. 59, no. 1, pp. 47–61, 2013.
- [46] Y. Qi and M. Dai, "The effect of frame freezing and frame skipping on video quality," in *2006 International Conference on Intelligent Information Hiding and Multimedia*, 2006.
- [47] Q. Huynh-Thu and M. Ghanbari, "Temporal aspect of perceived quality in mobile video broadcasting," *IEEE Transactions on Broadcasting*, vol. 54, no. 3, pp. 641–651, 2008.
- [48] T. Hacker, B. Athey, and B. Noble, "The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network," in *IEEE IPDPS*, 2002.
- [49] E. Altman *et al.*, "Parallel TCP sockets: Simple model, throughput and validation," in *IEEE Infocom*, 2006.
- [50] F. Baccelli and K. B. Kim, "TCP throughput analysis under transmission error and congestion losses," in *IEEE Infocom*, 2004.

- [51] F. Baccelli, D. R. McDonald, and J. Reynier, “A mean-field model for multiple TCP connections through a buffer implementing RED,” *Perf. Eval.*, vol. 49, no. 1, 2002.
- [52] J. F. Kurose, *Computer Networking: A Top-Down Approach Featuring the Internet, 3/E*. Pearson Education India, 2005.
- [53] J. Gettys and K. Nichols, “Bufferbloat: Dark buffers in the internet,” *IEEE Internet Comput.*, vol. 15, no. 3, 2011.
- [54] “Computing TCP’s retransmission timer.” [Online]. Available: <http://tools.ietf.org/html/rfc6298>
- [55] “ns3 network simulator.” [Online]. Available: <https://www.nsnam.org>
- [56] “Big buck bunny.” [Online]. Available: <http://www.bigbuckbunny.org>
- [57] “big buck bunny dash dataset.” [Online]. Available: <http://www-itec.uni-klu.ac.at/ftp/datasets/DASHDataset2014/BigBuckBunny/15sec/>
- [58] M. Fiedler, T. Hossfeld, and P. Tran-Gia, “A generic quantitative relationship between quality of experience and quality of service,” *IEEE Network*, vol. 24, no. 2, pp. 36–41, 2010.