ETH ZURICH - SYSTEMS GROUP

Video Streaming with Combined Congestion Control and ABR Algorithms

Gamze Islamoglu

July-August 2019

CONTENTS

1. INTRODUCTION

This project is a two-month research project under the scope of ETH Computer Science Summer Student Research Fellowship. The main objective is to examine the effect of combining two different control loops involved in video streaming: one is Adaptive Bitrate (ABR) algorithm in application layer and other is congestion control in transport layer.

1.1. VIDEO STREAMING

Videos are one of the most common contents of the Internet, nowadays. In popular video providers like YouTube or Netflix, although users are allowed to choose between video qualities, the automatic quality selection is performed to provide the best quality of experience (QoE) to users. QoE depends on rebuffering rate, average video quality, startup delay and how often the quality rate changes [1].

1.1.1. ADAPTIVE BITRATE STREAMING (ABR)

In the server side, there are video files of various qualities for each video. The client sends a request to download a video chunk of a certain quality. The quality of each chunk is totally determined by the client and the algorithm that makes the decision is called *Adaptive Bitrate Streaming (ABR)*.

The video formats on the server side can be HLS (HTTP Live Streaming) supported by Apple, HDS (HTTP Dynamic Streaming) supported by Adobe, MSS (Microsoft Smooth Streaming) supported by Windows and MPEG-DASH developed by MPEG Standardization Group. HLS and DASH are dominant in the sector today [2]. In this project, Dynamic Adaptive Streaming over HTTP (DASH) format is preferred.

In ABR algorithms, two main parameters are taken into consideration to automatically decide on video quality. These parameters are buffer occupancy and capacity estimation. *Buffer-Based* approaches choose the quality of the next chunk according to the current buffer occupancy. *Rate-Based* approaches selects the upcoming chunk quality according to estimated network throughput which is calculated by looking at last segments' arrival time.

1.1.2. CONGESTION CONTROL

Transport layer provides end-to-end communication between processes running on different hosts and some of the responsibilities of the transport layer are multiplexing/demultiplexing, error detection, reliable data transfer (only for reliable protocols) and congestion control. Congestion control is responsible for adjusting the transmission rate in order to avoid or recover from congestion in the network [3].

Most importantly, congestion control runs independently of ABR. This means that there are two separate control loops to stream a video. The question is *"Can a combined single algorithm increase the Quality of Experience (QoE) for video streaming?"*

2. BACKGROUND

In this section, some background information about video streaming, Adaptive Bitrate (ABR) and congestion control is provided.

2.1. VIDEO STREAMING

As mentioned in the Introduction, *Dynamic Adaptive Streaming over HTTP (DASH),* one of the most common video formats, is used in this project. In DASH, each video is divided into chunks of few seconds and these chunks are encoded at different discrete bitrates [4]. ABR algorithm on the client side decides on the next chunk quality and requests the chunk at that quality from the server. In the server side, videos are stored in chunks of various qualities. Also, a manifest file called Media Presentation Description (MPD) includes the information about the video. This file is requested from the server at the beginning of the video.

2.1.1. QUALITY OF EXPERIENCE (QOE)

Quality of Experience (QoE) is a metric to express the level of pleasure of watching a video in terms of various factors such as video quality, rebuffer time, startup time and quality changes.

Video Quality: Higher video quality means higher resolution or higher bitrates for each chunk. All video viewers prefer watching high quality videos. However, this requires downloading bigger and bigger chunks which takes longer time.

Rebuffer Time: Downloaded video bits are stored in a buffer on the client side temporarily and played as the time comes. When the buffer empties, video is paused until the next chunk arrives. Rebuffering is undesired and reduces the QoE.

Startup Time: Startup time is the time that passes between the play request of the user and the actual starting time of the video. The reason for the startup time is that the buffer is initially empty and at least the first chunk should be downloaded to play. Users desire an instant play and hence, a smaller startup time increases the QoE.

Quality Changes: Variations of quality between chunks frequently is not desired by users. Therefore, quality changes should be minimized while maximizing video quality.

It is possible to define different QoE metrics by combining these parameters in various ways. The metric used in this project is as follows [4]:

$$QoE = \sum_{k=1}^{N} q(R_k) - \delta T_{start} - \lambda \sum_{k=1}^{N} T_{buf,k} - \mu \sum_{k=1}^{N-1} |q(R_k) - q(R_{k+1})| \tag{2.1}$$

N is the total number of chunks. The function $q(R)$ maps bitrates to quality values. The first term represents video quality, the second term represents startup time, the third one represents rebuffer time and the last one represents quality changes. The parameters δ, λ and μ are used to adjust the relative effects of each term.

2.2. Adaptive Bitrate Streaming Algorithms

Adaptive Bitrate Streaming (ABR) algorithms are responsible for choosing the right video quality to play on the client side. Based on various parameters, ABR decides on a next chunk quality and requests it from the server. According to how this decision is made, there are many ABR algorithms and they affect the QoE substantially.

2.2.1. Buffer Based (BB)

In buffer based approach, the buffer occupancy on the client side is used to determine the next chunk quality. If the occupancy is low, a lower quality chunk is requested and if the buffer is high, a high quality chunk is requested.

2.2.2. Rate Based (RB)

Rate based methods try to estimate the available bandwidth by measuring the download time of each chunk. Depending on the download rate of last few chunks, a new chunk is requested in a certain quality.

2.2.3. Festive

Festive is an algorithm that aims to maximize fairness, stability, efficiency and adaptiveness at the same time. Firstly, Festive uses harmonic mean over 20 samples to estimate bandwidth. Secondly, Festive makes use of the current bitrate to choose the next quality. If a player has a high current bitrate level, it tends to estimate the bandwidth higher than the players with lower quality. This obstructs fair allocation of the bandwidth. Therefore, Festive lowers the rate of increase possibility and enhances the rate of decrease possibility with increasing current bitrate. Festive also incorporates delayed update method to increase stability. Lastly, the randomized scheduler downloads the next chunk immediately if buffer occupancy is less than the target. Otherwise, it adds a random delay to prevent start time biases [5].

2.2.4. Model Predictive Control (MPC)

Model predictive control is actually an advanced process control method and adapted to video streaming. MPC tries to solve an exact optimization problem, in our case, it tries to maximize QoE. By combining rate-based and buffer-based data from the last few chunks, MPC predicts the throughput for the next few chunks and makes an optimal bitrate decision [4].

There are two types of implementation of MPC in this project: robust MPC and fast MPC. Robust MPC considers the maximum prediction error of previous optimizations in addition to MPC. On the other hand, fast MPC utilizes a lookup table instead of solving the optimization problem [6].

2.2.5. PENSIEVE

Pensieve is a learning-based approach. With the help of reinforcement learning, ABR agent learns an ABR policy by observing throughput estimation, buffer occupancy and past bitrate decisions. Additionally, the QoE due to the decision of the agent is passed back as a reward which improves the model [7].

2.2.6. BOLA

BOLA formulates bitrate adaptation as a utility maximization problem and uses Lyapunov optimization techniques based on buffer occupancy to minimize rebuffering and maximize video quality [8].

2.3. TRANSPORT LAYER PROTOCOLS AND CONGESTION CONTROL

Transport layer sits just above the application layer and provides end-to-end communication between processes running on different hosts. There are two types of protocols: reliable (like TCP) and unreliable (like UDP).

2.3.1. TRANSMISSION CONTROL PROTOCOL (TCP)

TCP is the most widely used transport protocol and it provides reliable connection-oriented service. TCP makes sure that data is delivered from sending process to receiving process by using flow control, sequence numbers, acknowledgements and timers [3]. Moreover, TCP provides congestion control which utilizes Additive Increase - Multiplicative Decrease (AIMD) approach for a fair connection and there are various versions of congestion control for TCP such as TCP Tahoe, Reno, Vegas, CUBIC and BBR.

2.3.2. USER DATAGRAM PROTOCOL (UDP)

UDP is a connectionless, unreliable protocol and does not utilize any congestion control. However, there are other protocols built on top of UDP that provides reliability and congestion control. One such example is QUIC generated by Google [9]. Because TCP is the most common protocol and hard to modify, Google preferred to produce its protocol over UDP.

In this project, we also preferred to use UDP since the modification of congestion control was required.

3. QUIC IMPLEMENTATION

In the first part of the project, the idea was using QUIC to modify congestion control and then, merging the congestion and Adaptive Bitrate Streaming (ABR) control loops. After spending three weeks to setup QUIC, we decided that the setup time was too long compared to our time limitation. However, the progress made is explained in this section.

3.1. WHAT IS QUIC?

QUIC is an alternative protocol to TCP which is the most common transport protocol. The early version was proposed by Google in 2013. QUIC is implemented on top of the UDP protocol, which is an unreliable transport protocol. The main motivation was to eliminate some problems in TCP such as long handshake processes and latency. In Fig. 3.1, it can be seen that QUIC changes HTTP/2, TLS and TCP parts in the general HTTPS stack [9].

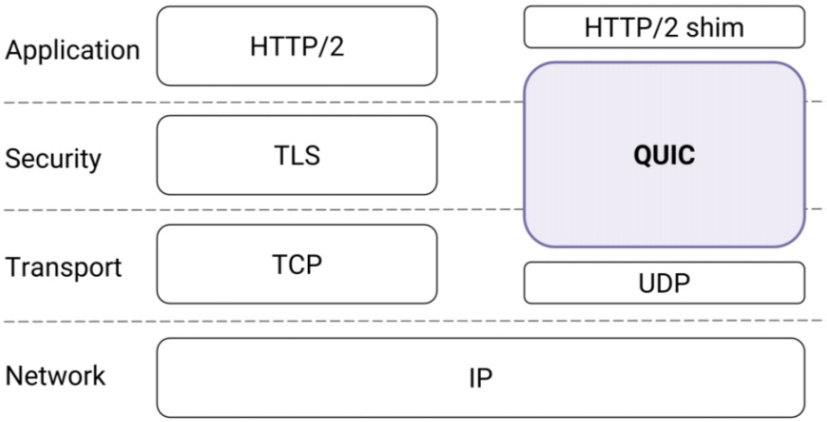

Figure 3.1: QUIC in the traditional HTTPS stack. [9]

After Google's launch of QUIC, an IETF working group is formed to standardize it in 2016. IETF has made some adjustments and continues its work. Currently, Google and IEFT versions of QUIC have some discrepancies.

3.2. WHY QUIC?

Since our aim was to combine congestion control and ABR, we needed the flexibility to change transport layer and congestion control parameters. However, TCP does not provide this flexibility with its isolated layers and settled usage. On the other hand, QUIC can easily be modified thanks to its design.

QUIC as an emerging transport protocol utilizes various congestion algorithms. By default, CUBIC is used with some additions. These additions provide some advantages [10]:

- Different levels of congestion control algorithms can be implemented at the application level without the need for operating system or kernel support.
- Different connections for a single application can also support configuring different congestion controls.

This features made QUIC a plausible option and thus, I started with the implementation of QUIC.

3.3. Choosing the right QUIC implementation

There are many different QUIC implementations available. [11] lists some of them. The most outstanding and complete one is Google Chromium implementation which was chosen to be implemented in the first weeks. The details of the implementation of Google QUIC are explained in Section 3.4.

Even though Chromium version was downloaded and used successfully, making modifications was hard for few reasons: the incompatibility with previous versions and already written video streaming codes, complex structure and time constraints. Therefore, other implementations can be more beneficial for further studies. For instance, there is a GO implementation of QUIC called *"quic-go"* [12] which seems more user-friendly and there is a Bachelor project [10] as an example.

3.4. QUIC Installation from Google

The main document followed is *Playing with QUIC* [13]. This requires us to first download *Chromium*. Note that a Linux-Ubuntu 16.04 operating system is used.

3.4.1. Chromium Installation

The main instructions can be found in [14]. Also, [15] is helpful for some possible problems. The steps are as follows:

1. `sudo apt install git`
2. `git clone` https://chromium.googlesource.com/chromium/tools/depot_tools.git
3. `cd depot_tools`
4. `export PATH = $PATH:/home/gamzeisl/depot_tools`
5. `mkdir  /chromium`
6. `cd  /chromium`
7. `fetch -nohooks chromium` (at least 30 min)
8. `cd src`
9. `./build/install-build-deps.sh`
10. `gclient runhooks`

11. `gn gen out/Default` (Folder name "Default" can be changed and should be changed for separate builds.)

    If this command fails, check [15] and follow these commands:

    - `sudo apt install libpango1.0-dev`
    - `sudo apt install libcogl-pango-dev`
    - `sudo apt install libjpeg-dev`

      and run again:

    - `gn gen out/Default`

    From [15]:

12. `sudo apt install flex`
13. `sudo apt install bison`
14. `sudo apt-get update`
15. `sudo apt-get install autoconf`
16. `sudo apt-get install libjson-perl`
17. `sudo apt-get install emboss bioperl ncbi-blast+ gzip libjson-perl libtext-csv-perl libfile-slurp-perl liblwp-protocol-https-perl libwww-perl`
18. `sudo apt install json`
19. `sudo apt install gperf`

    Install Java, if there is not already:

20. `sudo apt update`
21. `sudo apt install default-jdk`

    In src:

22. `autoninja -C out/Default chrome` (few hours)

3.4.2. Playing with QUIC

After the Chromium installation, we can go back to *Playing with QUIC* [13]. The following instructions should be followed:

1. `export PATH = $PATH:/home/gamzeisl/depot_tools`
2. `ninja -C out/Debug quic_server quic_client`

   Generate server data folder:

3. `mkdir /tmp/quic-data`
4. `cd /tmp/quic-data`
5. `wget -p -save-headers https://www.example.org`

Generate certificates, QUIC only supports HTTPS:

6. `cd net/tools/quic/certs`
7. `./generate-certs.sh`
8. `cd -`
9. `certutil -d sql:$HOME/.pki/nssdb -A -t "C,," -n quic -i net/tools/quic/certs/out/2048-sha256-root.pem` ([16], [17])

Preparation of server files

For each file in the server folder (*/tmp/quic-data*), a header should be specified:
For example, for index.html header should be:

HTTP/1.1 200 OK
X-Original-Url: https://www.example.org/index.html

There are few ways of adding these headers. However, one should be careful especially for different types of files like m4s. Although m4s files can be opened in a text editor, it corrupts the file. One option is to use *emacs*.

Install emacs:

- `sudo apt install emacs25`

Open the files with emacs editor and add the header:

- `emacs 1.m4s`

The other option is to use *cat* command. A Python script is provided in Appendix A. This script takes all the files in a folder and adds the header in the specified format with the file name. All files in the server side should have these headers, otherwise, server fails.

Server

To run server through the terminal, in *src*:

- `sudo ./out/Default/quic_server --v=1 -quic_response_cache_dir=/tmp/quic-data/www.example.org -certificate_file=net/tools/quic/certs/out/leaf_cert.pem -key_file=net/tools/quic/certs/out/leaf_cert.pkcs8`
  --v=1: opens verbose for debugging.
  --log-net-log=C:\some_path\some_file_name.json parameter can be added for log files.
  To visualize these log files, [18] can be used.

Client

After running the server, client can be run from another terminal in *src*:

To run the client through Chromium:

- `out/Default/chrome --v=1 --user-data-dir=/tmp/chrome-profile --no-proxy-server --enable-quic --origin-to-force-quic-on=www.example.org:443`

```
--host-resolver-rules='MAP www.example.org:443 127.0.0.1:6121'
https://www.example.org/
```
Check [19]

Or following command can be used (does not open browser):

- `./out/Default/quic_client --host=127.0.0.1 --port=6121 https://www.example.org/ --v=1`

To get rid of cache, add --disable-application-cache to both.
To run the client in Chrome, use:

- `google-chrome --v=1 --user-data-dir=/tmp/chrome-profile --no-proxy-server --enable-quic --disable-application-cache --origin-to-force-quic-on=www.example.org:443 --host-resolver-rules='MAP www.example.org:443 127.0.0.1:6121' https://www.example.org/index.html`

Opening Pages through Chrome

After running the server as usual, Chrome should be opened from the terminal:

- `google-chrome --v=1 --user-data-dir=/tmp/chrome-profile --no-proxy-server --enable-quic --disable-application-cache --origin-to-force-quic-on=www.example.org:443 --host-resolver-rules='MAP www.example.org:443 127.0.0.1:6121'`

Then, you can open your pages from address line with url such as https://www.example.org/index.html

3.4.3. Video Streaming

Basic Video Streaming

To play a video in the browser using QUIC server, a basic dash player is used initially. To run dash player, *dash.all.min.js* file is needed which can be downloaded from [20]. The index file used is given in Appendix B.

Note that videos do not autoplay with sound, so I used muted autoplay. Chromium does not support our video codec, therefore I used Chrome instead.

AStream Player

Because the aim of the project was to modify both ABR and congestion algorithms, we needed another solution. The main problem was QUIC client and server cannot be called and ruled simply from Python and we could not modify old codes to work with QUIC server and client.

Therefore, my second attempt was trying to use *QUIC-Streaming* [21] project. This project uses *AStream* video player [22] with QUIC.

This player has a specific format for *mpd* files which can be found in [23]. Also, the related video files are provided in [24].

I managed to run these videos with built QUIC server and client after adding the files to */tmp/quic-data*. However, to run all the code in [21], server and client had to be modified.

Although modified QUIC files are provided in [21], I could not use them because they were for an earlier version of QUIC.

In short, after three weeks of trials, we decided to leave this QUIC implementation to a more long-termed project rather than a two-month project.

4. IMPLEMENTATION WITH UDP

Due to the problems encountered in QUIC implementation, we decided to use a different setup already implemented using UDP. The structure of the code that I started with is given in Fig. 4.1. In this version, there is a fixed bandwidth known by the controller and controller distributes the bandwidth according to the number of clients, that is, video players. The video server sends the packets to the proxy server with UDP. Then, packets are sent to the ABR server with TCP. The reason for using a proxy server is that the browser that plays the video cannot communicate with UDP. The controller also informs the ABR server about the suggested bitrate and ABR algorithm takes this value into account to determine video quality.

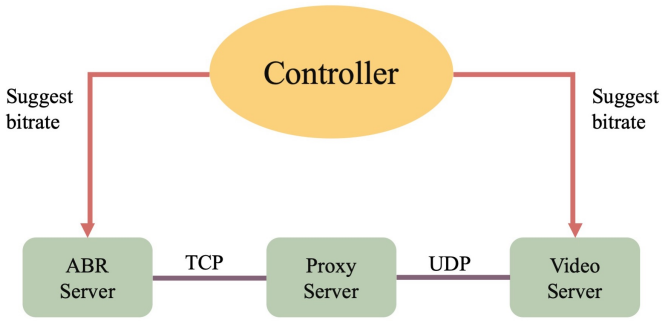

Figure 4.1: Initial Structure.

4.1. SETTING UP THE INITIAL INFRASTRUCTURE

In this section, some useful resources are provided to ease the construction of the code.

To list local servers and their status, npm's *webdriver-manager* can be used [25].

The program uses localhost to serve the videos, therefore video files should be stored in */var/www/html*. However, this folder cannot be modified without permission. To copy and paste and modify the files in this folder easily, the following command can be used: "`sudo chmod -R 777 /var/www`" [26]. After getting permission with this command, you can change the folder as the usual folders.

Lastly, the videos are played in Chrome browser and *selenium* module is used to automatically open the browser. *webdriver* from *selenium* calls Chrome, but it needs a *chromedriver* which can be downloaded from [27] according to the owned Chrome version. The location of this driver should be provided to the *webdriver* which is called in *run_browser_test.py* in the following manner:

```
driver_path = "/.../chromedriver"
browser = webdriver.Chrome(executable_path=driver_path)
```

4.2. Modified Infrastructure

As stated in the Introduction, the aim of this project was to combine congestion control and ABR. Therefore, the initial infrastructure needed some modifications. First of all, the UDP connection was not reliable and there were not any acknowledgement (ACK) signals. ACKs are added to make the protocol reliable. The controller is totally eliminated and full knowledge of bandwidth is not available anymore. Also, a connection between the ABR server and video server is utilized to enable communication and combination of the control loops. The overview of the structure is shown in Fig. 4.2. The details of each block are presented in the following sections.

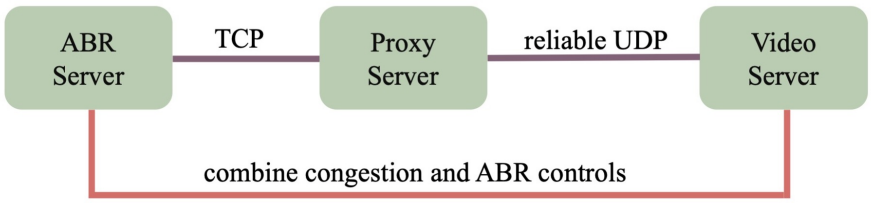

Figure 4.2: Modified Structure.

4.2.1. Video Server

Video server is the real server in terms of a classical client/server model, whereas ABR server is the video player, namely the client. The main task of the video server is to send video chunks over UDP to the proxy server. Also, it has a TCP connection with the ABR server through which it takes the current buffer occupancy value from the client. Then, the video server decides what will be the next chunk quality and tells the next chunk quality to the ABR server over the same TCP connection. The details about the selection of the next chunk quality are provided in Section 5.

4.2.2. Proxy Server

Proxy server is only responsible for directing video chunks coming from the video server to the ABR server. It also sends acknowledgements to the video server. The proxy server is implemented because the video player, namely the browser is not capable of handling UDP packets. UDP packets are converted into TCP packets in the proxy server and send through the TCP connection between proxy and ABR server. Note that the UDP connection is throttled to a certain bandwidth during the experiments and TCP connection is not and send packets almost immediately.

4.2.3. ABR Server

As mentioned, ABR server is the client side where the video is being played. In the classical case, client chooses the next chunk quality according to the ABR algorithm used. In our case, selection of next chunk quality is transferred to the video server and ABR server only takes the determined next chunk quality and requests the corresponding video chunk from the video server.

In our configuration, certain ports are allocated for each job. For each additional stream, one more set of server scripts are run and one more set of ports are allocated. The port numbers are adjusted as `(port_offset + stream_id)`. `stream_id` starts from zero and increased by 1 for each additional stream. Port offsets are shown in Fig. 4.3.

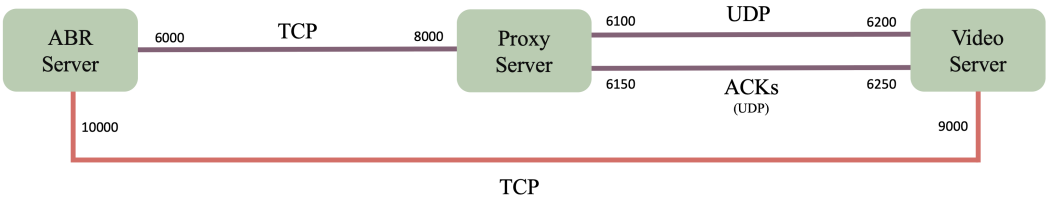

Figure 4.3: Structure with Port Numbers.

5. IMPLEMENTATION OF CONGESTION CONTROL AND ABR ALGORITHMS

As discussed in the Introduction, we needed to modify and combine both congestion control and ABR algorithms. Thus, we first added the TCP connection between ABR and video servers, this enabled us to make the quality decision on the server side and convey it to the player. After this, we implemented a basic congestion algorithm in UDP and designed a server side ABR algorithm.

5.1. CONGESTION CONTROL ALGORITHM

Because UDP is an unreliable and connectionless protocol, it does not utilize acknowledgement (ACK) signals. So, I first added ACKs sent from proxy server to the video server for each packet. An ACK includes the acknowledged video chunk and sequence number. These ACKs are listened and checked after sending several packets equal to *congestion window (cwnd)*.

The congestion control is similar to TCP Reno. It waits for ACKs after bursting *cwnd* number of packets. If they arrive before *timeout*, the *congestion window (cwnd)* is increased either by one for congestion avoidance or doubled for slow start. If *timeout* occurs, *congestion window (cwnd)* and *slow start threshold (ssthreshold)* are halved. The units of *cwnd* and *ssthreshold* can be thought as packets per interval because next packets are sent immediately after taking all expected ACKs. The default starting value of *congestion window (cwnd)* is chosen 1 and starting *slow start threshold (ssthreshold)* is chosen 64.

To calculate timeout, deviation of RTT is calculated with Eq. 5.1 and next RTT is estimated with Eq. 5.2 [3].

$$dev\_RTT = 0.75 * dev\_RTT + 0.25 * |estimated\_RTT - sample\_RTT| \tag{5.1}$$

$$estimated\_RTT = 0.875 * estimated\_RTT + 0.125 * sample\_RTT \tag{5.2}$$

Then, timeout is calculated with Eq. 5.3 [3]. Starting value of timeout is chosen as 1 second.

$$timeout = estimated\_RTT + 4 * dev\_RTT \tag{5.3}$$

The pseudo-code of this simple congestion control algorithm is as follows:

```
if CWND ACKs come before timeout:
    if CWND < ssthreshold:
        CWND *= 2       - > Slow Start
    else:
        CWND += 1       - > Congestion Avoidance
    dev_RTT = 0.75 * dev_RTT + 0.25 * abs(estimated_RTT - sample_RTT)
    estimated_RTT = 0.875 * estimated_RTT + 0.125 * sample_RTT
    timeout = estimated_RTT + 4 * dev_RTT
    bitrates[i] (in bps) = CWND / sample_RTT * PACKET_SIZE (in bits)
```

```
        i += 1
    if timeout occurs:
        timeout *= 2
        ssthreshold = CWND/2
        CWND = ssthreshold
        retransmit packets after last ACKed packet
```

We also estimate the bandwidth (called `bitrates`) in congestion control algorithm and use it to determine next chunk quality together with buffer occupancy on the client side.

5.2. ABR ALGORITHMS

While starting this project, the main idea was that while bandwidth is continuous, the video qualities are discrete. Therefore, we can limit the transmission rate of video packets in a discrete quality level so that we do not waste the excess bandwidth. Although this idea is appealing, it discards the advantage of video player buffer. We can use the excess bandwidth to fill the buffer and pass to higher qualities without rebuffering even though the bandwidth is not enough. Also, we assumed that the bandwidth is fixed in a bottleneck network in this project, but available bandwidth can change due to joining or leaving streams. The new joining players will reduce the bandwidth and can cause rebuffering. In short, using all available bandwidth is better to maximize the quality of experience (QoE).

However, limiting the rate can be useful for other purposes. For example, if there are other types of traffic like random TCP traffic in the background, the excess bandwidth can handle this traffic without disturbing video players. In short, choosing the suitable ABR algorithm depends on the objective of the designer. In designing ABR algorithm `GAMZE`, I tried to maximize the quality of experience (QoE); so, all available bandwidth is used. In the second part, I also designed `LIMIT` ABR algorithm which limits the transmission rate according to video quality level.

5.2.1. GAMZE

This algorithm determines the next chunk quality according to rates estimated by the congestion control algorithm and buffer occupancy. This method uses the available bandwidth estimation to foresee buffer occupancy and makes the decision based on this predicted buffer occupancy. It should be noted that all available bandwidth is used, namely, packets are sent consecutively without any breaks. The pseudo-code is as follows:

```
BITRATES = [300, 750, 1200, 1850, 2850, 4300]
i = length(bitrates)
average_rate = mean(bitrates[i-20:i-1])

# remove instant peaks
sum_bitrates = sum(bitrates[i-20:i-1])
```

```
m = 19
for i in range(i-20,i-1):
    if bitrates[i] > 1.5 * average_rate:
        sum_bitrates -= bitrates[i]
        m -= 1
determined_quality = sum_bitrates / m

# find out matching quality level
tmp_bitrate = determined_quality
tmp_quality = 0
i = 5
while i >= 0:
    if tmp_bitrate >= BITRATES[i]:
        tmp_quality = i
        break
    tmp_quality = i
    i -= 1

# check whether buffer allows an upgrade
if i < 5:
    if BufferOccupancy > BUFFER_UPGRADE_LIMIT:
        if BufferOccupancy + 4 * determined_quality / BITRATES[tmp_quality+1] > 10:
            tmp_quality += 1
    elif last_quality > tmp_quality:
        if BufferOccupancy + 4 * determined_quality / BITRATES[tmp_quality+1] > 10:
            tmp_quality += 1

# check whether buffer requires a downgrade
if i > 0:
    if BufferOccupancy + 4 * determined_quality / BITRATES[tmp_quality] < 10:
        tmp_quality -= 1

next_quality = tmp_quality
```

First, the average rate of the last 20 congestion window is found by removing some peaks that deviates from the average a lot and corresponding quality level out of six levels is found. After finding the network's bandwidth and the closest lower quality level, we check the situation of the player buffer. If the buffer has a certain amount of backup (more than *BUFFER_UPGRADE_LIMIT*) and is expected to have enough backup after 4 seconds with increased quality, we upgrade the quality. Also, if tmp_quality is lower than the previous chunk, the expected buffer situation is checked and quality is upgraded. This prevents rebuffering if the quality was upgraded in the previous chunk due to high buffer occupancy. Finally, we check the expected buffer occupancy after incoming 4 seconds and downgrade if necessary.

5.2.2. LIMIT

If the objective is not to use the player buffer and keep excess bandwidth empty, LIMIT algorithm can be used. To limit the transmission rate, we first need to estimate the available bandwidth. I encountered few problems while estimating bandwidth. Firstly, we emulate the bottleneck link with a Token Bucket Filter. This filter has a certain buffer capacity which can be specified by the user with the following command (see `competitve_tests.py` for details):

```
sudo /sbin/tc qdisc add dev lo parent 1:3 handle 30: tbf rate "BW"mbit
latency 20000ms burst 10000
```

The burst field shows the maximum amount of bytes that tokens can be available for instantaneously [28]. The minimum size of this buffer should be calculated according to the maximum bandwidth. For example, to reach a speed of 10 Mbps, the buffer should be at least 10 KB.

On the other hand, this filter adds some delay after each burst to limit the bandwidth. For instance, if the burst value is 10 KB and our packets are 1024 B:

$$number\ of\ packets\ in\ buffer = \frac{burst}{packet\ size} = \frac{10000}{1024} \simeq 9.76 \tag{5.4}$$

This means that the Token Bucket Filter will add a certain delay after each approximately 10 packets to limit the bandwidth. Therefore, when we send a random number of packets in one congestion window, the delays will be between 10 packets and there will be no delay for congestion windows that are smaller than 10 packets. In GAMZE, this situation does not create a problem, because we send each congestion window without waiting in between and we can emulate the bandwidth correctly. However, we wait some time between congestion windows to send the packets at a certain video quality which reduces the accuracy of our bandwidth estimates. To increase the accuracy, burst should be decreased; but it should be above a certain value to reach the bandwidth at the same time because of the precision of time in a computer. This is one of the problems in estimating bandwidth in LIMIT algorithm.

The other problem occurs when there is more than one stream. Because we have some idle intervals for each stream, some congestion windows of different streams do not overlap and bandwidth seems higher than it is. To overcome this difficulty, I decided to take the minimum rate estimation of the congestion control algorithm over last 15 rates. However, this might not be the best way to estimate the bandwidth because it responds late when a stream leaves, that is, the increase in bandwidth, although it ignores the deceiving high predictions.

During the experiments, I observed another problem. When CWND values of different streams are similar to each other, the bandwidth estimation is the worst. Therefore, I added some randomness to CWND values. Firstly, the initial value of CWND is a random integer between 20 and 50. Then, when a stream reaches to CWND that is larger than a random value between 100 and 150, CWND is reinitialized to a random number between 40 and 80. This precautions result in more variant CWND values and lead to better estimations of bandwidth.

After the estimation of bandwidth, we calculate the pace interval value which is the time window in which we send CWND number of packets. We find the highest quality level lower than the predicted bandwidth. Then, the pace interval is calculated as in Eq. 5.5.

$$pace\ interval = \frac{CWND * packet\ size}{determined\ bitrate} \tag{5.5}$$

After sending CWND number of packets, the program waits until the pace interval time expires and sends the next CWND packets. The pseudo-code of `LIMIT` is as follows:

```
BITRATES = [300, 750, 1200, 1850, 2850, 4300]
i = length(bitrates)
determined_quality = mean(sorted(bitrates[i-15:i-1])[0:2])

# find out matching quality level
tmp_bitrate = determined_quality
tmp_quality = 0
i = 5
while i >= 0:
    if tmp_bitrate >= BITRATES[i]:
        tmp_quality = i
        break
    tmp_quality = i
    i -= 1

# if quality is downgraded, check buffer to prevent switching to lower quality
if i < 5:
    if last_quality > tmp_quality:
        if BufferOccupancy[i-1] + 4 * determined_quality / BITRATES[tmp_quality+1] > 8.5
            tmp_quality += 1

pace_interval = CWND * PACKET_SIZE / BITRATES[tmp_quality] / 1000 * 0.93

# check whether buffer requires a downgrade
if i > 0:
    if BufferOccupancy[i-1] < 4.5:
        tmp_quality -= 1

next_quality = tmp_quality
```

Initially, the bandwidth is estimated as the average of the lowest two rates in last 15 congestion window. After finding the corresponding quality level, the buffer situation is considered. If new quality is lower than the last one, we check whether buffer can meet a higher quality to prevent switching. If buffer occupancy is enough, quality is increased. Then, the pace interval is calculated. Note that 0.93 is chosen to reduce the interval to slightly fill the buffer and

prevent unnecessary rebuffering in case of incoming streams. Lastly, we check if the buffer is close to emptiness and reduce the quality to keep a certain buffer value. Most importantly, we do not change the pace interval value, we keep it lower to fill the buffer more quickly.

In short, it is possible to limit the transmission rate with this algorithm. Yet, the bandwidth estimation requires some improvement. Due to time constraints, I continued with this algorithm.

6. TEST ENVIRONMENT

In this project, we aimed to make better quality decisions by combined ABR and congestion control algorithms. The details of the new algorithm are presented in Section 5. In this section, the test environment to try and compare the new algorithms is explained.

Main tests consist of a certain number of video players (number of streams) which can start playing at determined times. The bandwidth is assumed to be constant in a bottleneck link. In these tests, we also assumed that all users use the same ABR algorithm. The tests can be conducted either with a direct TCP connection between ABR server and Video Server or with the UDP setup explained in Section 4.

For our new `GAMZE` algorithm, we use the new UDP setup because we need to change the congestion algorithm and changing TCP congestion control was not possible. For all other ABR algorithms, tests can be performed with both setups. However, if one wants to stream videos with different algorithms and if one of them is `GAMZE`, then UDP setup should be used.

Test environment includes a bunch of files. Firstly, test descriptions (jobs) are given in JSON files. Secondly, DASH player is used on the client side which is written in JavaScript. HTML and video files together with DASH player are provided in *"html"* folder. All other codes are written in Python. The test environment is shown in Fig. 6.1 and explained in the following sections.

6.1. PYTHON SCRIPTS

6.1.1. `COMPETITIVE_TESTS.PY`

The main program is `competitive_tests.py`. It can run both UDP and TCP setups. User can select the setup and specify the ABR algorithm to be used via JSON files as explained in Section 6.2.2. Also, there are some optional command-line arguments:

- **-b:** It is used to not open the browser(s) that displays video.
- **-t:** It is used to add background TCP traffic.

Firstly, `competitive_tests.py` reads JSON files that include video jobs to be executed and generates log files for each video stream. It also throttles the bandwidth by using Linux Traffic Control with `tc` command. First, the program removes any network shaping files and generates a new one. Then, it adds a Token Bucket Filter (`tbf`) to limit the bandwidth. For UDP setup, only UDP ports through which video server sends video packets are throttled with the help of network shaping filters because the TCP connection between proxy and ABR server should not be throttled. For TCP setup, all ports are throttled, so filters are not used in this case.

Moreover, `competitive_tests.py` is responsible for starting other necessary scripts as shown in Fig. 6.1. If background TCP traffic is turned on by user with `-t` command, the program starts `tcp_traffic.py` and `tcp_traffic_server.py` with the help of `subprocess` module. Then, it starts `run_browser_test.py` for both TCP and UDP setups. Finally, if UDP

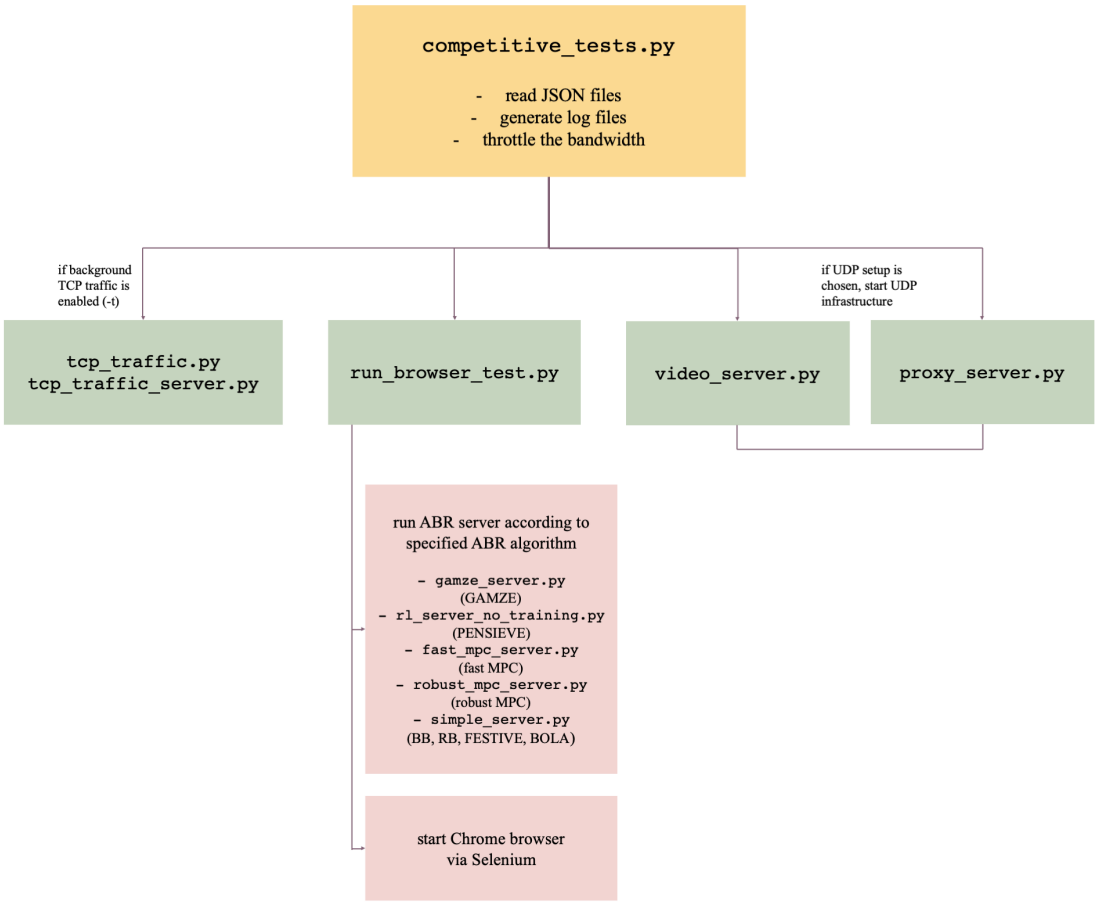

Figure 6.1: Test Environment.

setup is chosen, it starts a `video_server.py` and a `proxy_server.py` for each stream.

6.1.2. `TCP_TRAFFIC.PY` AND `TCP_TRAFFIC_SERVER.PY`

`tcp_traffic.py` sends some web page requests to the localhost server with Poisson intervals. It sends the requests by opening subprocesses with `client.py` for each different request. The sample web pages are stored in `/var/www/html/websites` folder. The localhost server which is run with `tcp_traffic_server.py` script provides the stored web pages. Port 5600 is allocated for this server and it is also throttled separately in UDP setup.

6.1.3. `RUN_BROWSER_TEST.PY`

This script determines which `.html` file will be open according to selected setup (TCP, UDP) and ABR algorithm (GAMZE, PENSIEVE, BB, RB, ...). Then, it starts the ABR server corre-

sponding to the selected ABR algorithm as shown in Fig. 6.1. Depending on the ABR algorithm, the operation slightly differs. In BB, RB, FESTIVE and BOLA, next chunk quality decision is made in `dash.all.min.js`. However, in other cases (GAMZE, PENSIEVE, fast MPC, robust MPC), the decision is made in ABR server and sent to the media player (`dash.all.min.js`); therefore, they have separate ABR server scripts.

Note: `dash.all.min.js` includes a function called *"nextChunkQuality"* which determines the quality of next chunk. It includes various cases and the case number is provided in the `.html` file as `abr_id`. For cases that the decision is made by the ABR server (GAMZE, PENSIEVE, fast MPC, robust MPC), `abr_id` is given 4. Also, it should be noted that the first chunk quality has a default quality level which is given as a constant in `dash.all.min.js`. The name of the constant is *DEFAULT_VIDEO_BITRATE* and it should be set to the desired bitrate in Kbps. In this project, it is equal to the second bitrate level, namely 750. If the specified bitrate is not available highest lower bitrate level is used.

After starting ABR server, `run_browser_test.py` will open the Chrome browser unless user disables it with `-b` command. To run the browser, `selenium` module's `webdriver` is used. The program also allows use of *QUIC*, but this property is not used in this project.

6.1.4. VIDEO_SERVER.PY

`video_server.py` is only used in UDP setup. It both includes the congestion control algorithm and the ABR algorithm. Mainly, the video server takes the video request coming from the browser (by passing through the proxy server) and sends the requested video chunk to the proxy server. While doing this operation, it detects the available bandwidth by measuring RTT of packets and adjusts the congestion window (cwnd). In addition, it decides on the next chunk quality depending on buffer occupancy and available bandwidth and sends this decision to ABR server directly. The details of the algorithms can be found in Section 5.

6.1.5. PROXY_SERVER.PY

As explained above, the proxy server is only implemented because the browser is unable to communicate with UDP. Therefore, a proxy server is used as an intermediate step when UDP setup is preferred. Moreover, proxy servers send ACKs to the video server.

6.2. OTHER USEFUL FILES

6.2.1. VIDEO

I used an EnvivioDASH3 video of 193 seconds. It consists of 49 video chunks of 4 seconds (the last chunk is 3 s) and a header file for all quality levels. This video has six quality levels which are [300, 750, 1200, 1850, 2850, 4300] Kbps.

6.2.2. JSON FILES

`json` files contain the details of each experiment. An example is provided below for two streams. File name and `test_id` should be in the specific format: *"test number_ ABR_number _streams_bandwidth"*. `trace` determines the network trace file that will be used (can be found in */network_traces* folder) and hence the bandwidth. `video` specifies the video to be used and `repeat_n` shows how many times the experiment will be repeated. `udp` is either True or False and used to select UDP or TCP setup. Lastly, `jobs` part gives the details for each stream. `abr` shows which ABR algorithm will be used. `quic` and `transport` fields are not used in this project. `start` shows when the stream will enter to experiment and `duration` is always chosen equal to *"duration = start + 200"* to wait the end of the video.

```
{
    "test_id":"301_GAMZE_2streams_4.8Mbps",
    "comment":"two GAMZE fighting for 4.8Mbps",
    "trace":"4.8Mbps",
    "video":"testVideo",
    "repeat_n":"1",
    "udp":"true",

    "jobs":[

            "name":"GAMZE1",
            "abr":"GAMZE",
            "quic":"false",
            "transport":"cubic",
            "start":"0",
            "duration":"200"
        ,

            "name":"GAMZE2",
            "abr":"GAMZE",
            "quic":"false",
            "transport":"cubic",
            "start":"30",
            "duration":"230"

    ]
}
```

6.3. EXAMINING RESULTS

There are few different files which calculate performance metrics and visualize the results of experiments.

6.3.1. PLOTRESULTS.PY

This function reads log files for each stream and calculates all values given in Eq. 7.1-5 by calling `mqoe.py`. It also plots bitrate level vs. time and buffer occupancy vs. time graphs.

6.3.2. PLOTRESULTS_TCP.PY

This function is the modified version of `plotresults.py` and used when there is TCP traffic in the experiment. Additionally, it puts vertical lines to bitrate level vs. time graph when TCP request is sent (yellow line) and content is received (grey line).

6.3.3. PLOTRESULTS_LIMIT.PY

This function is the modified version of `plotresults.py` and used for `LIMIT` algorithm. It adds another plot to show determined quality values. TCP section can be uncommented/-commented to add vertical TCP requests lines.

6.3.4. PLOT_RTT-CWND.PY

This function is used to examine what happens inside the `GAMZE` and `LIMIT` algorithms. It plots sample RTT, estimated RTT, pace interval, bitrate estimate, determined quality and CWND vs. time.

7. RESULTS

We performed five different experiments and compared `GAMZE` and `LIMIT` with other ABR algorithms. For `GAMZE` and `LIMIT`, UDP setup is used, for others, TCP setup is used. The assumption is a bottleneck link with a fixed bandwidth but video streams can enter and leave which causes changes in bandwidth.

To compare the results, quality of experience (QoE) metric is used . The parameters are chosen as follows: $q(R_k) = R_k$ where R_k equals to the bitrate of chunk k in Mbps, $\delta = 0$, $\lambda = 4.3$ and $\mu = 1$. The final equation is as follows:

$$QoE = \frac{\sum_{k=1}^{N} R_k - 4.3 \sum_{k=2}^{N} T_{buf,k} - \sum_{k=1}^{N-1} |R_k - R_{k+1}|}{N} \tag{7.1}$$

This equation was used to train PENSIEVE and MPC also tries to optimize QoE with these parameters. In the tables below, there are some other QoE values in addition to QoE in Eq. 7.1.

$$Quality\ QoE = \sum_{k=1}^{N} R_k \tag{7.2}$$

$$Rebuffer\ QoE = -4.3 \sum_{k=2}^{N} T_{buf,k} \tag{7.3}$$

$$Switch\ QoE = -\sum_{k=1}^{N-1} |R_k - R_{k+1}| \tag{7.4}$$

$$Fairness = 1 - 2\,\frac{std(QoEs)}{Highest\_QoE - Lowest\_QoE} \tag{7.5}$$

7.1. `GAMZE`

In `GAMZE` algorithm, there is a parameter called *BUFFER_UPGRADE_LIMIT* which specifies the buffer occupancy at which a quality upgrade is made even though the bandwidth is not enough. Hence, ABR algorithm `GAMZE` is tested with various *BUFFER_UPGRADE_LIMIT* values and compared with other algorithms without TCP traffic.

When *BUFFER_UPGRADE_LIMIT* gets higher and higher, switching to better quality is delayed. However, it is easier to keep the higher quality for a long time after the upgrade decision as the buffer occupancy is high. This reduces the switching cost which decreases the QoE. On the other hand, we do not know how long the viewer will watch the video and we do not increase the quality although it is possible. This can reduce the QoE, especially in a more dynamic environment. Therefore, the choice of parameter *BUFFER_UPGRADE_LIMIT* depends on the environment and expectation from the ABR.

7.1.1. 2 STREAMS - 4.8 MBPS

In this experiment, there are two streams in a 4.8 Mbps link. The second stream enters 30 s later than the first one. The Quality of Experience (QoE) values are presented in Table 7.1. The values are the averages of two streams and *BUFFER_UPGRADE_LIMIT* is given in parentheses for `GAMZE`.

Table 7.1: Quality of Experience for 2 streams with 4.8 Mbps Bandwidth

	GAMZE (10)	**GAMZE (15)**	**GAMZE (20)**	**BB**	**RB**
QoE	2.167	2.222	2.201	1.830	1.768
Quality QoE	119.100	116.825	114.600	114.325	102.575
Rebuffer QoE	0	0	0	0	-11.169
Switch QoE	-12.925	-7.925	-6.775	-24.675	-4.775
Fairness	0.99	0.99	0.93	1	0.96

	FESTIVE	**fastMPC**	**robustMPC**	**PENSIEVE**	**BOLA**
QoE	1.855	1.743	1.946	2.127	1.989
Quality QoE	93.500	117.750	115.550	110.200	100.050
Rebuffer QoE	0	-17.056	-8.269	0	0
Switch QoE	-2.600	-15.275	-11.925	-6.000	-2.600
Fairness	1	0.92	0.98	0.99	0.97

In general, `GAMZE` algorithm outperforms all other ABR algorithms. The main reason for this performance is that we have more knowledge about the system, bandwidth estimation from congestion control part is very accurate and allows us to foresee next buffer occupancy. Therefore, we are able to make more informed decisions. The only possible problem can occur when the second stream enters in the middle of the download of the chunk of the first stream because bandwidth halves suddenly. However, even this situation does not cause any rebuffering since we always keep buffer occupancy at least above 2 seconds. It can be seen from the table that `GAMZE` prevents any rebuffering.

Moreover, switching QoE should be higher (closer to zero) which corresponds to less switching. We can see that increasing *BUFFER_UPGRADE_LIMIT* reduces switching, but also lowers quality since we upgrade after a larger buffer occupancy. Therefore, there is a trade-off between switching and quality depending on *BUFFER_UPGRADE_LIMIT*. According to experimental results, *BUFFER_UPGRADE_LIMIT=15* seems to be a good compromise to maximize general QoE.

In Fig. 7.1, bitrate and buffer occupancy in time is given for `GAMZE10`. The high number of switching for *BUFFER_UPGRADE_LIMIT=10* can be understood from this figure. Firstly, we see that the first stream reaches to 4300 Kbps quality after staying at one lower quality for few chunks because it needs to fill the buffer up to a certain value. When the second stream enters, the first stream reduces its quality to fill the buffer again and they start to fill their buffers at 1850 Kbps. Because *BUFFER_UPGRADE_LIMIT* is low, the buffers quickly reach to 10 s and both streams upgrade. However, they quickly hit the lower limit of the buffer so they return to 1850 Kbps after few chunks. This loop continues until the first stream leaves.

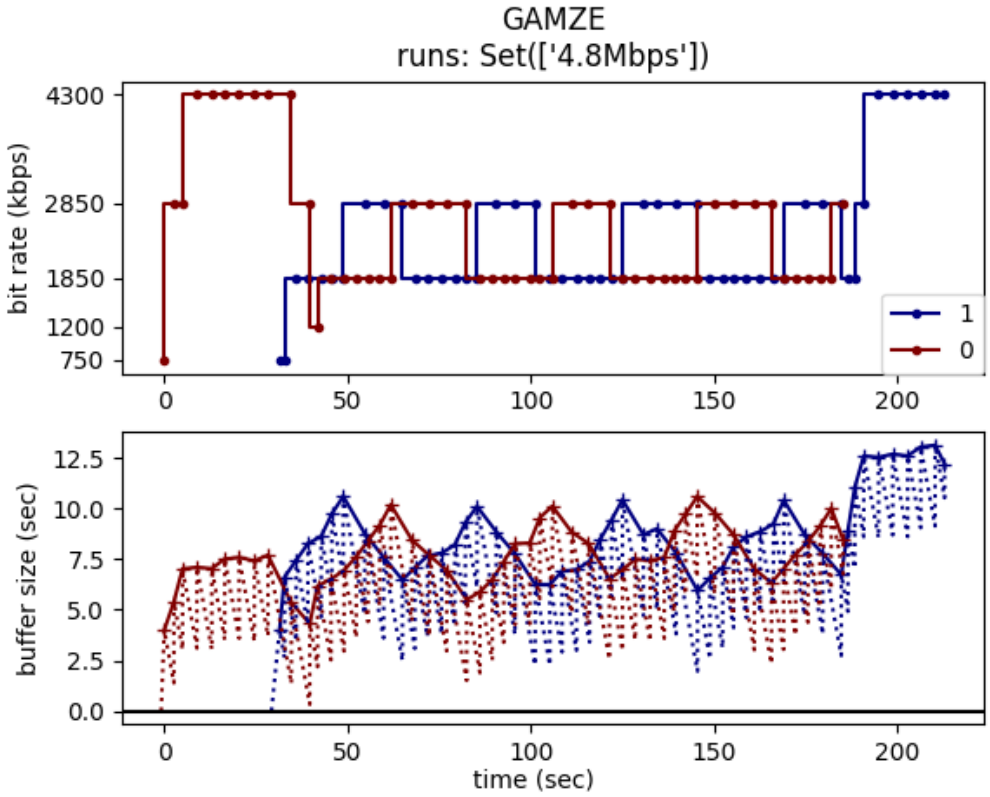

Figure 7.1: `GAMZE10` - 2 streams with 4.8 Mbps.

In Fig. 7.2, bitrate and buffer occupancy in time is given for `GAMZE15`. Firstly, we see that the first stream reaches to 4300 Kbps quality after staying at one lower quality for few chunks. When the second stream enters, two streams come to equilibrium at 1850 Mbps rapidly. Around 80 s, buffer occupancy of streams reaches to 15 s and they upgrade to 2850 Kbps even though the bandwidth cannot support this quality. Around 130 s, both streams reduce their qualities because the buffer occupancy declines and enters to a risky region. Then, the buffer occupancy starts to increase again because each stream has approximately $2400 - 1850 = 550$ Mbps excess bandwidth.

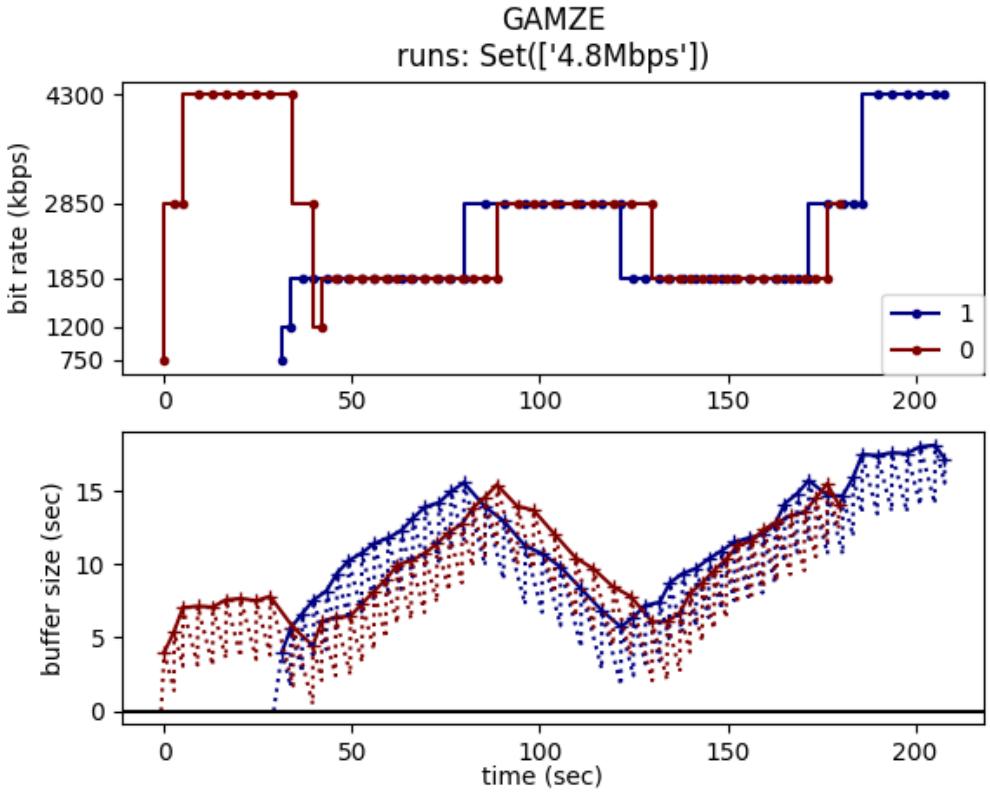

Figure 7.2: `GAMZE15` - 2 streams with 4.8 Mbps.

Lastly, bitrate and buffer occupancy for `GAMZE20` is provided in Fig. 7.3. We see that streams stay at 1850 Kbps for a longer period and upgrade when the buffer reaches to 20 s. This wider upgrade and downgrade limits in buffer occupancy results in less switching. However, time spent in 2850 Kbps is lower due to the same reason.

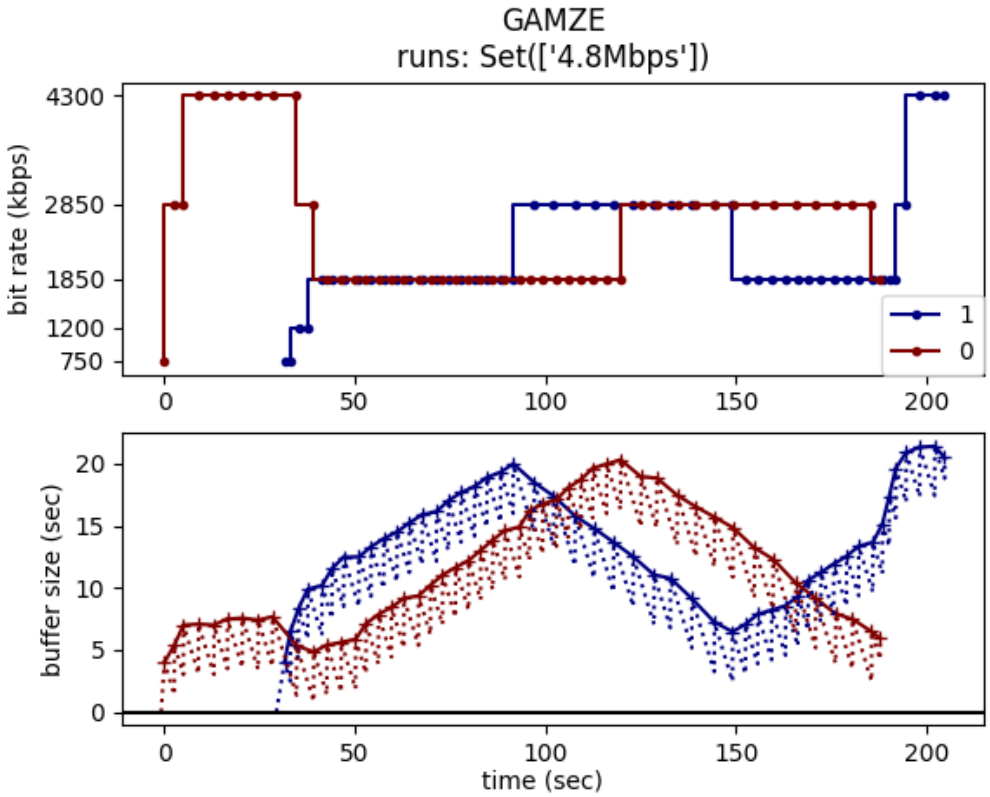

Figure 7.3: `GAMZE20` - 2 streams with 4.8 Mbps.

The closest rival of GAMZE is PENSIEVE. In Fig. 7.4, PENSIEVE seems more conservative about upgrading. For example, the first stream initially stays at 1850 Kbps for a much longer time. Also, PENSIEVE only switches to the highest quality after filling buffers to more than 30 s. Therefore, it has high switch QoE, better than GAMZE. However, the quality QoE is lower due to its late upgrade.

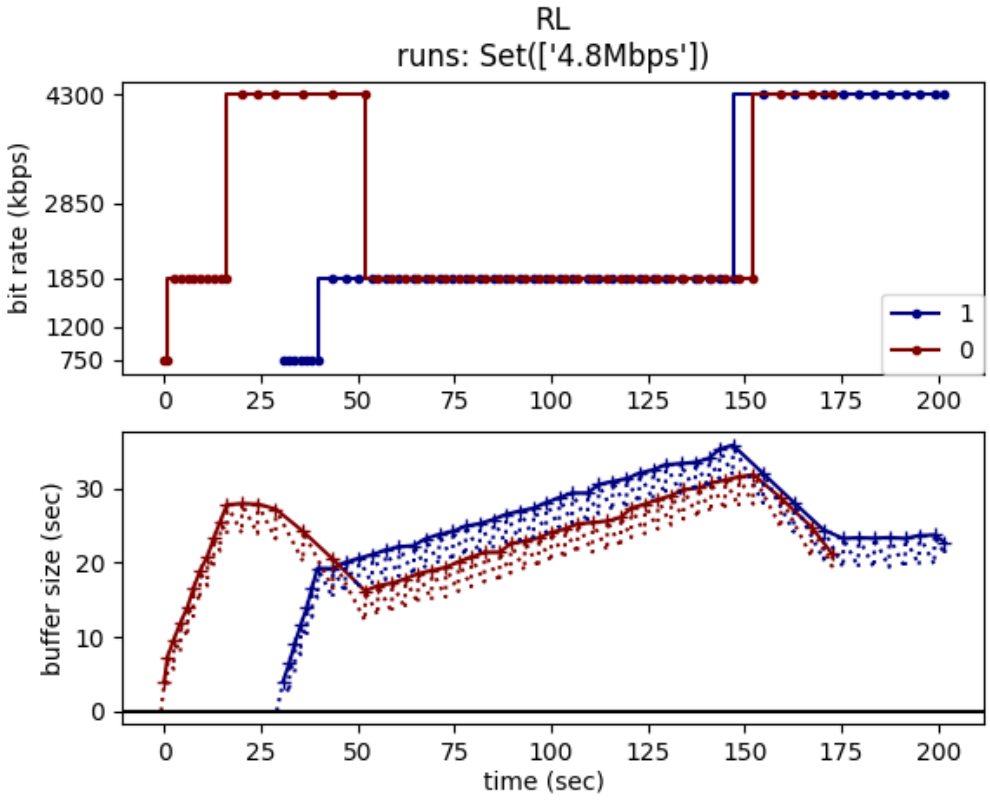

Figure 7.4: PENSIEVE - 2 streams with 4.8 Mbps.

The graphs for other algorithms can be found in Appendix C.

7.1.2. 2 STREAMS - 4.8 MBPS WITH TCP TRAFFIC

We also wanted to test how our new algorithm behaves when there is additional TCP traffic in the system. To test all algorithms fairly, TCP traffic is predetermined. During the experiment, six TCP requests for six different websites arrive at given times with Poisson intervals. The results of the experiments are presented in Table 7.2.

Table 7.2: Quality of Experience for 2 streams with 4.8 Mbps Bandwidth and TCP Traffic

	GAMZE (10)	**GAMZE (15)**	**GAMZE (20)**	**BB**	**RB**
QoE	2.132	2.202	2.094	1.718	1.880
Quality QoE	115.875	114.650	112.200	111.625	99.600
Rebuffer QoE	0	0	-2.152	0	-2.042
Switch QoE	-11.425	-6.775	-7.425	-27.425	-5.425
Fairness	0.99	0.99	0.97	1	0.95

	FESTIVE	**fastMPC**	**robustMPC**	**PENSIEVE**	**BOLA**
QoE	1.744	1.634	1.939	1.952	1.920
Quality QoE	88.225	114.525	111.325	100.400	97.775
Rebuffer QoE	0	-23.186	-4.072	0	-0.127
Switch QoE	-2.750	-11.275	-12.225	-4.775	-3.575
Fairness	0.94	0.86	0.98	0.99	0.96

Similar to the case without TCP traffic, `GAMZE15` outperforms the other ABR algorithms, again. We see that quality QoE reduces for all algorithms. Switch and rebuffer QoE is better, same or worse for different algorithms. In Table 7.3, the changes in all QoE values with the addition of TCP traffic can be seen. Only, `RB` performs better than it was. The closest ABR algorithm to `GAMZE` is again `PENSIEVE`. However, the reduction in performance of `PENSIEVE` (8.228%) is more significant compared to `GAMZE` (1.615%,0.900%, 4.861%).

Table 7.3: Changes in Quality of Experience with TCP Traffic for 2 streams with 4.8 Mbps Bandwidth

	GAMZE (10)	**GAMZE (15)**	**GAMZE (20)**	**BB**	**RB**
QoE	-0.035	-0.020	-0.107	-0.112	+0.112
Quality QoE	-3.225	-2.175	-2.400	-2.700	-2.975
Rebuffer QoE	0	0	-2.152	0	+9.127
Switch QoE	+1.500	+1.150	-0.650	-2.750	-0.650
Fairness	0	0	+0.04	0	-0.01

	FESTIVE	fastMPC	robustMPC	PENSIEVE	BOLA
QoE	-0.141	-0.109	-0.007	-0.175	-0.069
Quality QoE	-5.275	-3.225	-4.225	-9.800	-2.275
Rebuffer QoE	0	-6.130	+4.197	0	-0.127
Switch QoE	-0.150	+4.000	-0.300	+1.225	-0.975
Fairness	-0.06	-0.06	0	0	-0.01

When we compare `GAMZE` with other algorithms, we see that answering TCP requests take much longer time. This is not related to ABR algorithms but the setups. In Fig. 7.5, `GAMZE15` with TCP traffic is shown. When we compare this figure with Fig. 7.6 which shows the same experiment with `PENSIEVE`, it is obvious that TCP requests take far more time to answer in UDP setup. This means that our UDP setup is not as fair as TCP setup to other types of requests. The other algorithms behave similarly and graphs can be found in Appendix D.

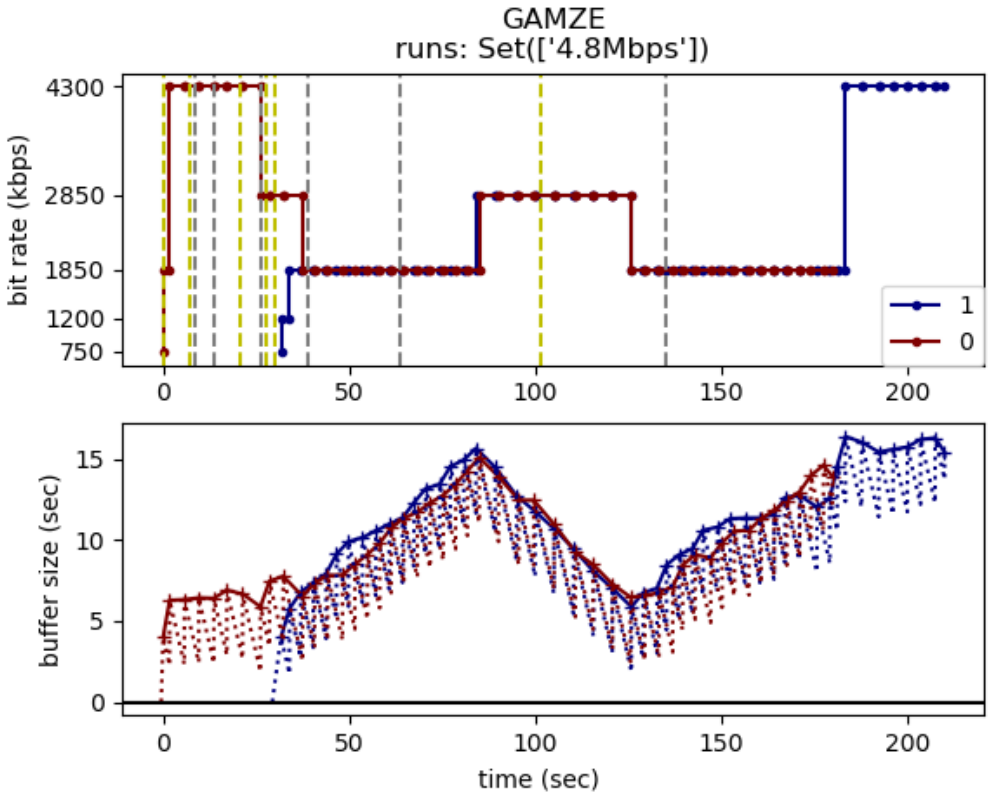

Figure 7.5: `GAMZE15` - 2 streams with 4.8 Mbps with TCP Traffic.

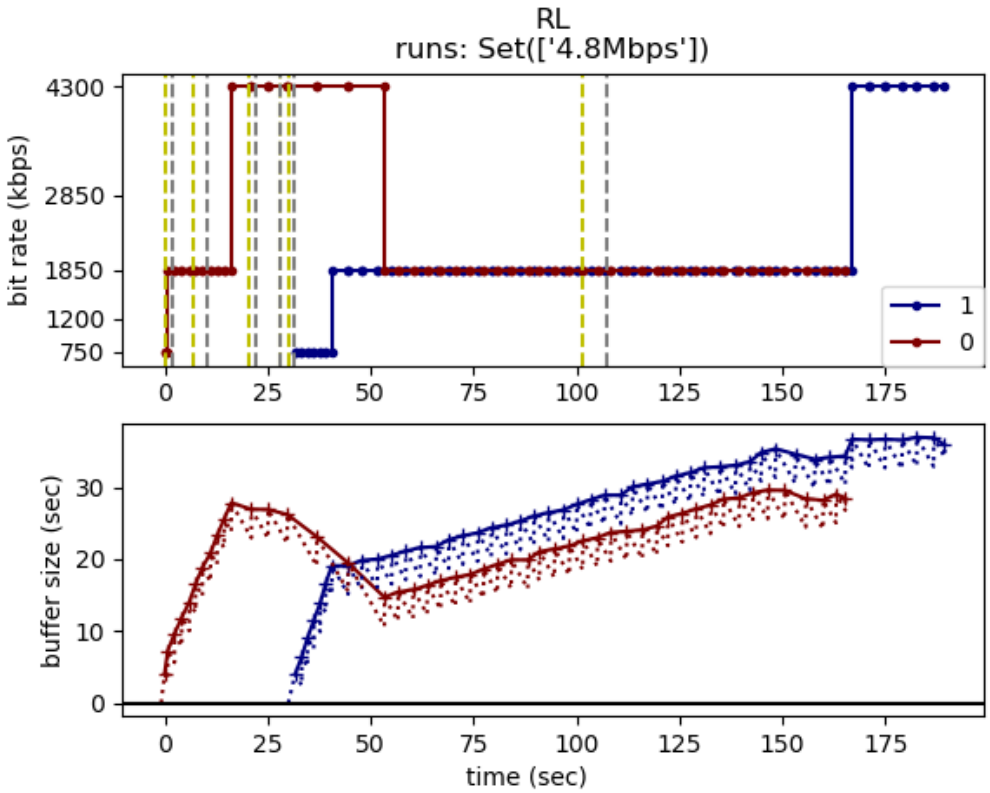

Figure 7.6: PENSIEVE - 2 streams with 4.8 Mbps with TCP Traffic.

We also see that algorithms behave similarly to the case without TCP traffic. However, upgrades are delayed in general. This results in lower QoE values. QoE of PENSIEVE reduces more because the first stream is not upgraded again when there is TCP traffic.

7.1.3. 6 STREAMS - 10 MBPS

In this experiment, there are six streams in a 10 Mbps link. Two streams start together and one enters after 30 s, two enters after 50 s and last one after 100 s. The Quality of Experience (QoE) values are presented in Table 7.4. The values are the averages of six streams.

Table 7.4: Quality of Experience for 6 streams with 10 Mbps Bandwidth

	GAMZE (10)	**GAMZE (15)**	**GAMZE (20)**	**BB**	**RB**
QoE	1.835	1.885	1.845	1.574	1.404
Quality QoE	101.558	99.733	97.950	96.292	87.442
Rebuffer QoE	0	0	0	0	-13.964
Switch QoE	-10.108	-7.367	-7.542	-19.175	-4.675
Fairness	0.9	0.88	0.9	0.88	0.87

	FESTIVE	**fastMPC**	**robustMPC**	**PENSIEVE**	**BOLA**
QoE	1.553	1.222	1.659	1.728	1.617
Quality QoE	78.700	98.142	97.825	90.467	82.942
Rebuffer QoE	0	-26.432	-6.313	0	-0.151
Switch QoE	-2.608	-11.825	-10.208	-5.792	-3.575
Fairness	0.86	0.82	0.87	0.89	0.83

Similar to the experiment with 2 streams, we see that `GAMZE` outperforms other algorithms and *BUFFER_UPGRADE_LIMIT=15* is the best for 6 streams. Quality QoE is highest for `GAMZE10` as expected, but switch QoE of `GAMZE20` is a little worse than `GAMZE15`. Yet, this is expectable as there are more streams and more randomness involved. The closest QoE from other algorithms belongs to `PENSIEVE` with better switch QoE, but quality QoE is much lower.

In Fig. 7.7, bitrate decisions and buffer occupancy vs. time is presented for `GAMZE15`. For each stream, approximate bandwidth can be thought as $10/6 = 1.67 Mbps$, thus we expect streams to have at least quality of 1200 Kbps. After 100 s, all streams are live and we see that quality levels change between 1200 and 1850 Kbps as we expected. Between 50 and 100 s, there are 5 streams and each stream has $10/5 = 2 Mbps$ bandwidth. Therefore, we expect the quality to be 1850 Kbps; however, it reduces to 1200 Kbps for streams 1 and 5 to save some buffer. Keeping buffer above a certain value causes small glitches at some points and this reduces switch QoE. We also see that streams are not able to fill their buffers up to 15 s most of the time, only 3 streams hit 15 s at around 150 s and upgrade their quality. This shows that some streams are able to hold more bandwidth than others.

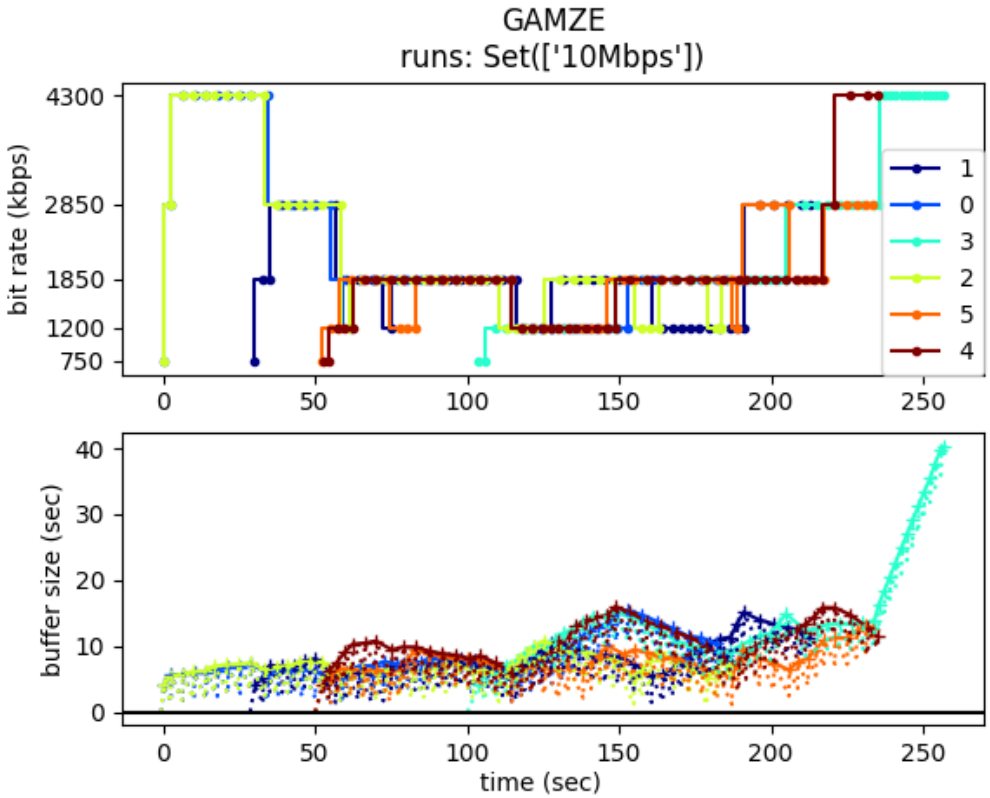

Figure 7.7: GAMZE15 - 6 streams with 10 Mbps.

In Fig. 7.8, the first thing we observe is less switching consistent with higher switch QoE. The buffer occupancy levels are much more in PENSIEVE compared to GAMZE. However, the total quality is lower because PENSIEVE is again more conservative in upgrade decisions. Especially, the difference stems from the region when streams start to leave around 180 s. GAMZE is better in obtaining freed bandwidth, it is more adaptive. On the other hand, PENSIEVE stays at a lower quality for a long period for streams 1, 3, 4, 5. Streams 1, 4, 5 leaves with low quality but high buffer occupancy. Only stream 4 is upgraded as it stays longer. In short, PENSIEVE is not good at adapting to improvements in bandwidth. The figures for other algorithms are presented in Appendix E.

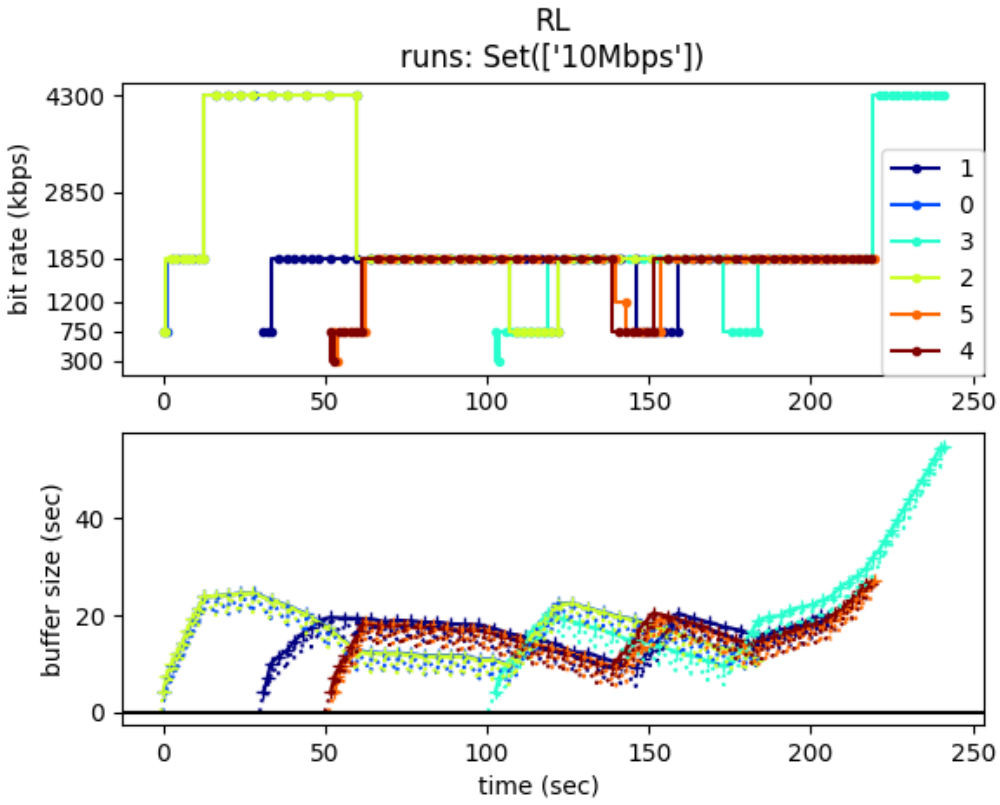

Figure 7.8: PENSIEVE - 6 streams with 10 Mbps.

7.2. LIMIT

7.2.1. 1 STREAM - 2.2 MBPS

I conducted this experiment to test the LIMIT algorithm and compare it to the case with TCP traffic. We expected that the video quality would be 1850 Kbps all the time and TCP traffic should not affect it because there is an excess bandwidth. This experiment confirmed the expectation, exactly same QoE values are obtained with TCP traffic. In Table 7.5, the QoE values for LIMIT with and without TCP traffic (they are the same) can be seen.

Table 7.5: Quality of Experience for 1 Stream with 2.2 Mbps Bandwidth

	LIMIT
QoE	1.792
Quality QoE	88.900
Rebuffer QoE	0
Switch QoE	-1.100

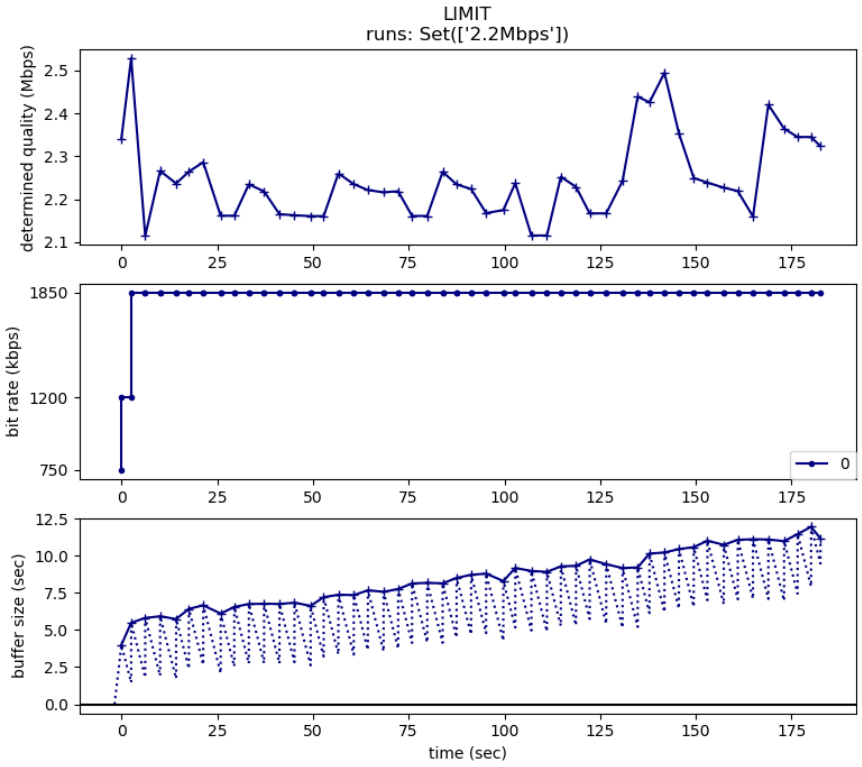

Figure 7.9: LIMIT - 1 stream with 2.2 Mbps.

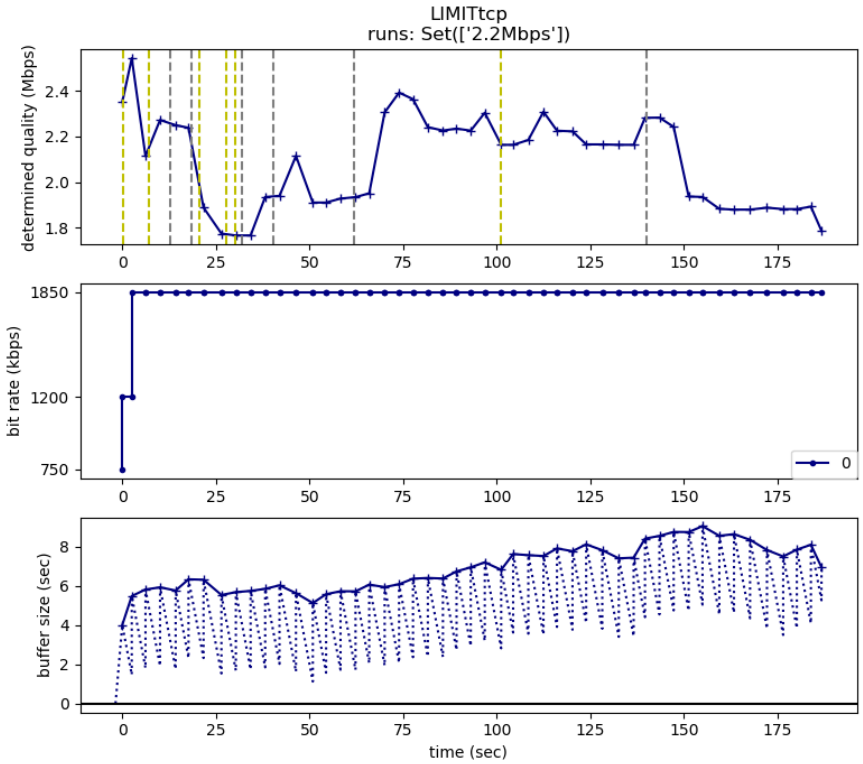

Figure 7.10: `LIMIT` - 1 stream with 2.2 Mbps with TCP Traffic.

In Fig. 7.9, the determined quality (according to bandwidth estimate), the buffer occupancy and quality levels are shown for the case without TCP traffic and in Fig. 7.10, same plot is given for the case with TCP traffic. We see that bitrate decisions are the same in both cases. As we kept pace a bit higher to prevent rebuffering, we see that buffers fill up slightly. However, buffer occupancy levels are lower when there is TCP traffic. The reason is that the excess bandwidth of $2200 - 1850 = 350 Kbps$ is used for both filling buffer and TCP traffic. We also observe that the bandwidth is estimated lower when there is TCP traffic. It should be noted that bandwidth estimation is not accurate enough as explained in Section 5.2.2 and needs some improvements.

7.2.2. 2 Streams - 4.8 Mbps

I tested `LIMIT` algorithm with two streams as well. In Table 7.6, QoE values with and without TCP traffic can be seen. For 2 streams, there is some performance reduction in the case with TCP traffic. All QoE values are lower. Even a 3 s rebuffering occurred.

Table 7.6: Quality of Experience for 2 Streams with 4.8 Mbps Bandwidth

	LIMIT	**LIMIT-TCP**
QoE	2.099	1.932
Quality QoE	110.150	108.000
Rebuffer QoE	0	-3.281
Switch QoE	-7.275	-10.075
Fairness	0.99	1

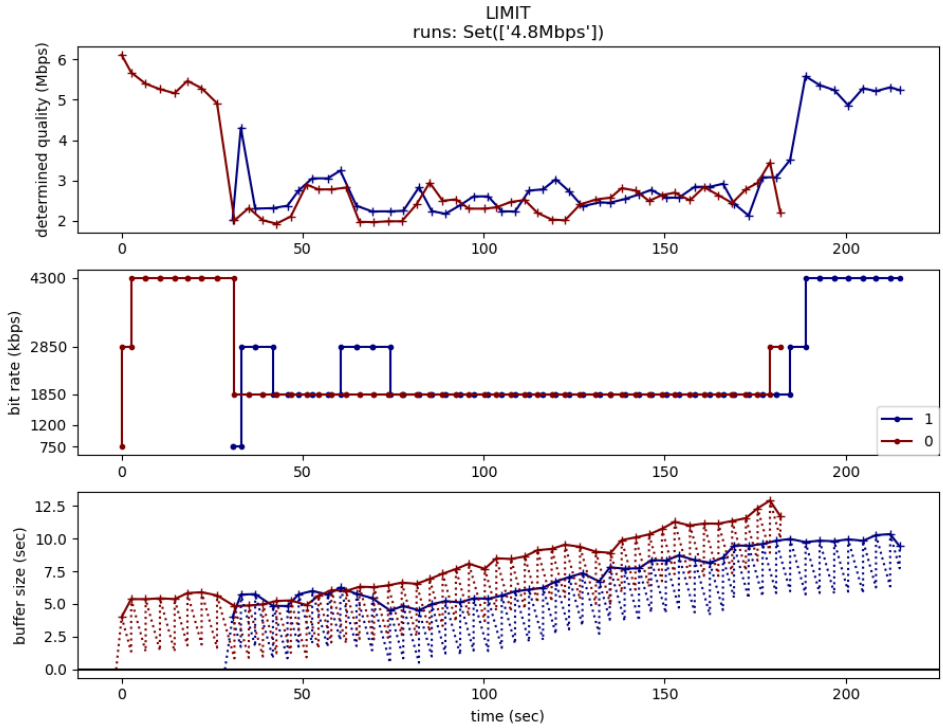

Figure 7.11: `LIMIT` - 2 streams with 4.8 Mbps.

In Fig. 7.11 and 7.12, the determined quality, the buffer occupancy and quality levels are plotted. At some instances, we see that the bandwidth is estimated higher than it is. Buffer occupancy levels are lower when there is TCP traffic. Lastly, the TCP requests were answered more quickly compared to GAMZE.

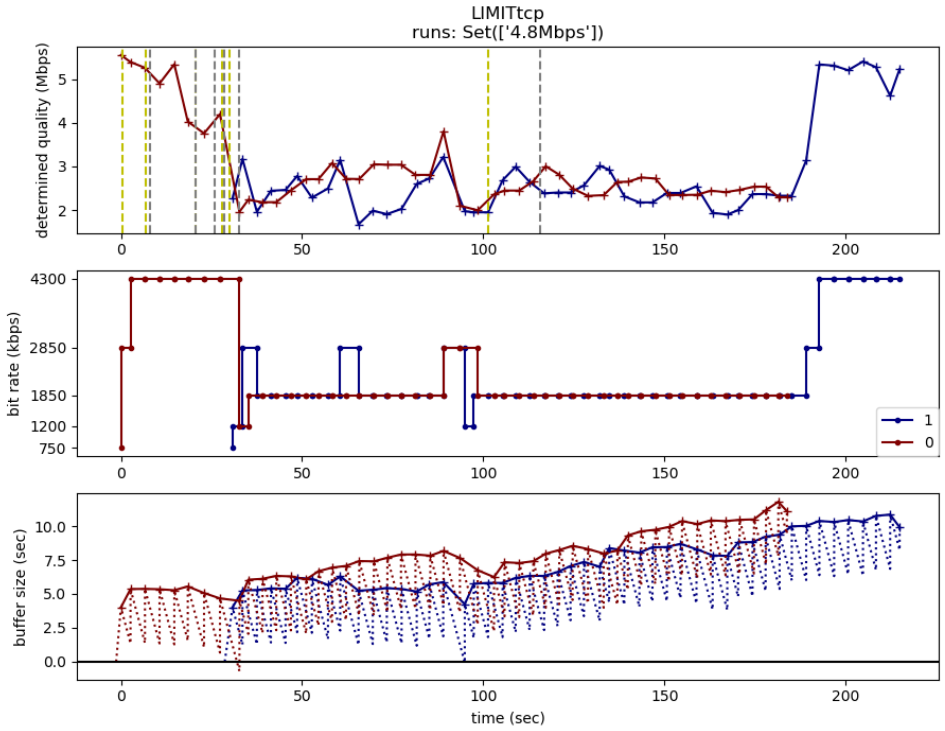

Figure 7.12: LIMIT - 2 streams with 4.8 Mbps with TCP Traffic.

8. CONCLUSION

During two months, we tried to understand whether combining ABR and congestion control algorithms can improve QoE for video players. In short, the answer is yes.

The first approach was to use QUIC to modify congestion control. However, we decided to use another setup after three weeks of trial due to the complexity of QUIC. Then, we set the predesigned UDP implementation and modified it for our purposes. On top of this UDP setup, we implemented a congestion control algorithm and made the protocol reliable. Finally, we designed two different ABR algorithms `GAMZE` and `LIMIT` which work together with congestion control algorithm.

The results of `GAMZE` algorithm have shown that the combination of congestion control and ABR produces better QoE values than any other algorithm under different circumstances. The *BUFFER_UPGRADE_LIMIT* parameter was tested with different values and 15 has been found as a good compromise between quality and switching. After `GAMZE`, `PENSIEVE` has been the best algorithm, but it was not as adaptive when the bandwidth increased.

`GAMZE` outperformed other algorithms when there is TCP traffic as well. Yet, UDP setup is not fair to TCP traffic and download times of websites are much longer compared to TCP setup.

Secondly, `LIMIT` algorithm was designed not to maximize QoE but to use bandwidth more efficiently. However, the bandwidth estimation was not as good as the estimations of `GAMZE` due to idle periods. This caused some degradation in performance.

When there is TCP traffic, we expected `LIMIT` not to lose performance. Although it produced same results with and without TCP traffic for 1 stream, the QoE reduced for 2 streams with TCP traffic.

To sum up, providing communication between congestion control and ABR algorithm improves performance. The accurate bandwidth estimation from congestion control algorithm allows ABR to make better bitrate decisions.

8.1. FUTURE WORK

The first important step may be improving congestion control with a more realistic one. Using QUIC could be the possible solution, but some time should be devoted to implementation. We saw that the current UDP setup fails in fairness with respect to TCP. Improvements in congestion control can also increase fairness to other types of streams.

The bandwidth estimation in `LIMIT` algorithm is not accurate enough and needs improvement. If one wants to use this approach, the estimation part should be revisited first.

Lastly, I did not make experiments where different ABR algorithms are used by different players. The current UDP setup can handle such experiments but JSON file format should be changed and some small adjustments are needed.

REFERENCES

[1] T.-Y. Huang, R. Johari, N. McKeown, M. Trunnell, and M. Watson, "A buffer-based approach to rate adaptation: Evidence from a large video streaming service," *ACM SIGCOMM Computer Communication Review*, 08 2014.

[2] Eyevinn Technology, *Internet Video Streaming - ABR part 2*. https://medium.com/@eyevinntechnology/internet-video-streaming-abr-part-2-dbce136b0d7c.

[3] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach, 7th Edition*. Pearson, 2017.

[4] X. Yin, A. Jindal, V. Sekar, and B. Sinopoli, "A control-theoretic approach for dynamic adaptive video streaming over http," *ACM SIGCOMM Computer Communication Review*, vol. 45, pp. 325–338, 08 2015.

[5] J. Jiang, V. Sekar, and H. Zhang, "Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive," *IEEE/ACM Transactions on Networking*, vol. 22, pp. 326–340, Feb 2014.

[6] Y. Wang and S. Boyd, "Fast model predictive control using online optimization," *IEEE Transactions on Control Systems Technology*, vol. 18, pp. 267–278, March 2010.

[7] H. Mao, R. Netravali, and M. Alizadeh, "Neural adaptive video streaming with pensieve," pp. 197–210, 08 2017.

[8] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "Bola: Near-optimal bitrate adaptation for online videos," in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, April 2016.

[9] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), pp. 183–196, ACM, 2017.

[10] W. Qu, "Congestion control tuning of the quic transport layer protocol," 2018. https://upcommons.upc.edu/bitstream/handle/2117/121934/136386.pdf?sequence=1&isAllowed=y.

[11] *quicwg/base-drafts/Implementations*. https://github.com/quicwg/base-drafts/wiki/Implementations.

[12] L. Clemente, "A quic implementation in pure go." https://github.com/lucas-clemente/quic-go.

[13] *Playing with QUIC*. https://www.chromium.org/quic/playing-with-quic.

[14] *Checking out and building Chromium on Linux*. https://chromium.googlesource.com/chromium/src/+/master/docs/linux_build_instructions.md.

[15] *how_to_build_chromium_project_for_linux*. https://programmer.group/how_to_

build_chromium_project_for_linux.html.

[16] *Linux Cert Management*. https://chromium.googlesource.com/chromium/src/+/master/docs/linux_cert_management.md.

[17] *Error: QUIC_PROOF_INVALID --with verbose details from client*. https://groups.google.com/a/chromium.org/forum/#!topic/proto-quic/mD5dLoXP6BM.

[18] *Netlog Viewer*. https://netlog-viewer.appspot.com/#import.

[19] *Problem using quic toy server*. https://groups.google.com/a/chromium.org/forum/#!topic/proto-quic/LVFkyEDhHac.

[20] *dash.all.min.js*. https://cdn.dashjs.org/latest/dash.all.min.js.

[21] S. Arisu, *QUIC-Streaming*. https://github.com/sevketarisu/quic-streaming.

[22] pari685, *AStream*. https://github.com/pari685/AStream.

[23] pari685, *AStream/dist/sample_mpd*. https://github.com/pari685/AStream/tree/master/dist/sample_mpd.

[24] ITEC-Institute of Information Technology, *Index of /ftp/datasets/DASHDataset2014*. http://www.itec.aau.at/ftp/datasets/DASHDataset2014/.

[25] npm, *webdriver-manager*. https://www.npmjs.com/package/webdriver-manager.

[26] *Thread: How to make /var/www/ writable?* https://ubuntuforums.org/showthread.php?t=1783016.

[27] *ChromeDriver - WebDriver for Chrome*. http://chromedriver.chromium.org/downloads.

[28] J. Fulkerson, *Traffic Shaping with tc*. https://www.badunetworks.com/traffic-shaping-with-tc/.

A. Python Script: add_headers.py

```
import subprocess
import errno
import os
import signal

os.chdir('/home/gamzeisl/www.example.org/media/BigBuckBunny/4sec')
list_bbb = os.listdir('.')
list_bbb = filter(os.path.isdir, list_bbb)
print(list_bbb)
os.chdir('..')
if os.path.exists('4sec_prime') == False:
    os.mkdir('4sec_prime')
os.chdir('4sec')
os.chdir('/home/gamzeisl/Desktop')
for j in list_bbb:
    print('Folder', j)
    os.chdir('/home/gamzeisl/www.example.org/media/BigBuckBunny/4sec')
    os.chdir('../4sec_prime')
    if os.path.exists(j) == False:
        os.mkdir(j)
    os.chdir('../4sec')
    files = os.listdir(j)
    for i in range(len(files)):
        print('File', files[i])
        name = '/home/gamzeisl/www.example.org/media/BigBuckBunny/4sec/' + j \
        + '/' + files[i]
        add = "HTTP/1.1 200 OK\nX-Original-Url:  \
        https://www.example.org/media/BigBuckBunny/4sec/"+j+'/'+files[i]+"\n\n"
        f = open("/home/gamzeisl/Desktop/a.m4s", "w")
        f.write(add)
        f.close()
        myc = 'cat /home/gamzeisl/Desktop/a.m4s '+ name + ' \
        >/home/gamzeisl/www.example.org/media/BigBuckBunny/4sec_prime/' + j \
        + '/' + files[i]
        print(myc)
        os.system(myc)
```

B. Basic Video Streaming: index.html

```
HTTP/1.1 200 OK
X-Original-Url: https://www.example.org/index.html

<!doctype html>
<html>
    <head>
        <title>Dash.js Rocks haha</title>
<script src="https://www.example.org/dash.all.min.js"></script>
    </head>
    <body>
   <div>
       <video data-dashjs-player muted autoplay src="Manifest.mpd" controls></video>
   </div>
    </body>
</html>
```

C. 2 Streams - 4.8 Mbps

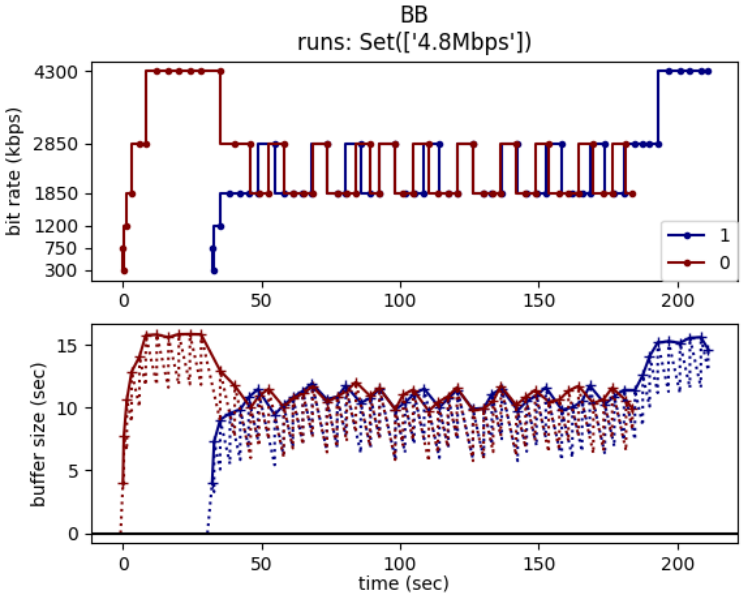

Figure C.1: BB - 2 streams with 4.8 Mbps.

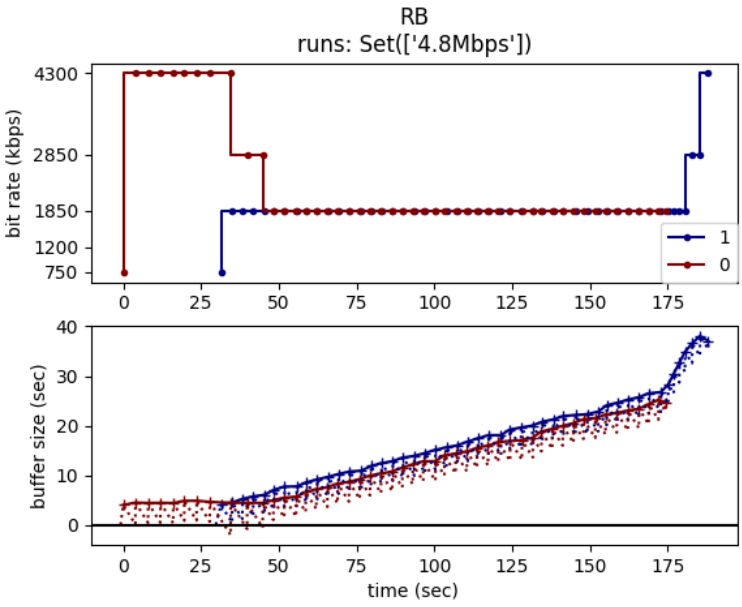

Figure C.2: RB - 2 streams with 4.8 Mbps.

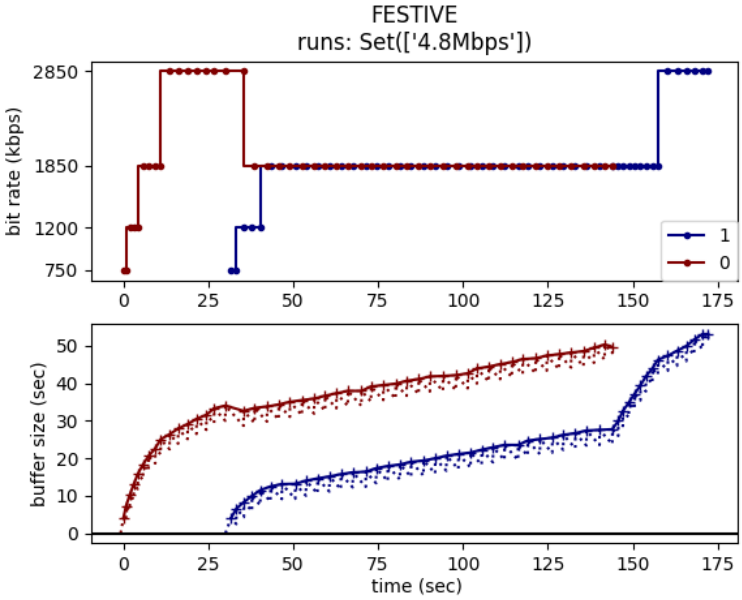

Figure C.3: `FESTIVE` - 2 streams with 4.8 Mbps.

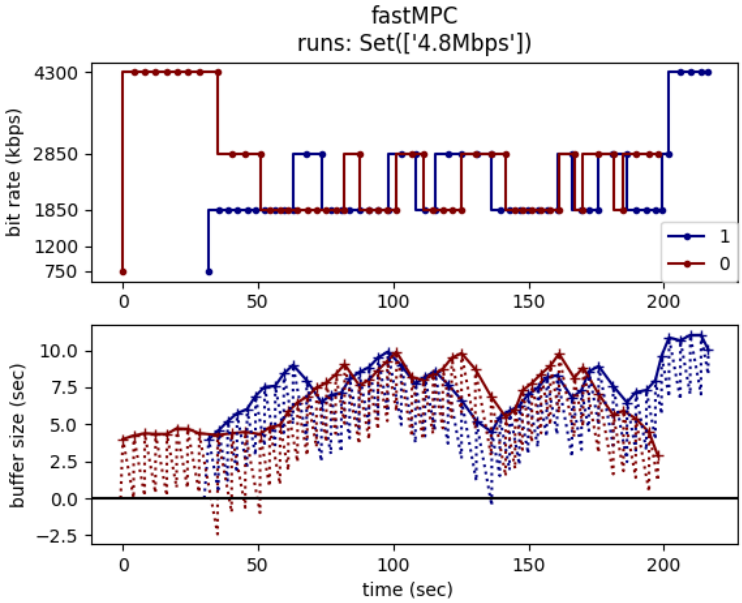

Figure C.4: `fastMPC` - 2 streams with 4.8 Mbps.

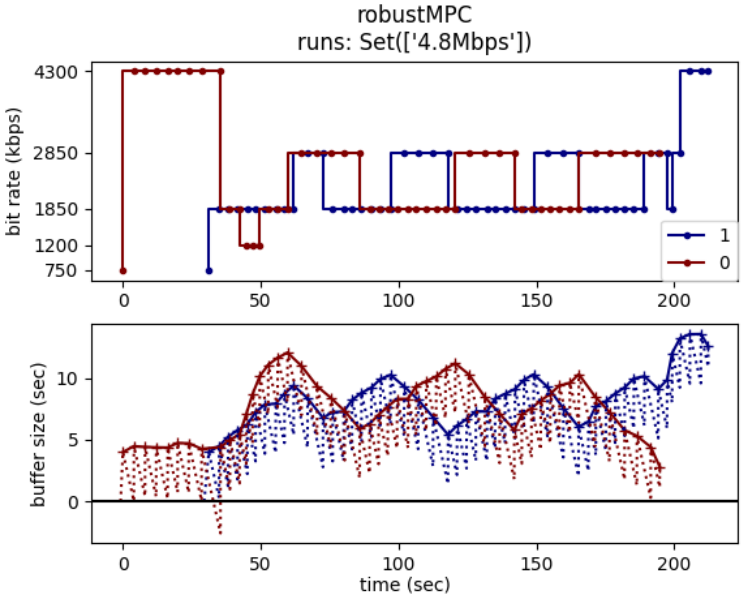

Figure C.5: `robustMPC` - 2 streams with 4.8 Mbps.

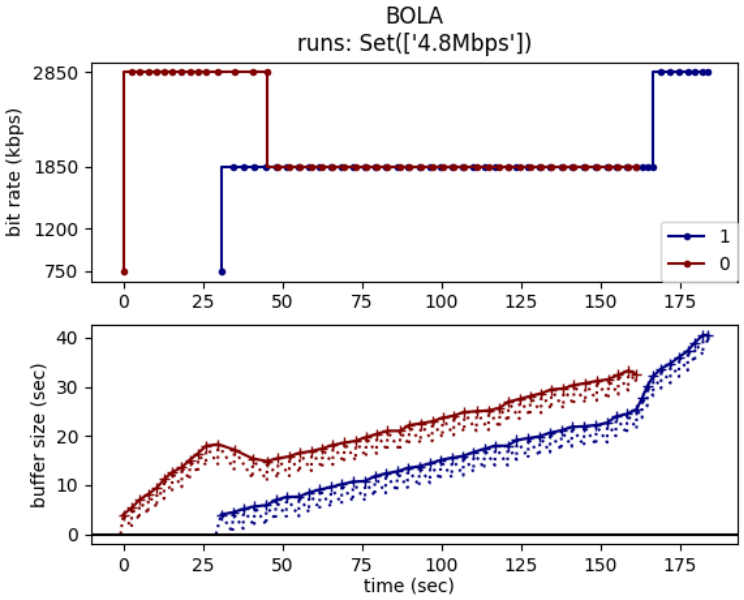

Figure C.6: `BOLA` - 2 streams with 4.8 Mbps.

D. 2 Streams - 4.8 Mbps with TCP Traffic

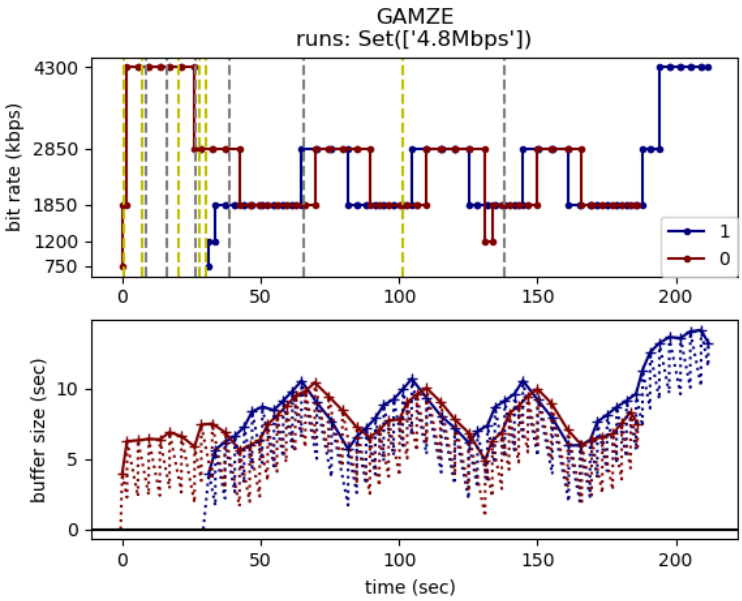

Figure D.1: `GAMZE10` - 2 streams with 4.8 Mbps.

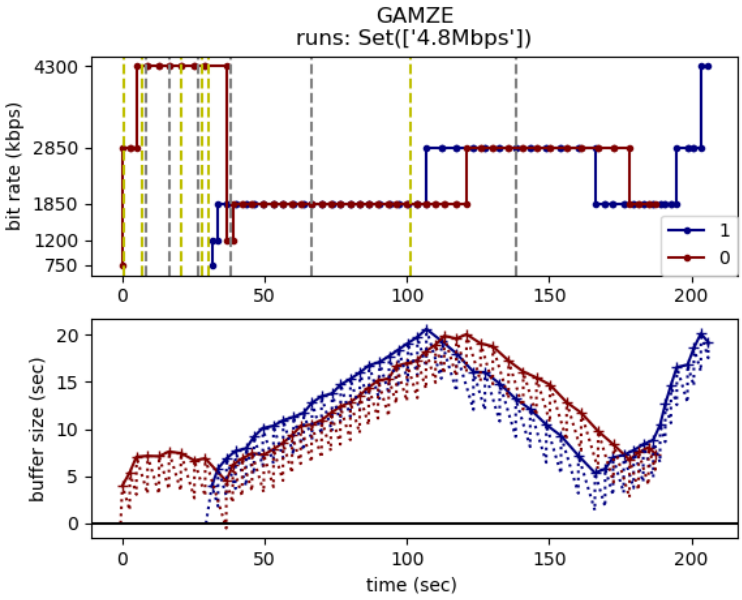

Figure D.2: `GAMZE20` - 2 streams with 4.8 Mbps.

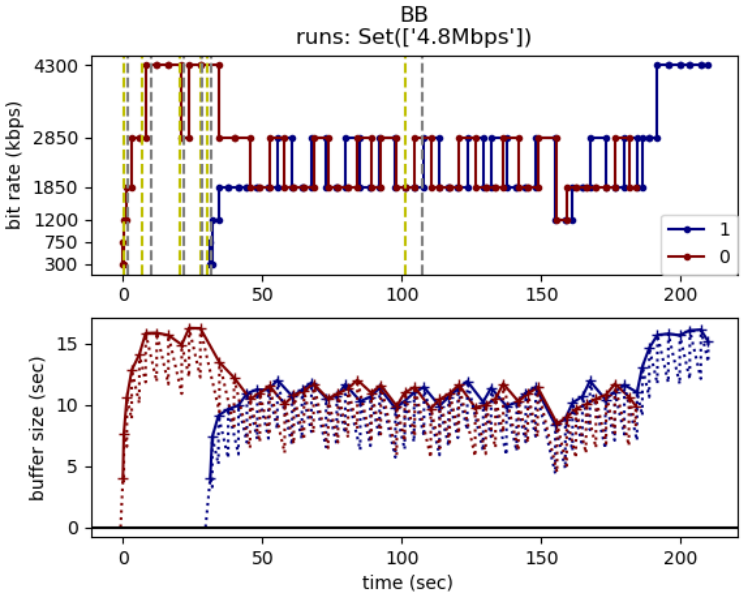

Figure D.3: BB - 2 streams with 4.8 Mbps.

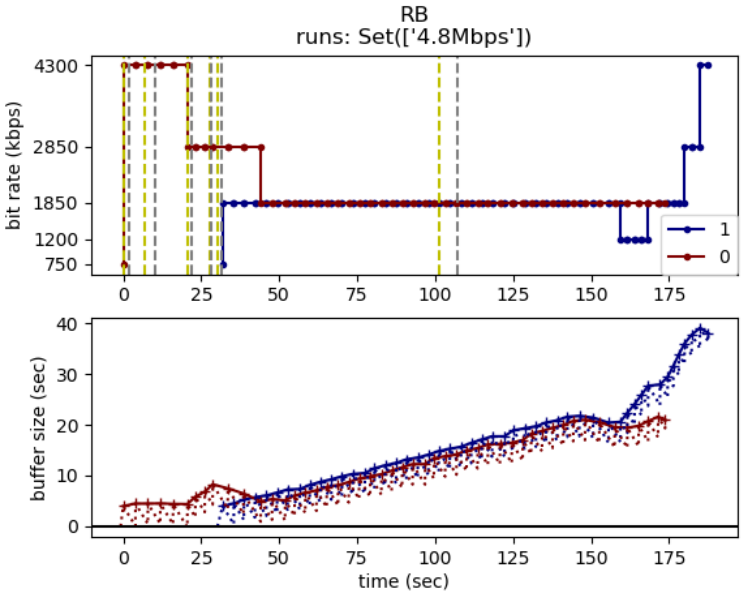

Figure D.4: RB - 2 streams with 4.8 Mbps.

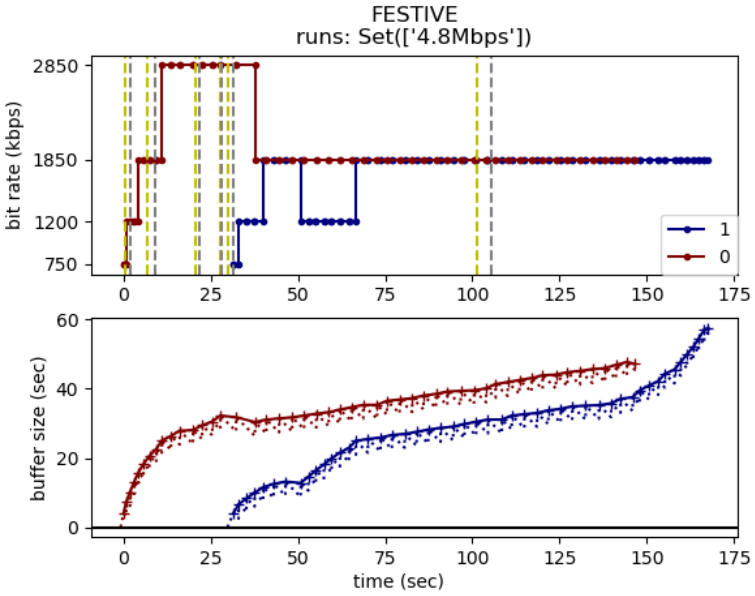

Figure D.5: `FESTIVE` - 2 streams with 4.8 Mbps.

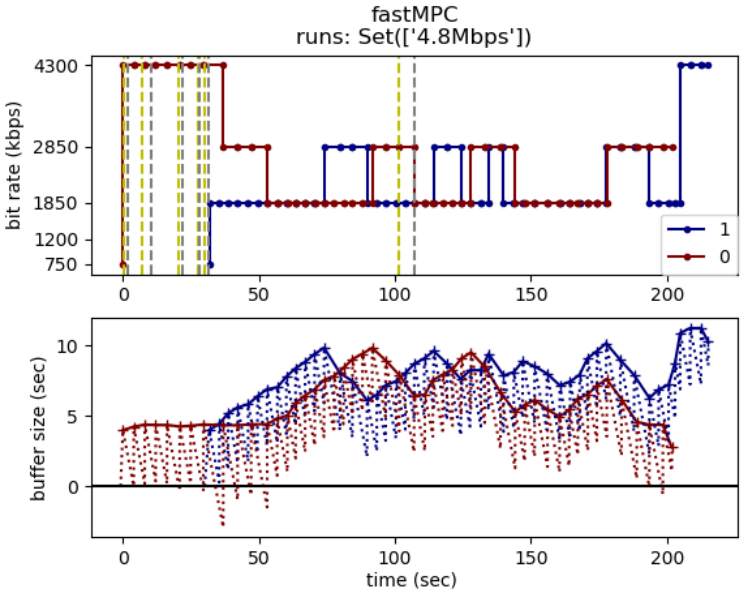

Figure D.6: `fastMPC` - 2 streams with 4.8 Mbps.

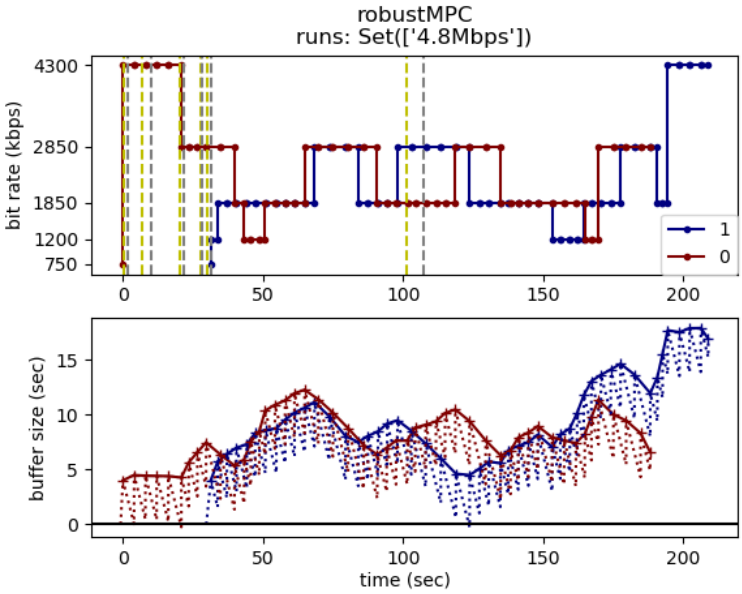

Figure D.7: `robustMPC` - 2 streams with 4.8 Mbps.

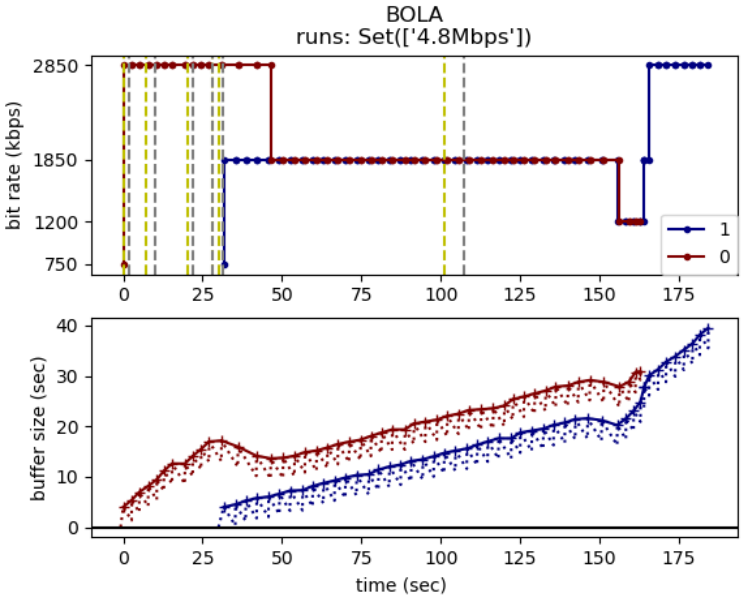

Figure D.8: `BOLA` - 2 streams with 4.8 Mbps.

E. 6 Streams - 10 Mbps

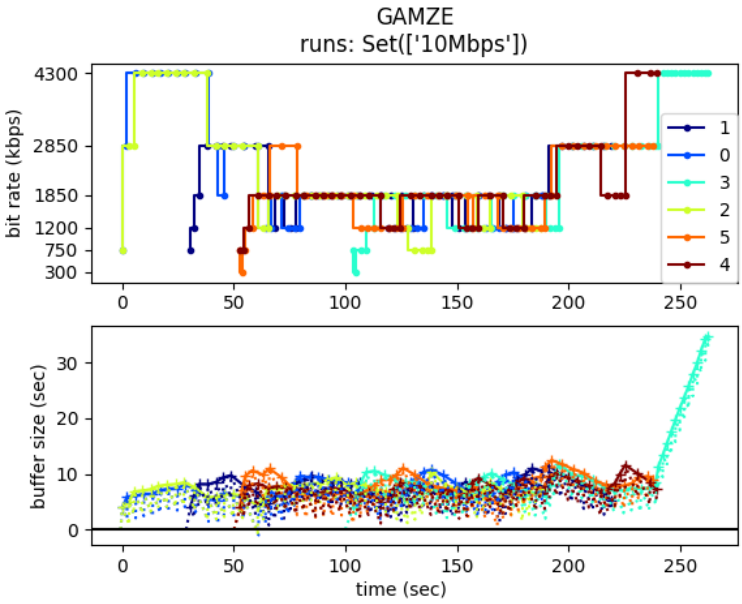

Figure E.1: `GAMZE10` - 6 streams with 10 Mbps.

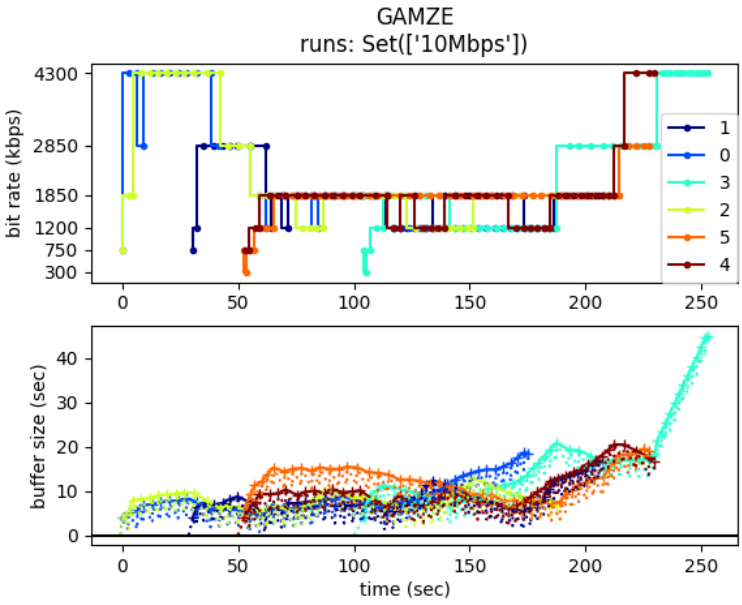

Figure E.2: `GAMZE20` - 6 streams with 10 Mbps.

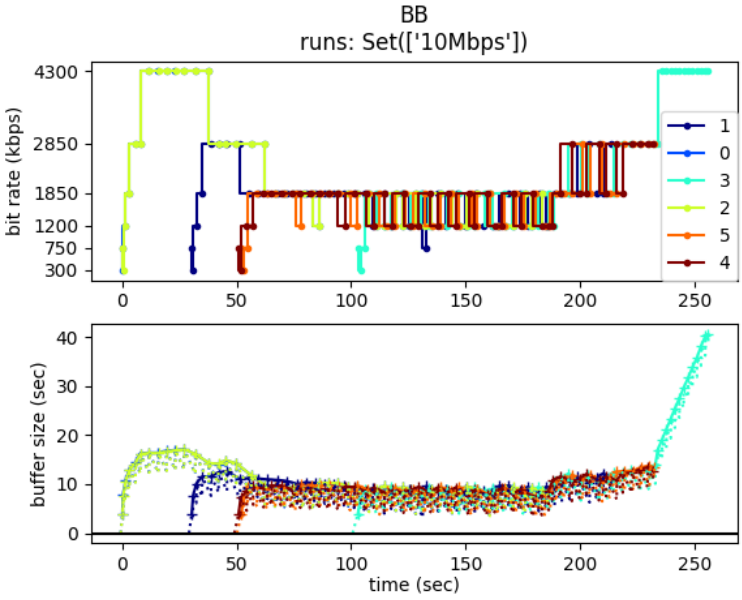

Figure E.3: BB - 6 streams with 10 Mbps.

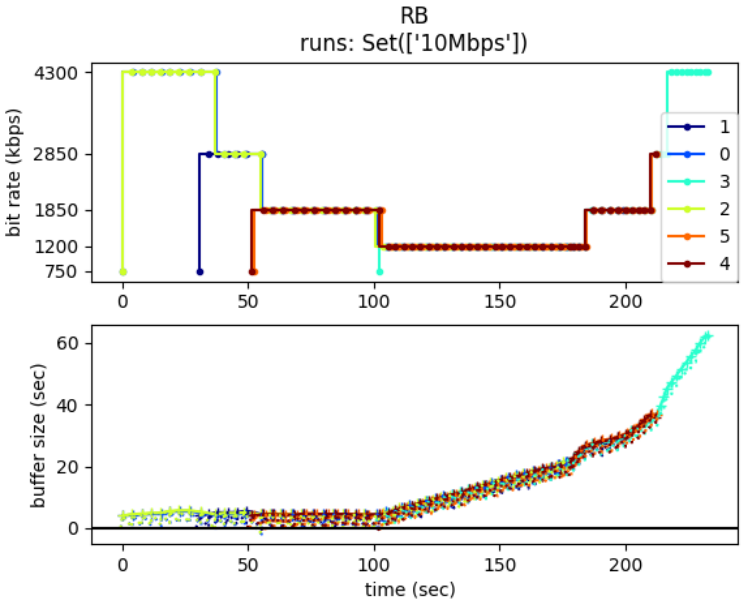

Figure E.4: RB - 6 streams with 10 Mbps.

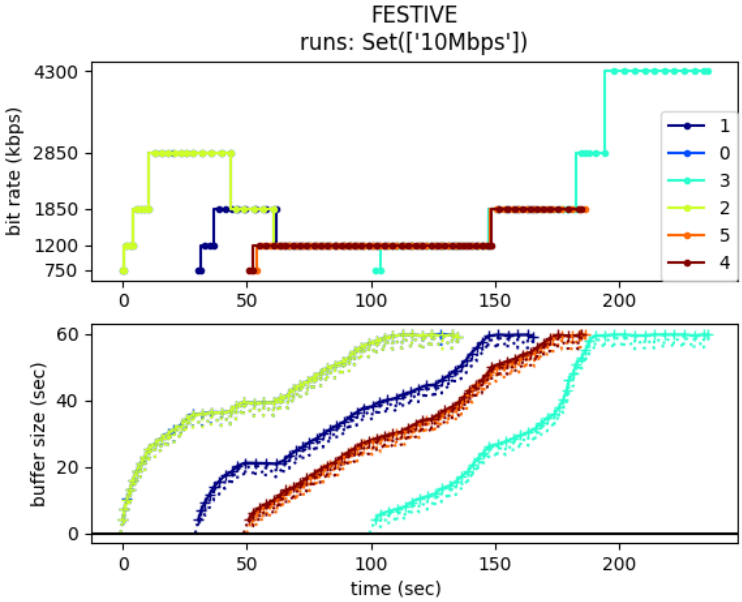

Figure E.5: `FESTIVE` - 6 streams with 10 Mbps.

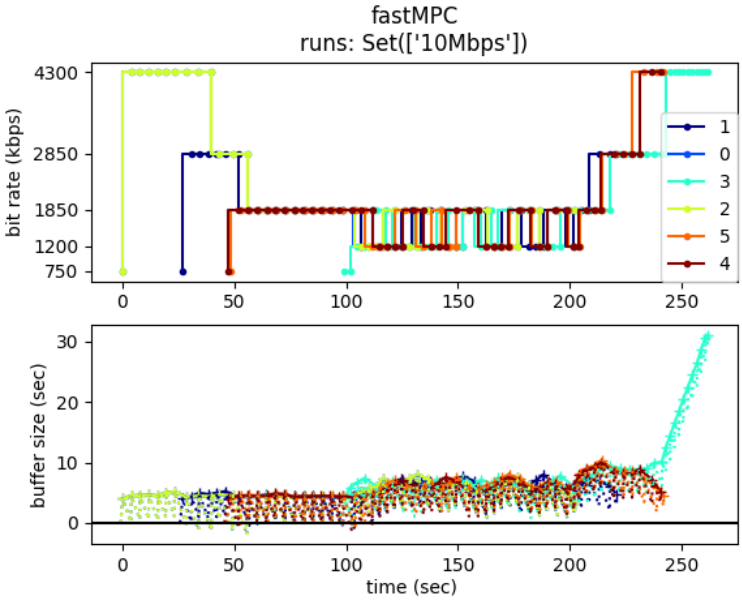

Figure E.6: `fastMPC` - 6 streams with 10 Mbps.

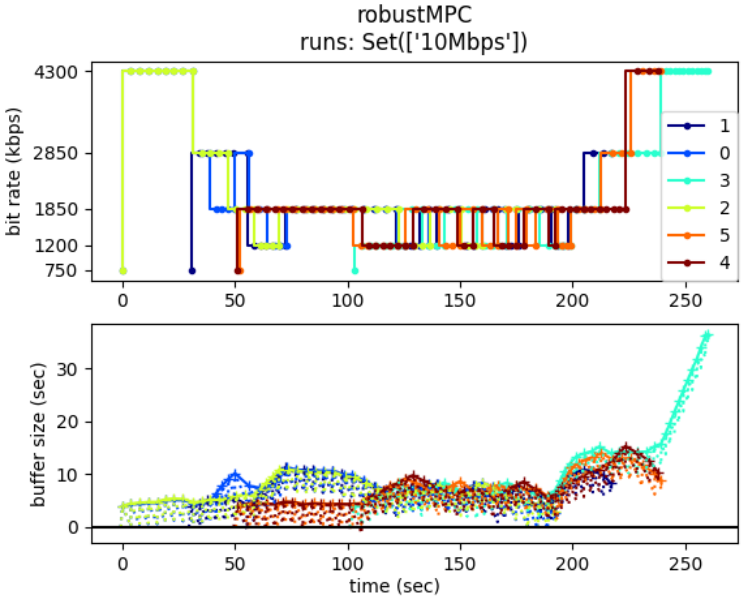

Figure E.7: `robustMPC` - 6 streams with 10 Mbps.

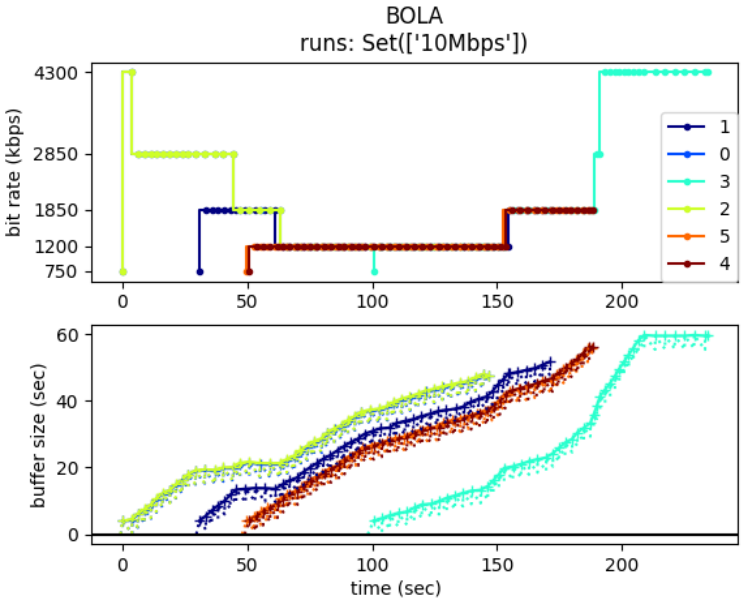

Figure E.8: `BOLA` - 6 streams with 10 Mbps.