Performance Analysis of HTTP based Dynamic Adaptive Streaming (DASH) over Google's QUIC

Submitted in partial fulfillment of
the requirements for the degree of

Bachelor of Technology
in
Computer Science and Engineering

Submitted by
Siva Kesava Reddy K(13CS10048)

Under the supervision of
Dr. Sandip Chakraborty

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

April 2017

CERTIFICATE

This is to certify that the project report titled "**Performance Analysis of HTTP based Dynamic Adaptive Streaming(DASH) over Google's QUIC**" submitted by **Siva Kesava Reddy K (13CS10048)** to the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, in partial fulfillment of the requirement for the award of degree of Bachelor of Technology (Hons) is a record of bonafide research work carried out by him under my supervision and guidance. The project report has fulfilled all the requirements as per the regulations of the institute and, in my opinion, has reached the standard needed for submission

Dr. Sandip Chakraborty
Assistant Professor
Dept. of Computer Science and Engineering,
Indian Institute of Technology,
Kharagpur -721 302, India

Date:

ACKNOWLEDGEMENTS

I am deeply grateful to my supervisor **Dr. Sandip Chakraborty** who gave me the opportunity to work on this project. I am thankful for his aspiring guidance, invaluably constructive friendly advice during the course of the project. I am sincerely grateful to him for sharing his truthful and illuminating views on a number of issues related to the project. I owe a lot to my teachers in the Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, who have instilled in me the scientific spirit of enquiry, experimentation, observation and inference, without which I would not have been able to produce this work. Also, I am thankful to the PhD scholars involved in this project who constantly motivated me to overcome all the challenges and helped me whenever I faced any issues. I hope I will be able to rise up to the high standards set by my predecessors in this department.

Siva Kesava Reddy K
13CS10048

Date:

Abstract

This project investigates whether “Dynamic Adaptive Streaming over HTTP” (DASH) performs better over “Quick UDP Internet Connections”(QUIC) protocol when compared with the existing TCP protocol. The investigation of DASH over QUIC is done with the help of YouTube since it can employ both DASH and QUIC simultaneously. DASH has become quite popular in the last few years for adaptive video streaming over networks which are time-varying and where video quality is tuned based on the available link bandwidth. QUIC is a secure transport protocol developed by Google and representing one of the most promising solutions to decreasing latency while intending to provide security properties similar with TLS. In this project we have collected YouTube HTTP request/response messages while playing videos from YouTube under controlled link environment with QUIC protocol enabled as well as with QUIC protocol disabled so that TCP protocol is used for the same video. We have collected data over a set of 175 videos under controlled (link bandwidth is adjusted manually and monitored) environments and from this data set we observe that QUIC has a greater tendency than TCP to switch over to higher resolutions but at the cost of more data wastage. We find that DASH performs better over QUIC than TCP in terms of video quality since QUIC is able to render a better or same quality video as of TCP at any instance of time under realistic bandwidth limited setting.

Contents

List of Figures

Chapter 1

Introduction

As we know, adaptive bitrate streaming has become the standard for delivering video content online to multiple devices. This type of delivery is a combination of server and client software that detects a clients bandwidth capacity and adjusts the quality of the video stream between multiple bitrates and/or resolutions. The adaptive bitrate video experience is superior to delivering a static video file at a single bitrate, because the video stream can be switched midstream to be as good or bad as the clients available network speed (as opposed to the buffering or interruption in playback that can happen when clients network speed cant support the quality of video). Because it uses the standard HTTP port, the lack of firewalls, special proxies or caches, and its cost efficiency have increased its popularity and use. There are three main protocols for this type of delivery- HTTP Live Streaming, Microsoft Smooth Streaming, and HTTP Dynamic Streaming. Each protocol uses different methods and formats, and therefore, to receive the content from each server, a device must support each protocol. A standard for HTTP streaming of multimedia content would allow a standard-based client to stream content from any standard-based server, thereby enabling consistent playback and unification of servers and clients of different vendors.[1] In response to the scattered landscape, the research community has developed the specifications for *dynamic adaptive streaming over HTTP* (DASH), which has later on been standardized by DASH Industry Forum.[2]

[1] `https://www.encoding.com/mpeg-dash/`

[2] `http://dashif.org/`

1.1 Dynamic adaptive streaming over HTTP(DASH)

DASH provides a standard solution for the efficient and easy streaming of multimedia using existing available HTTP infrastructure (particularly servers and CDNs, but also proxies, caches, etc.). DASH specification provides a full set of HTTP adaptive bitrate streaming features for delivering multimedia over Internet.The features include the following:[3]

- Frame-synchronized adaptive bitrate switching.
- Codec-agnostic.
- DRM-agnostic. It specifically supports the Common Encryption (CENC) system.
- Evolving support for closed-captions and subtitles.
- Support for multiple file container formats.
- Support for multiple manifest formats for VOD and live streaming.
- Fast-growing industry support.

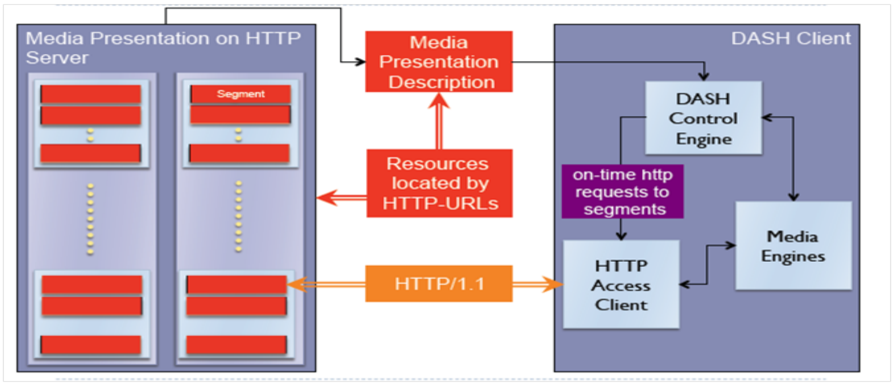

Media Presentation on HTTP Server
Segment
Media Presentation Description
Resources located by HTTP-URLs
HTTP/1.1
DASH Client
DASH Control Engine
on-time http requests to segments
Media Engines
HTTP Access Client

Figure 1.1: DASH conceptual Architecture.

Source:Thomas Stockhammer, Qualcomm, DASH Design Principles and Standards , Presentation at MMSys 2011

A DASH server provides client players with a list of the available media chunk URLs in a *Media Presentation Description (MPD)* manifest file.

[3]`https://www.wowza.com/forums/content.php?508-How-to-do-MPEG-DASH-streaming`

1.2 Quick UDP Internet Connections(QUIC): A 10,000 Feet View

QUIC (Quick UDP Internet Connections) is a new transport protocol for the internet, developed by Google. QUIC solves a number of transport-layer and application-layer problems experienced by modern web applications, while requiring little or no change from application writers. QUIC is very similar to TCP+TLS+HTTP2, but implemented on top of UDP. Having QUIC as a self-contained protocol allows innovations which are not possible with existing protocols as they are hampered by legacy clients and middle-boxes.[4]

Key advantages of QUIC over TCP+TLS+HTTP2 include:

- Connection establishment latency
- Improved congestion control
- Multiplexing without head-of-line blocking
- Forward error correction
- Connection migration

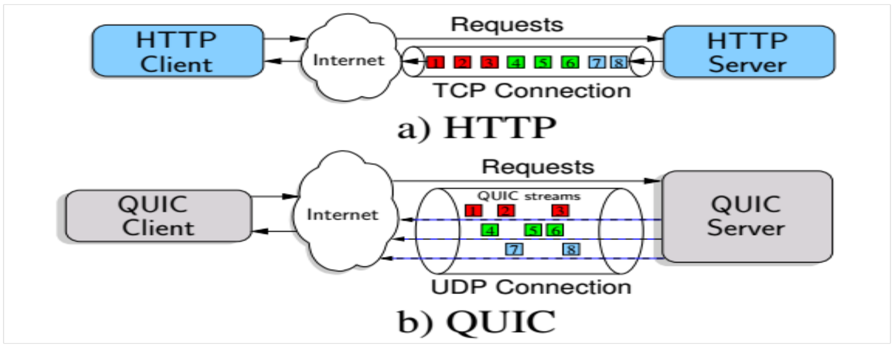


Figure 1.2: Multiplexing

Source:Gaetano Carlucci, Luca De Cicco and Saverio Mascolo. HTTP over UDP: an Ex-perimental Investigation of QUIC.ACM SAC15, April 13 - 17 2015, Salamanca,Spain.

[4]`https://www.chromium.org/quic`

1.3 Previous Work

Most of the work done on QUIC was how QUIC performs with respect to TCP and SPDY in terms of page load time. In “How quick is QUIC?” [1] they studied about the performance of QUIC, SPDY and HTTP particularly about how they affect page load time. They found that none of these protocols is clearly better than the other two and the actual network conditions determine which protocol performs the best. Similarly in “ HTTP over UDP: an Experimental Investigation of QUIC”[2] they found out that QUIC reduces the overall page retrieval time with respect to HTTP in case of a channel without induced random losses and outperforms SPDY in the case of a lossy channel. The FEC module, when enabled, worsens the performance of QUIC. Almost all the research was concentrated on how QUIC performs in terms of page load time, the number of RTT’s required and how it deals when there is packet loss so basically they were concerned about object transfers or static elements transfer and there were no major publications on how QUIC performs if it used for video streaming services like YouTube.

1.4 Motivation and Objective

In **Chromium blog** on Friday, April 17, 2015 Google has released some performance update about QUIC. In there words :

“ Results so far are positive, with the data showing that QUIC provides a real performance improvement over TCP thanks to QUIC’s lower-latency connection establishment, improved congestion control, and better loss recovery. For latency-sensitive services like web search, the largest gains come from zero-round-trip connection establishment. The standard way to do secure web browsing involves communicating over TCP + TLS, which requires 2 to 3 round trips with a server to establish a secure connection before the browser can request the actual web page. QUIC is designed so that if a client has talked to a given server before, it can can start sending data without any round trips, which makes web pages load faster. The data shows that 75% percent of connections can take advantage of QUICs zero-round-trip feature. Even on a well-optimized site like Google Search, where connections are often pre-established, we still see a 3% improvement in mean page load time with QUIC. Another substantial gain for QUIC is improved congestion control and loss recovery. Packet sequence numbers are never reused when retransmitting a packet. This avoids ambiguity about which packets have been received and avoids dreaded retransmission timeouts. **As a result, QUIC outshines TCP under poor network conditions, shaving a full second off the Google Search page load time for the slowest 1% of connections. These benefits are even more apparent for video services like YouTube. Users report 30% fewer rebuffers when watching videos over QUIC. This means less time spent staring at the spinner and more time watching videos.** ”[5]

In this context we are trying to know whether QUIC has any additional advantages for YouTube like improved video quality rendered over time besides fewer rebuffers. Is there any price that we need to pay for better video quality or there is no such trade-off? In this project we mainly compare the performance in terms of video quality rendered over time by both the protocols and check whether our hypothesis that QUIC will perform better over DASH when compared to TCP is valid or not and is there any price for it.

[5]`https://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html`

Chapter 2

Implementation

To understand the performance of DASH over QUIC and TCP we need to have certain parameters that affect the dynamic functionality of YouTube. These parameters can be found out from HAR files. In developer tools, there is a network monitor where the browsers dump information about all the requests made by the current page - it includes HTTP-request, HTTP-response, request/response time, link speed etc. This entire information can be saved as HTTP Archive (HAR) files. HAR stores the information in JavaScript Object Notation (JSON) file format [6]. First we will describe the challenges faced in the process to download the HAR files and then describe the experimental set-up.

2.1 Challenges

How to automate the HAR download process from Chrome?
A tool (AutoHarExporter) was developed using `selenium` to capture the HAR from Mozilla Firefox browser with the help of `har_export_trigger` (version 0.5.0-beta) Firefox plug-in . This tool automatically opens a Firefox browser, loads a YouTube video, waits for the video to finish and finally saves the HAR and other information to the disk. We need to do the process of HAR collection only in Chrome since QUIC is deployed in Chrome alone. But Chrome does not have any such plug-ins which with the help of `selenium` can automate the process of downloading HAR for a YouTube video. In an effort to automate the process we took the help of `BrowserMob Proxy`, a free utility that can capture performance data for web apps (via the HAR format), as

well as manipulate browser behavior and traffic, such as whitelisting and blacklisting content, simulating network traffic and latency, and rewriting HTTP requests and responses. This BrowserMob Proxy along with selenium chrome driver were expected to replicate the functionality that was observed in Firefox with the help of plug-in. `Selenium Chromedriver` allows you to use a programming language of your choice in designing your tests. We implemented the HAR downloading automation for Chrome in Python.

How to handle the network proxy?
Whenever we tried to download the HAR through the python script that we had developed we always encountered an error, "`ValueError: No JSON object could be decoded`". We found that browsermob-proxy will not work with our network proxy settings unless we explicitly encode the settings in the code. We found this explicit coding in Java but not in Python and then we realized that we require a connection without proxy for QUIC to be enabled since our proxy servers will not be able to allow QUIC packets to pass through.So we started using a connection without proxy and were able to run the script without any error.

Is QUIC really used as transport protocol during video playback?
Even though we can enable QUIC in chrome using `chrome://flags`, we can not be sure whether QUIC is used unless we capture data packets using `Wireshark` and verify them. When we opened the chrome browser manually and load a video we were able to see QUIC packets in wireshark but when we used the python script to open the browser and download HAR, QUIC falls back to TCP. In order to figure out whether the problem is with selenium chrome driver or with browsermob-proxy, we use selenium chrome driver alone to load a video and observed the packets in the wireshark. It was found that QUIC is being used as the underlying protocol to fetch the video data packets. So browsermob-proxy is the reason for QUIC not being used as the underlying protocol when chrome browser is opened through script. Selenium chrome driver requires an utility like browsermob-proxy to download HAR in chrome. So we scraped off this entire mechanism and searched for new alternatives that do not use a third-party proxy. We came across a GitHub repository(chrome-har-capturer)[1] that does the same thing as we wanted but with a minor problem. We will describe about it in the next section

[1]`https://github.com/cyrus-and/chrome-har-capturer`

2.2 Experimental Set-up

Using the GitHub repository we are able to load a web page and download the HAR after the page load is complete but for the YouTube video we want the HAR to be downloaded after the video has finished up playing. We contacted the repository owner and asked if there is any such provision and they said that there is no such provision but you can download HAR after a pre-specified delay. So we finally found out for each video its duration and added another 5 minutes to that in case buffering occurs as the pre-specified delay. After this we are able to download HAR though python script using Google Chrome Remote Interface[2] and chrome-har-capturer.

In order to simulate the realistic bandwidth limited setting we throttle the available link bandwidth. Although Chrome support throttling in network monitor, there is no suitable way to change it from script. Therefore we make use of the throttler developed with the Unix library `NetFilterQueue` [3]. It is a user-space library that provides an API to handle packets, which have been queued by the kernel packet filter, as per user requirement. Based on this library, traffic shaper was developed to control link bandwidth. However, we need to continuously monitor and ensure that the backbone network has sufficient bandwidth so that the overall link bandwidth is controlled only by the throttling procedure. We load a YouTube video, wait for the video to finish and then save the HAR and other information to the disk. During video playback, we also control network bandwidth by progressively increasing the bandwidth followed by a sudden decrease and the cycle repeats. The bandwidth levels used are from 64 Kbps to 1424 Kbps, in a step of 340Kbps. Each level of bandwidth is kept fixed for 220 seconds.

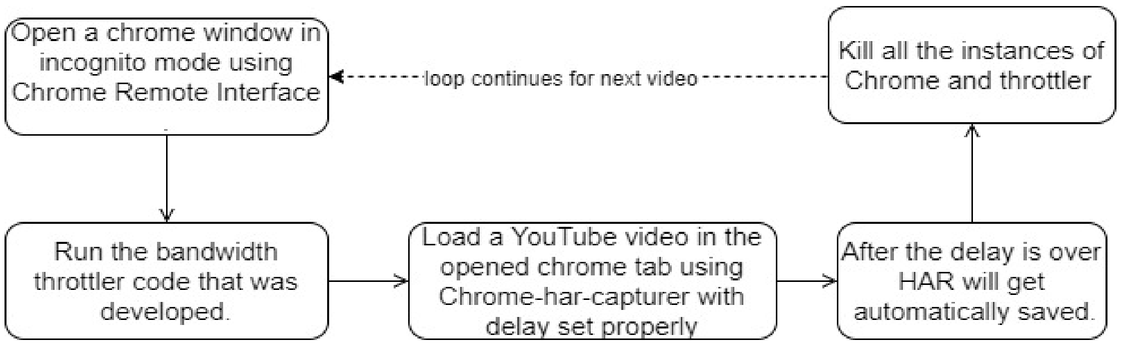

Open a chrome window in incognito mode using Chrome Remote Interface
loop continues for next video
Kill all the instances of Chrome and throttler
Run the bandwidth throttler code that was developed.
Load a YouTube video in the opened chrome tab using Chrome-har-capturer with delay set properly
After the delay is over HAR will get automatically saved.

Figure 2.1: Experimental Set-up

[2]`https://github.com/cyrus-and/chrome-remote-interface`

Chapter 3

Experimental Results

3.1 Extract Useful Information from the HTTP Headers

From the HAR traces, we observe that YouTube uses a video playback request to grab the media data from the server. URLs for these video playback requests contain 35 parameters and their values: *pl, dur, expire, sver, gir, pcm2cms, mime, itag, signature, ipbits, source, keepalive, mt, mv, ms, mm, mn, key, clen, requiressl, lmt, initcwndbps, id, upn, sparams, fexp, ip, cpn, alr, ratebypass, c, cver, range, rn, and rbuf.* By close inspection of these parameters, we observe that the HTTP requests and responses are forwarded separately for the audio channel and the video channel. The value of the parameter mime indicates whether the request is for audio channel or for video channel. Then, we figure out that the parameter itag actually indicates the video quality for which a DASH request is made. YouTube samples every video under different video quality levels based on its resolution, bit rate and encoding techniques used for sampling, and assigns a numeric level to every quality, which is the itag value. The mapping between a particular itag value and the corresponding video resolution, bit-rate and encoding parameters are available at [4].

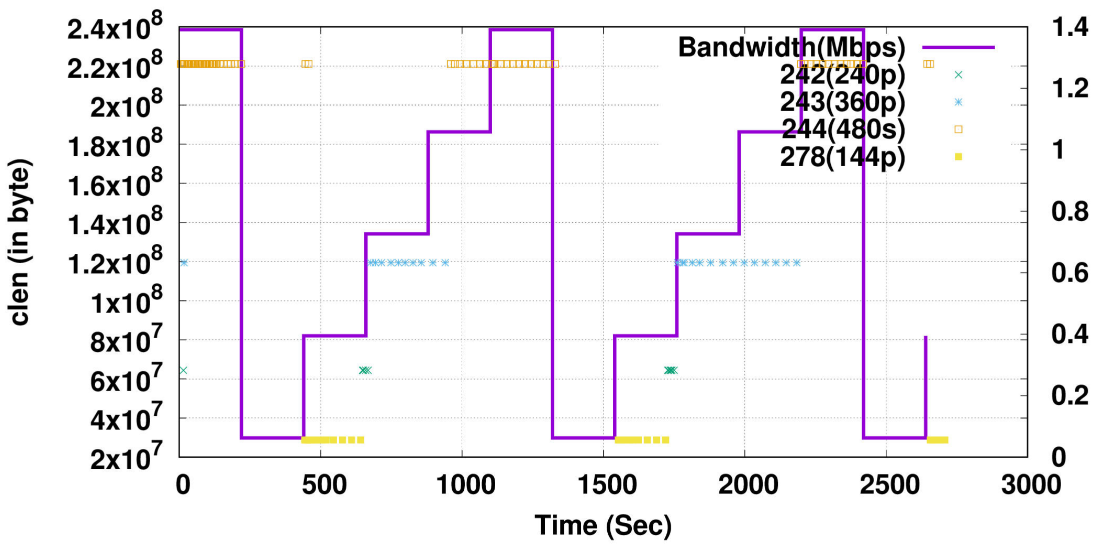
The behaviour of these parameters under different scenarios like "Multiple videos multiple sessions, Single video multiple sessions, Single video single session" was observed. When a value does not change over multiple videos multiple sessions, then it indicates that the parameter does not take part in video adaptation procedure, and it basically forwards some static information, like the device and the operating system related information. If the value of a parameter changes for multiple video multiple sessions, but does not change for single video multiple sessions, then we can say that it is a video specific parameter. If the value of a parameter changes for single video multiple sessions, but does not change for a single video single session even if the network quality changes, then we can say that it is session specific. The parameters that change only in this scenario when we change the link bandwidth, indicate that they possibly take part in the video adaptation process. From here, it was figured out in that *clen, dur, itag, lmt, mime, rbuf, rn, signature and range* are such parameters. However through close inspection, we find the parameters *mime* and *signature* relate to video channels, as we already discussed. Further the parameter *dur* denotes video duration, and it was observed that it changes only at microsecond order which is due to the change in video encoding technique. Consequently, we go for comparison of the other parameters between QUIC and TCP - *clen, itag, lmt, range, rbuf and rn.* It was observed that for the single video single session scenario, *rbuf, rn* and *range* change even for a single *itag*. On the other hand, parameters like *clen* change overall, but remain constant for a single *itag* value.

3.2 Comparison of Target Parameters of DASH over time between QUIC and TCP

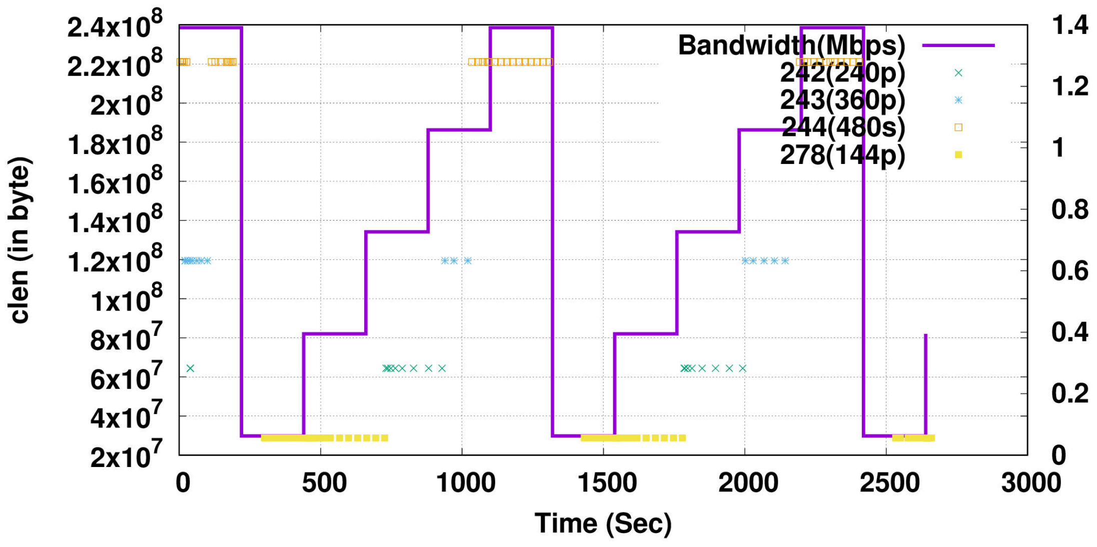
We will briefly state what particular information these parameters contain and then give the plot of each parameter with respect to time for QUIC and TCP for a YouTube video. Later we present the Cumulative Distribute Function graphs for the data obtained from the 175 YouTube videos for different parameters with respect to bandwidth and rbuf.

3.2.1 Parameter- *clen*

It has been identified that *clen* is the length of the video chunk for a particular *itag* value. The server creates a video chunk with *clen* amount of data for a particular *itag*. From the *clen* plots we observe that *clen* remains fixed for a particular *itag* value and does not depend on whether QUIC is used or TCP is used as the underlying protocol.



(a) QUIC

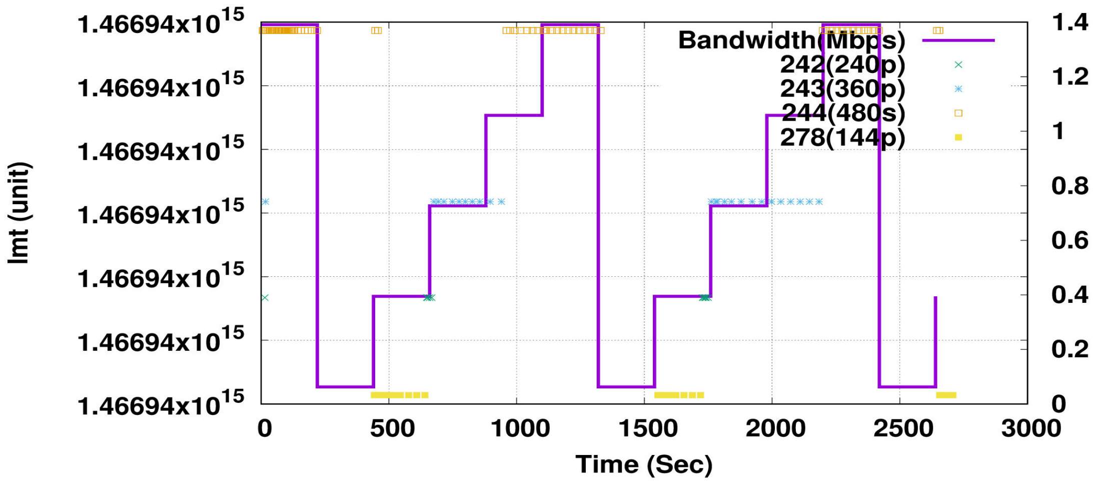


(b) TCP

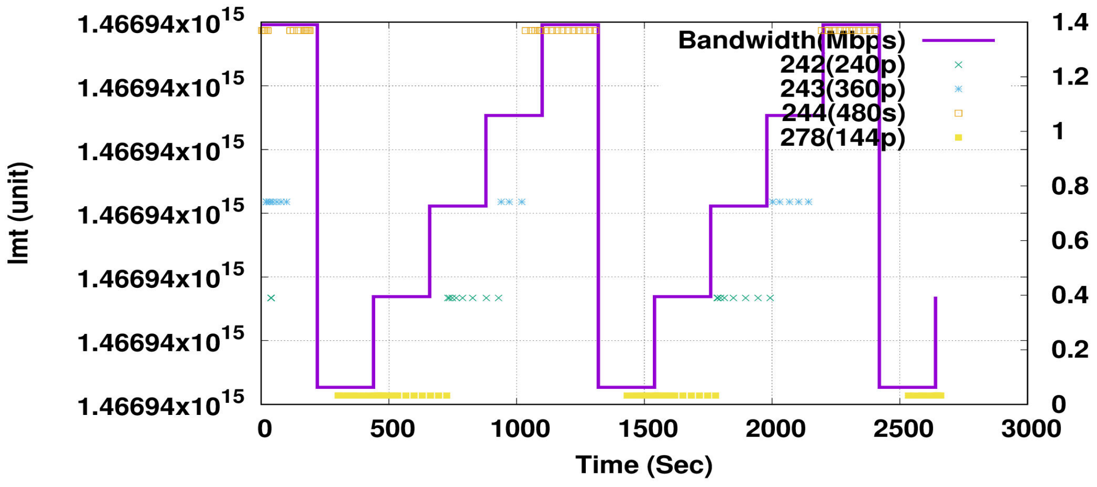
Figure 3.1: Plot for *clen* over time for a YouTube video of id <OJZgOOOE1zY>

3.2.2 Parameter- *lmt*

lmt also remain constant for a particular *itag* value of that video. It has been found out that *lmt* defines the time when the chunk was created at the YouTube server. However, this particular parameter does not help in video bit-rate adaptation, rather we presume that this is used to play the updated chunks at the clients. As YouTube downloads data from multiple servers [5], so this parameter is possibly used to avoid playing outdated video chunks. From the plots we can identify that *lmt* is independent of the transport layer protocol for a particular *itag*.



(a) QUIC

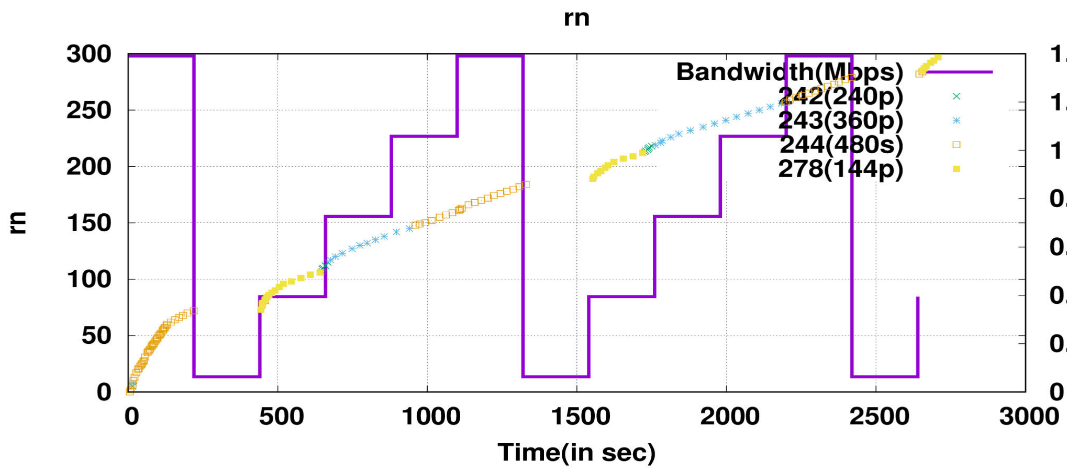


(b) TCP

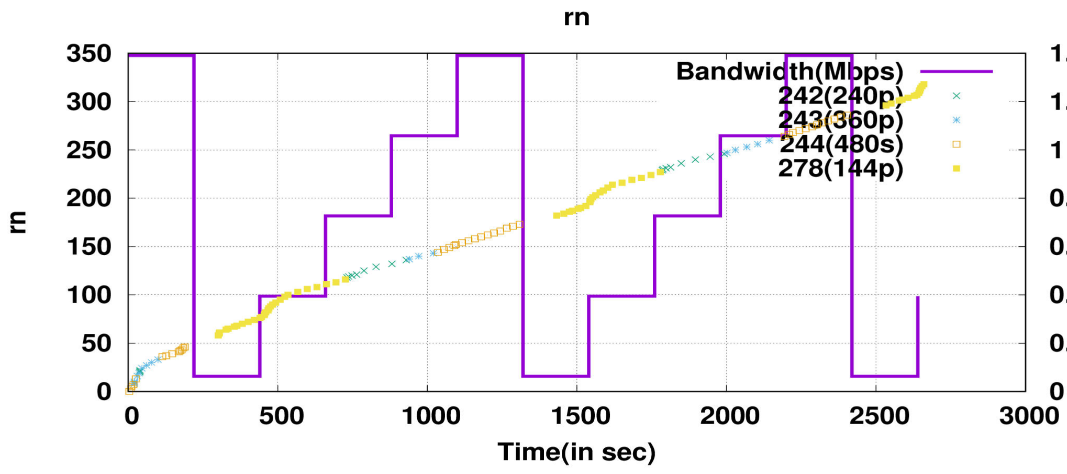
Figure 3.2: Plot for *lmt* over time for a YouTube video of id <OJZgOOOE1zY>

3.2.3 Parameter- *rn*

rn is non-decreasing for a session for any video it does not depend on a specific video ID or any other parameter. By observing the sequence of HTTP requests sent by the YouTube client to YouTube server, we can conclude that *rn* is the request number to uniquely identify a DASH video playback request. The plots shows that QUIC makes lesser number of DASH video playback requests when compared to TCP for the same video so it can be expected that QUIC will be able to render the data in lesser number of requests.



(a) QUIC



(b) TCP

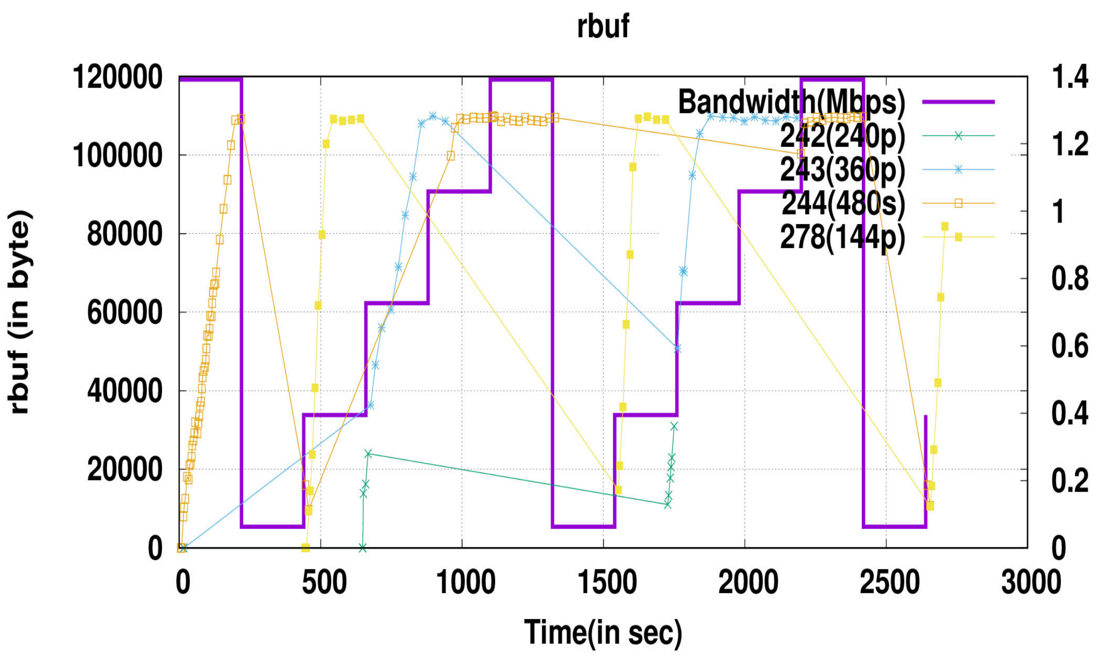
Figure 3.3: Plot for *rn* over time for a YouTube video of id <OJZgOOOE1zY>

3.2.4 Parameter- *rbuf*

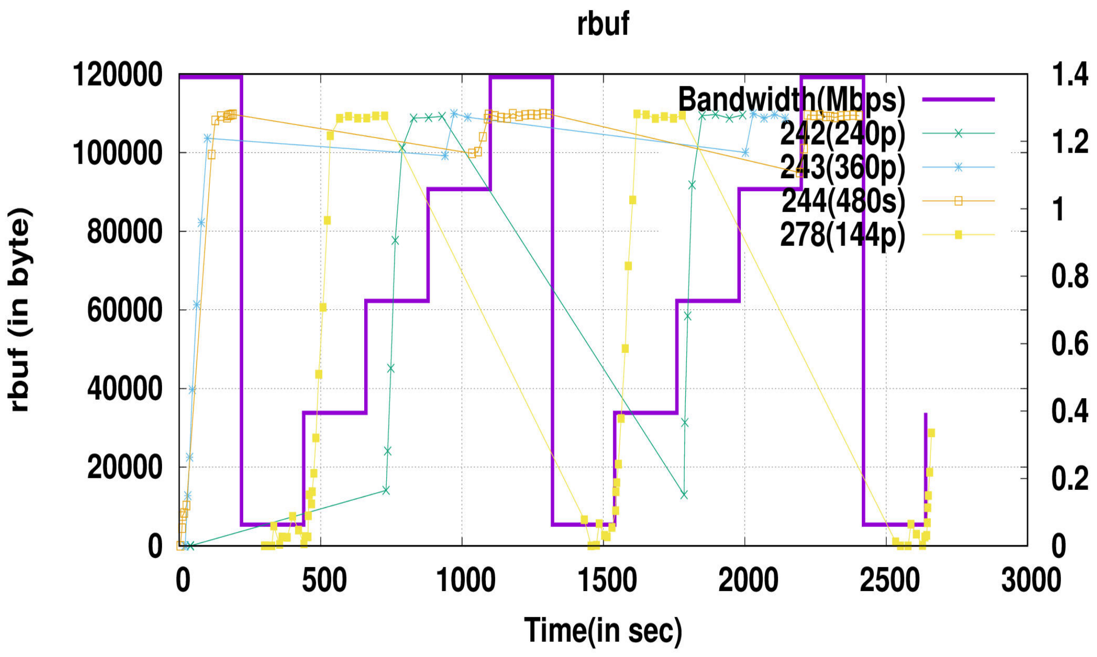
rbuf is the receive buffer at the client side. The figures indicates that rbuf grows as the YouTube client fetches more data from the server. However,with a close inspection of the HTTP request messages, we observe that the value of *rbuf* decreases if there is no request from the client to the server. Further, whenever *rbuf* starts decreasing, the client starts fetching data for a different *itag* value (as we observe near lower bandwidth ranges).

It can also be seen that whenever bandwidth increases, *rbuf* keeps on increasing up to a maximum threshold and then remains constant. Conversely, as the bandwidth drops, *rbuf* either remains constant or drops. This behavior of *rbuf* can be directly explained from the receive buffer evolution of a video playback client. As a general thumb-rule, the buffer size increases when the client fetches data from the server, and the buffer size drops as the video gets played. When the link bandwidth increases, the YouTube client has sufficient bandwidth to download data, and it fetches more data from the server than the data rendered for video playback. Therefore, buffer size keeps on growing as the data arrival rate (from the server) is more than the data service rate (for video rendering).

Nevertheless, when the link bandwidth drops, initially the data arrival rate becomes equal to the data service rate, resulting in constant buffer size. Further, as the bandwidth drops below a threshold, the YouTube server fails to fetch further data from the server at the current video quality, and at this point the buffer size starts dropping. During this period, YouTube client makes a transition to a lower video quality, and the buffer starts building up with the data of the lower video quality,as we observe near lower bandwidth ranges. We will describe more about this parameter in the next section along with the plot for time range of different qualities and plot for segment length downloaded over time.



(a) QUIC

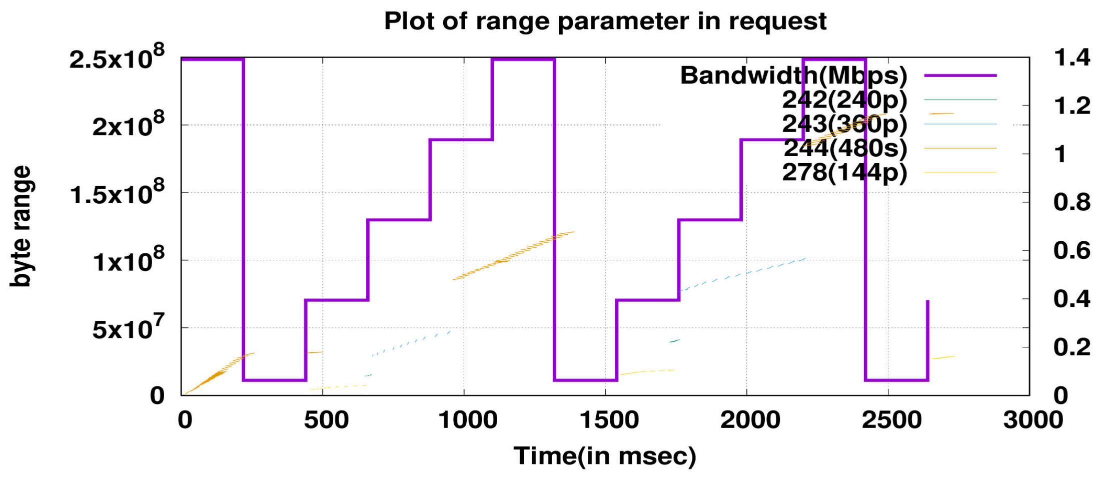


(b) TCP

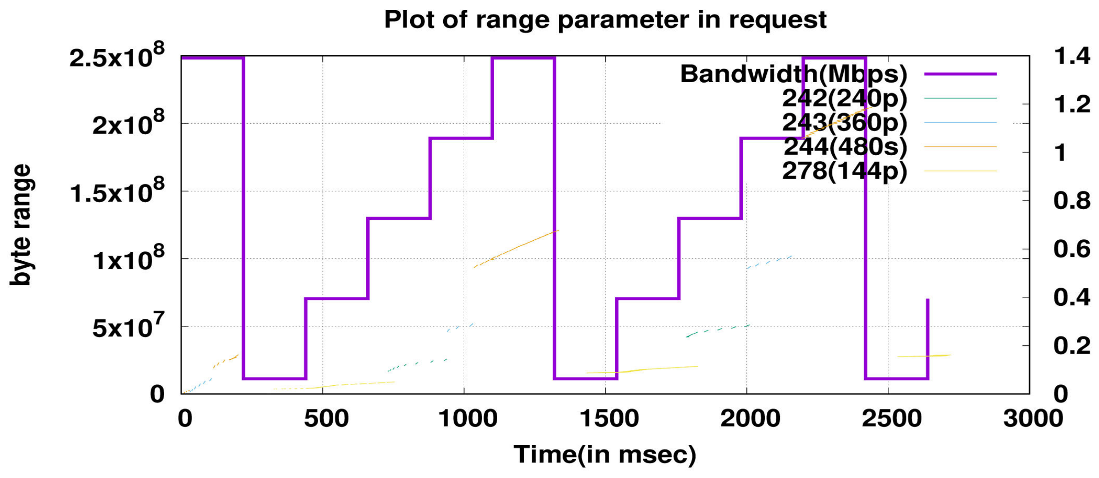
Figure 3.4: Plot for *rbuf* over time for a YouTube video of id <OJZgOOOE1zY>

3.2.5 Parameter- *range*

The *range* values have two integers separated by a dash (-). The first integer is always smaller than the second one, and therefore we can say that these two integers are the *start* and *end* of the *range* values. It behaves like byte range parameter in HTTP request header. It was concluded that range defines the byte range of the video for a *itag* value that the client requests from the server. YouTube client adaptively changes this parameter to increase or to decrease the video chunk size to download, based on network conditions.



(a) QUIC

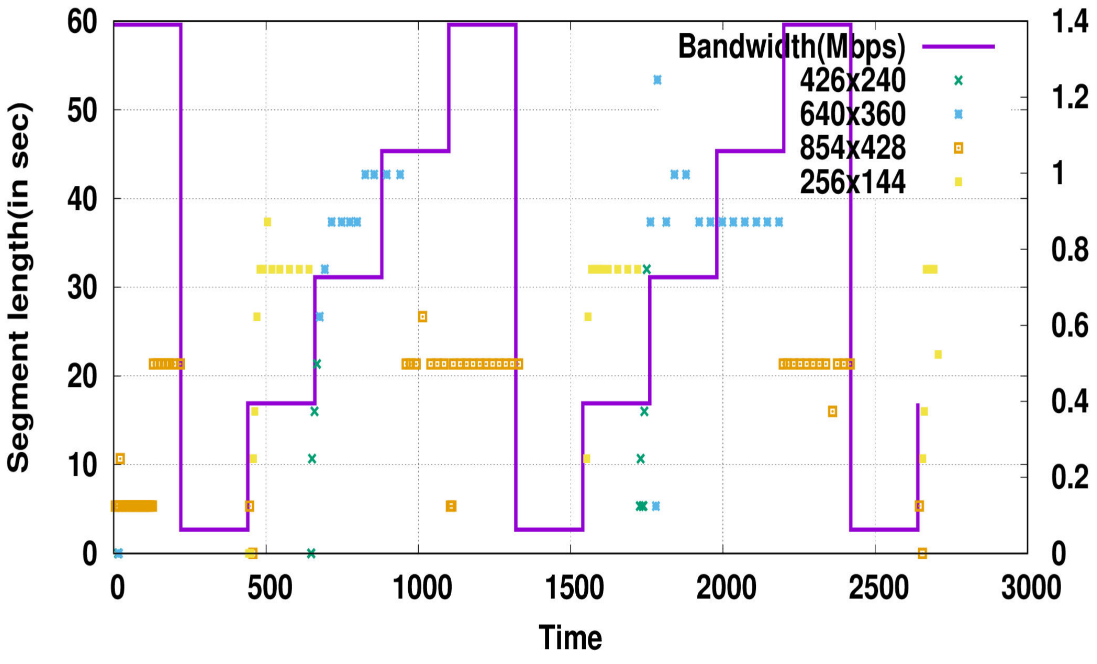


(b) TCP

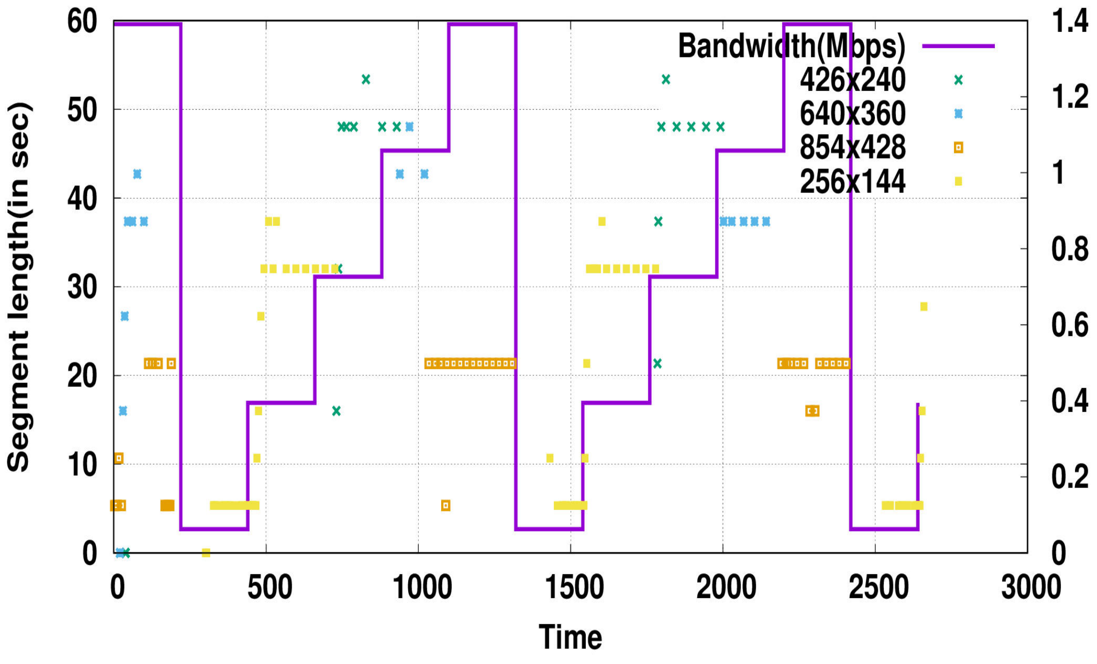
Figure 3.5: Plot for *range* over time for a YouTube video of id <OJZgOOOE1zY>

3.3 Comparison of Streaming data rate between QUIC and TCP

In this section we will examine whether the streaming data rate variation whenever the bandwidth changes, is similar in QUIC and TCP. To compare the behavior of streaming data rate adaptation, we plot how the requested video segment length (specified by the *range* parameter) per video playback request, changes with change in link bandwidth. For this, we convert the byte range mentioned in the video playback request to the equivalent video playback time, and find out the video segment length in terms of playback time. Whenever the link bandwidth increases, YouTube first increases the segment length of lower quality video and buffers maximum amount of video data. It then switches to the higher quality video but with smaller segment lengths. At this point, we observe an overlap between the segments of two different video qualities. It then progressively increases the segment length and repeats the procedure for the next higher quality level video if the link quality improves further (measured through the increase rate of *rbuf*). However, when the link quality drops, in a similar way, YouTube first starts requesting for same quality video chunks of smaller segments, and drops the segment length. If it still observes a drop in *rbuf* after reducing the segment length in the playback requests, then it switches to request for the next lower quality level video chunks of smaller segments. If the *rbuf* becomes stable, then only it again increases the segment length. This behaviour is similar to both QUIC and TCP over YouTube but from the plots we can observe that QUIC is able to switch to higher quality from a lower quality in a shorter interval of time when compared to TCP.



(a) QUIC



(b) TCP

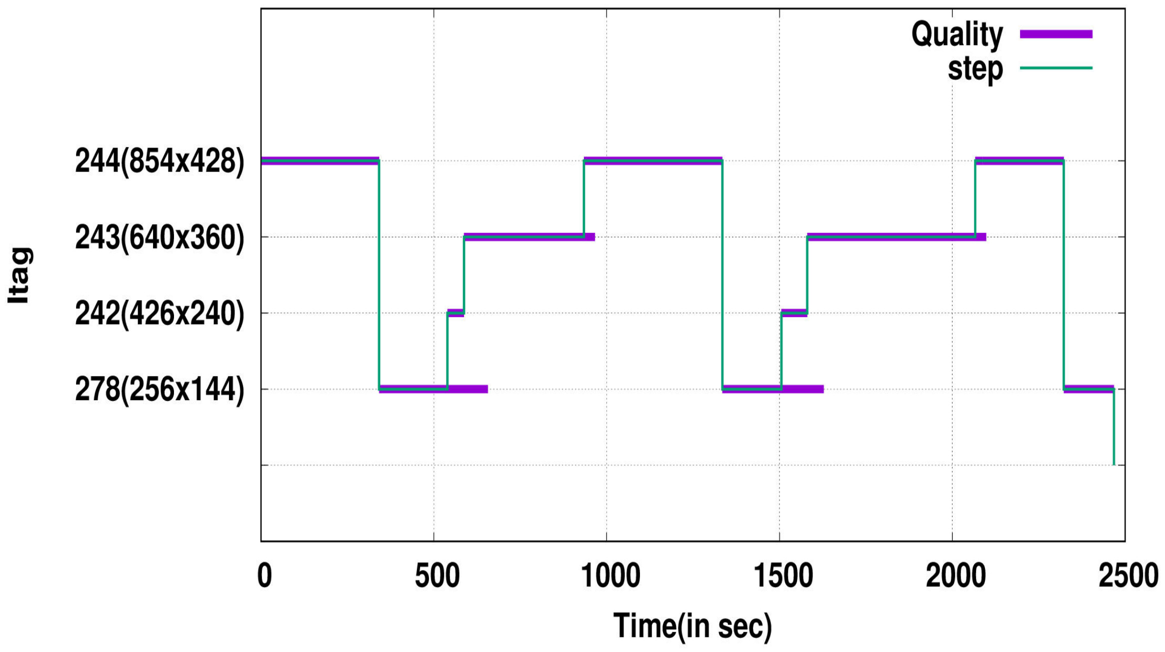
Figure 3.6: Plot for *Segment Length* over time for a YouTube video of id <OJZgOOOE1zY>

3.4 Comparison of Video Quality observed over time between QUIC and TCP

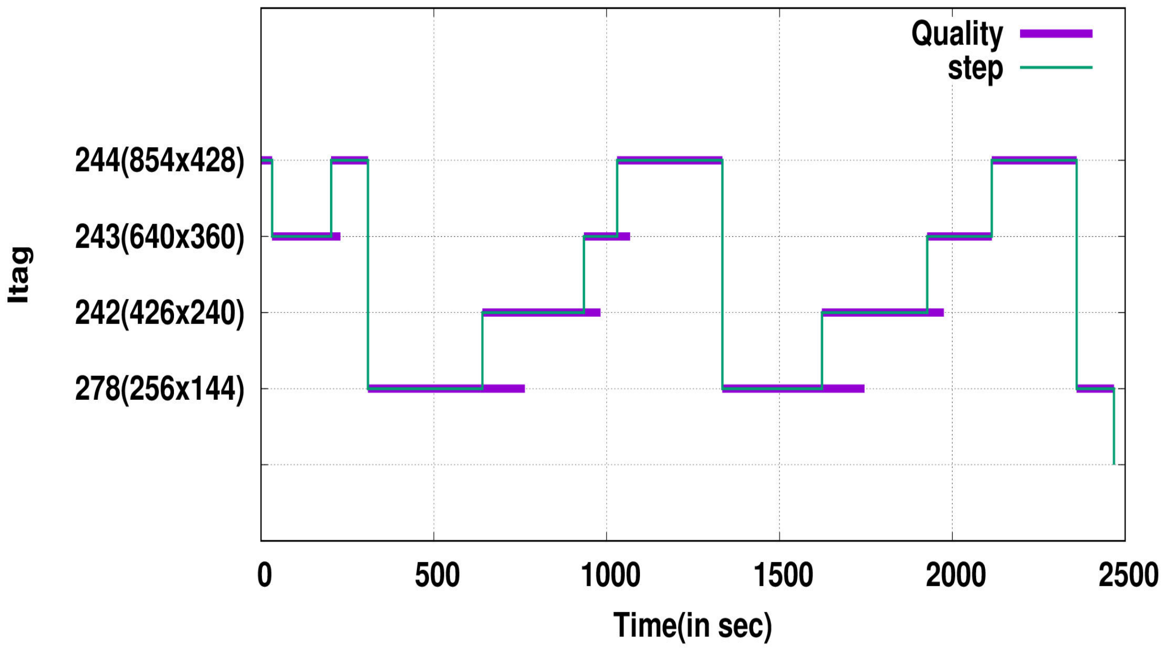
From Fig. 3.4, we observe that as link quality increases, *rbuf* also increases, and YouTube client progressively makes requests for higher *itag* values. Further, whenever the value of *rbuf* drops, YouTube client switches *itag* requests for a lower video quality. We already know that YouTube video quality adaptation algorithm is based on the clients observation of the change in receive buffer a sharp increase in receive buffer gives an indication for fetching data of higher video quality, whereas YouTube takes a conservative approach of requesting data for lower video quality whenever the client observes a sharp drop in receive buffer size.

From the earlier discussion,we know that *range* parameter gives the byte range of the video streaming data for which the client sent a request to the server. We first convert this range parameter to equivalent video segment length in terms of video playback time. This can be done by looking into the video file header that provides a mapping between the byte range and playback time. We use the Python package `python-ebml` to extract such information from the video files. The following figures plots the video segments (in terms of video playback time, as shown in Xaxis) and the corresponding itag values for which the client makes a request. YouTube takes an opportunistic approach for downloading higher quality video segments when the link quality improves, but takes a conservative approach when the link quality drops. In the opportunistic approach, it downloads the video chunks of both the video qualities in parallel, whenever it decides to switch from the lower quality to the higher quality. However, in the conservative approach, it directly sends the request for lower quality video when the link quality drops. That is why we notice an an overlap between the segments of lower quality and higher quality when the video quality improves.

When we observe the plot for a single video, it is evident that QUIC is able to maintain video quality that is higher or same as that of TCP in most cases. From these three plots we can expect that DASH will perform better when QUIC is employed as transport layer protocol rather than TCP in terms of video quality. In order to prove the above hypothesis we will show the Cumulative Distribute Function graphs collected over 175 videos in the next section.



(a) QUIC



(b) TCP

Figure 3.7: Plot for time range of different qualities for a YouTube video of id <OJZgOOOE1zY>

3.5 Cumulative Distribute Function Plots

We collected the HAR data for a total of 175 videos for QUIC and TCP. We wanted the bandwidth cycle to repeat at least once so we selected the videos that are on an average >30 minutes in duration. The following table summaries the statistics about the data.

Video Size	Number of Videos	Total Playback Duration
< 30 mins	13	5h 59m
30 – 40 mins	30	17h 08m
40 – 50 mins	90	66h 09m
50 – 60 mins	31	27h 56m
60 – 70 mins	6	6h 29m
70 – 80 mins	5	6h 11m

Table 3.1: Statistics OF YouTube Videos Used In The Experiments

3.5.1 CDF for *itag* with respect to various Bandwidth levels

For a particular bandwidth level we counted the number of requests made for each *itag* and from that we calculated the probability for an *itag* as the number of requests made for that itag divided by total number of requests made.

Itag	Resolution	Type
160	256x144	video/mp4
278	256x144	video/webm
133	426x240	video/mp4
242	426x240	video/webm
134	640x360	video/mp4
243	640x360	video/webm
135	854x428	video/mp4
244	854x428	video/webm

Table 3.2: Information about itags

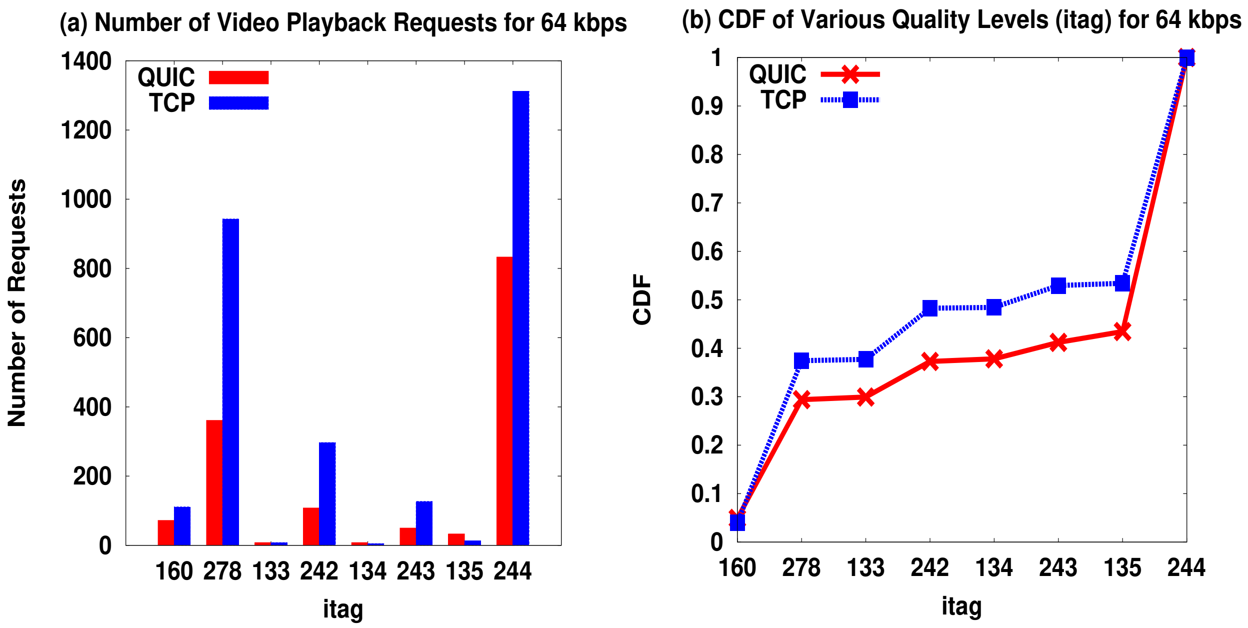


Figure 3.8: Number of requests and CDF of itag at 64 kbps

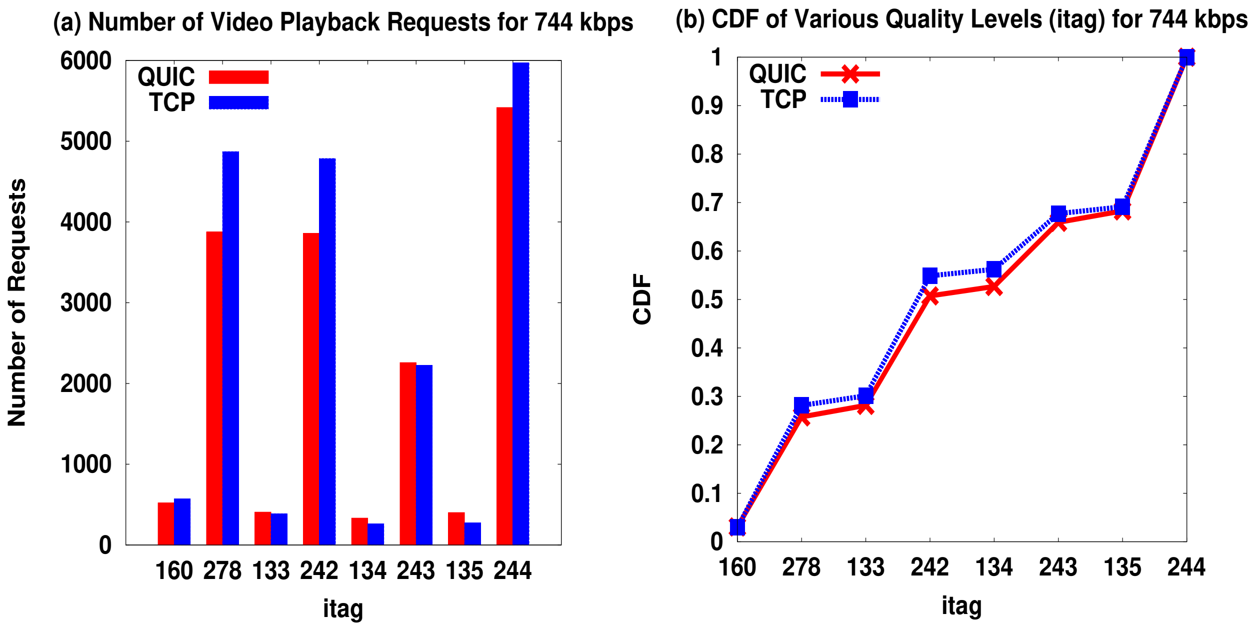


Figure 3.9: Number of requests and CDF of itag at 744 kbps

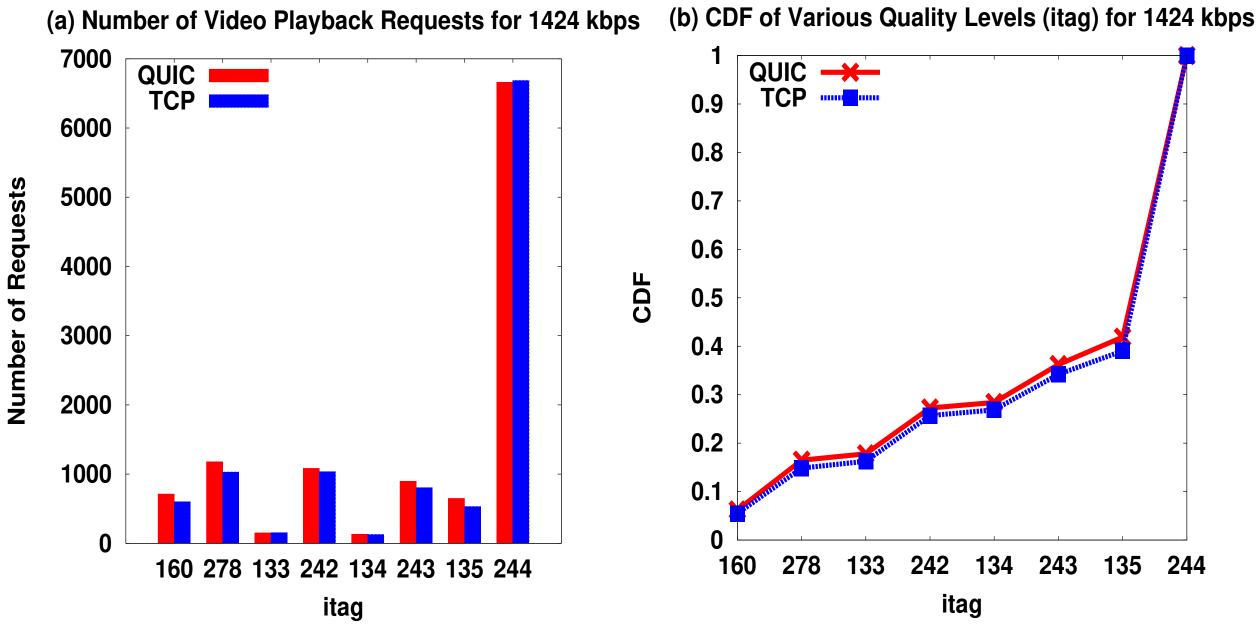


Figure 3.10: Number of requests and CDF of itag at 1424 kbps

The important observations that we can make from these CDF and number of requests plots for itag is that at lower bandwidth QUIC has higher tendency than TCP towards higher resolution but as the bandwidth increases both show similar tendencies. So at poor Internet connection speeds QUIC will provide the viewer with a better video quality than TCP. At lower speeds QUIC does not make as many requests to the server as TCP makes but at higher speeds both the protocols makes the same number of requests. This implies that QUIC is more aware of the network conditions than TCP. In the above plots when the bandwidth is at 64Kbps TCP has made almost twice the number of requests as QUIC made. This is because at 64Kbps when almost all the packets are dropped, TCP unable to quickly recognize the change in bandwidth makes the request for the same higher itag value and when they fail it makes requests for lower itags. For the total set of 175 videos the number of requests made by QUIC are lesser in number when compared to TCP which implies that QUIC requires less number of requests to serve the same or higher quality data.

3.5.2 CDF for *rbuf* with respect to various Bandwidth levels

rbuf unlike *itag* are a continuous domain so we won't be presenting the histogram. At higher bandwidth levels there is not much difference between the two protocols. At lower bandwidth, QUIC has greater tendency towards higher *rbuf* values when compared to TCP so the buffer is emptying at a slower rate when QUIC is used. This majorly implies that the chance for buffering is lower for QUIC when compared to TCP as buffering mainly occurs at poor Internet speeds supporting the statement made by Google that there were fewer rebuffers when QUIC is used instead of TCP.

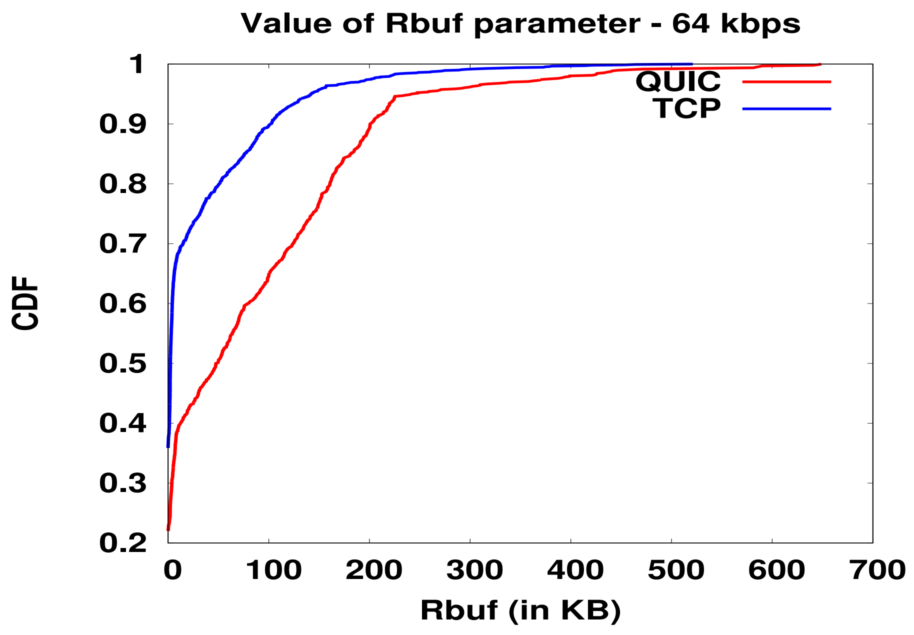


Figure 3.11: CDF of rbuf at 64 kbps

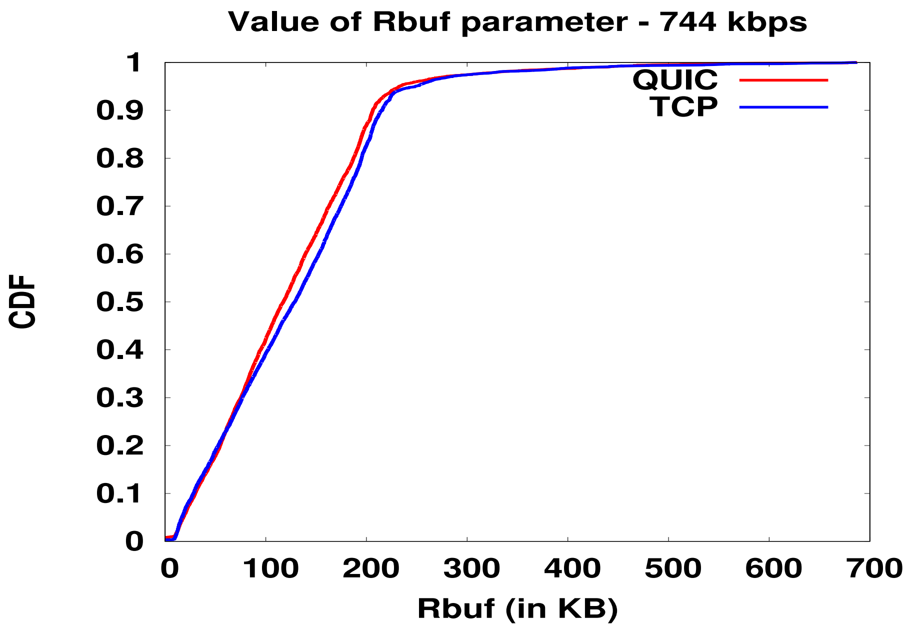


Figure 3.12: CDF of rbuf at 744 kbps

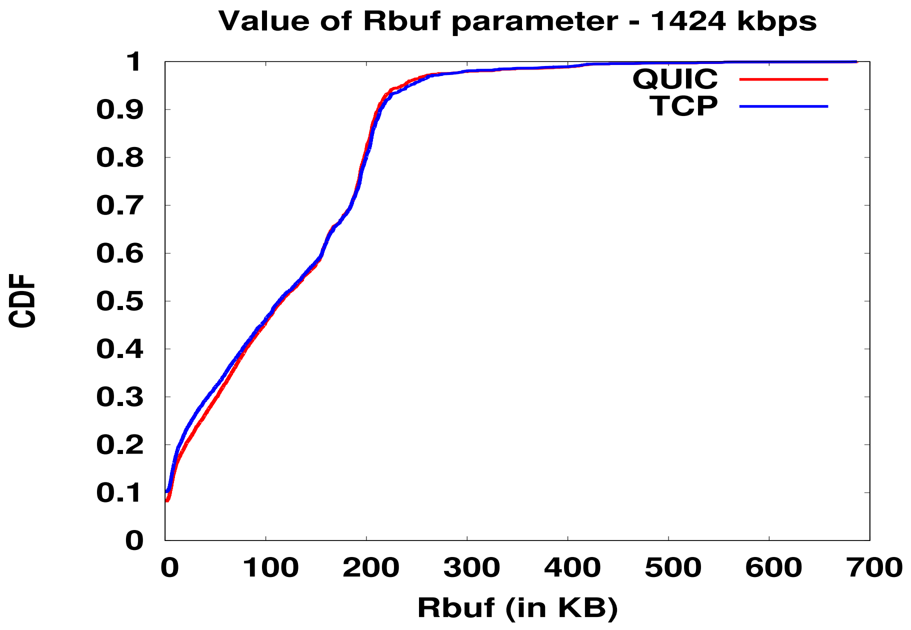


Figure 3.13: CDF of rbuf at 1424 kbps

3.5.3 CDF for *range* with respect to various Bandwidth levels

As described earlier *range* parameter consists of two values separated by a dash (-) and they define the byte range of the video for a itag value that the client requests from the server. We have taken the difference between these two values and plotted the CDF plots. At higher bandwidth levels the two protocols doesn't differ but at lower bandwidth QUIC has a greater tendency to request data in larger chunks when compared to TCP. Since QUIC is requesting in larger chunks we can estimate that it requires lesser number of requests to server to fetch data which is confirmed by the Fig. 3.8-3.10.

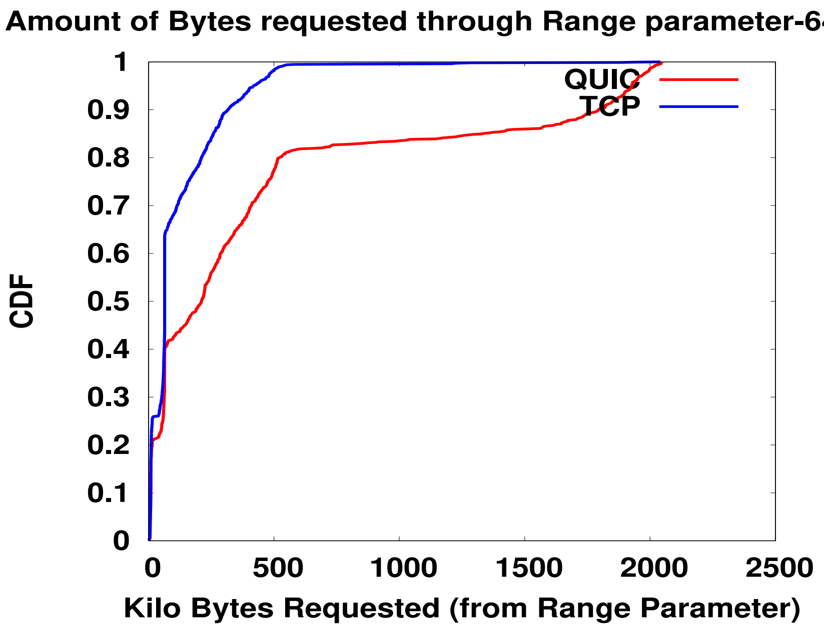


Figure 3.14: CDF of range at 64 kbps

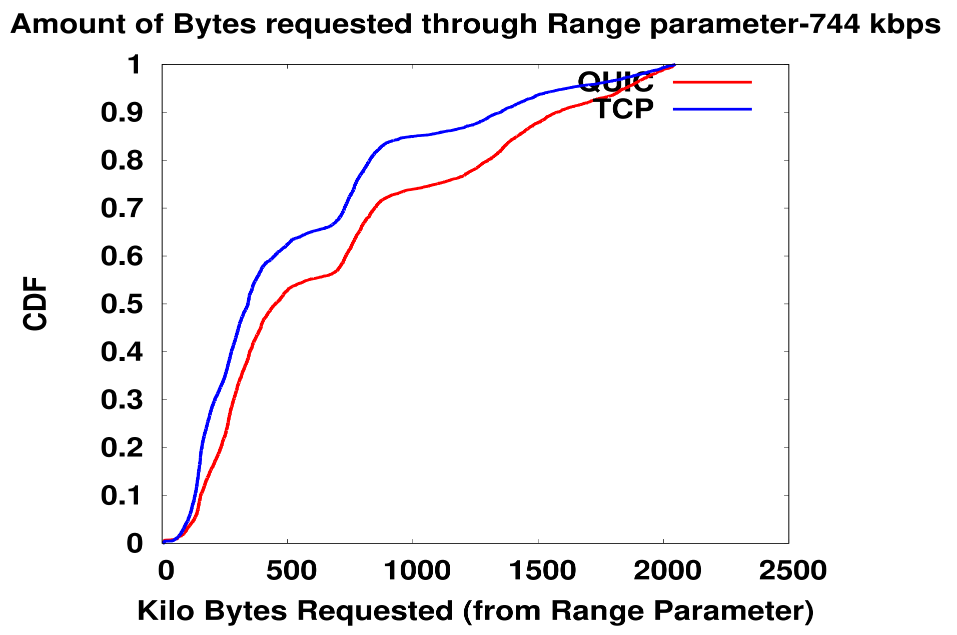


Figure 3.15: CDF of range at 744 kbps

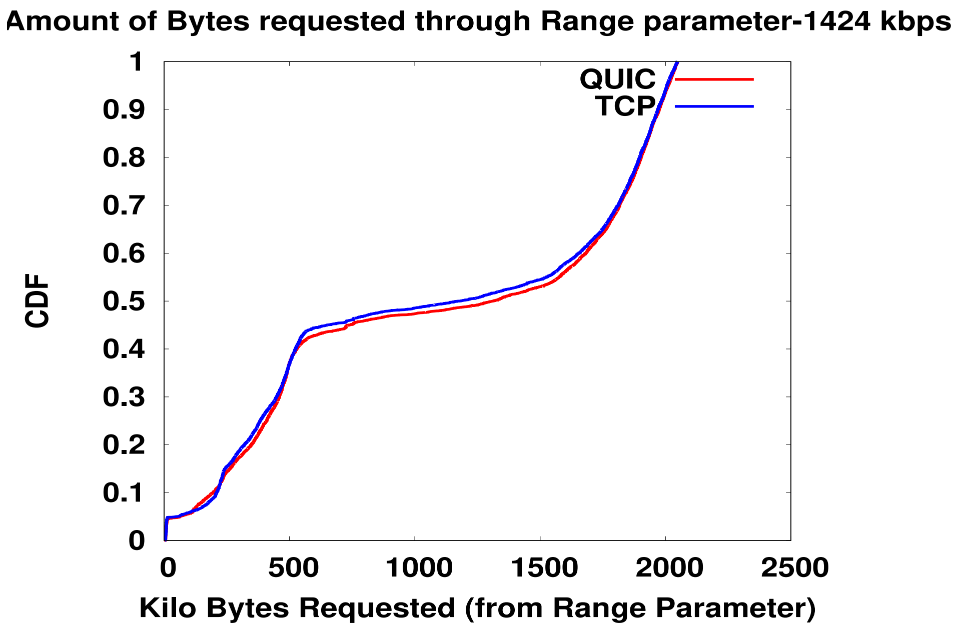


Figure 3.16: CDF of range at 1424 kbps

3.5.4 CDF for Segment Length with respect to various Bandwidth levels

This is another way of representing the *range* parameter. From the number of bytes requested through range parameter we can convert it into duration of playback seconds using the bit-rates for the *itags*. The trends will be similar to the CDF plots of *range* with QUIC trying to request segments with longer duration when compared to TCP at lower bandwidths but at higher bandwidths the difference is not much.

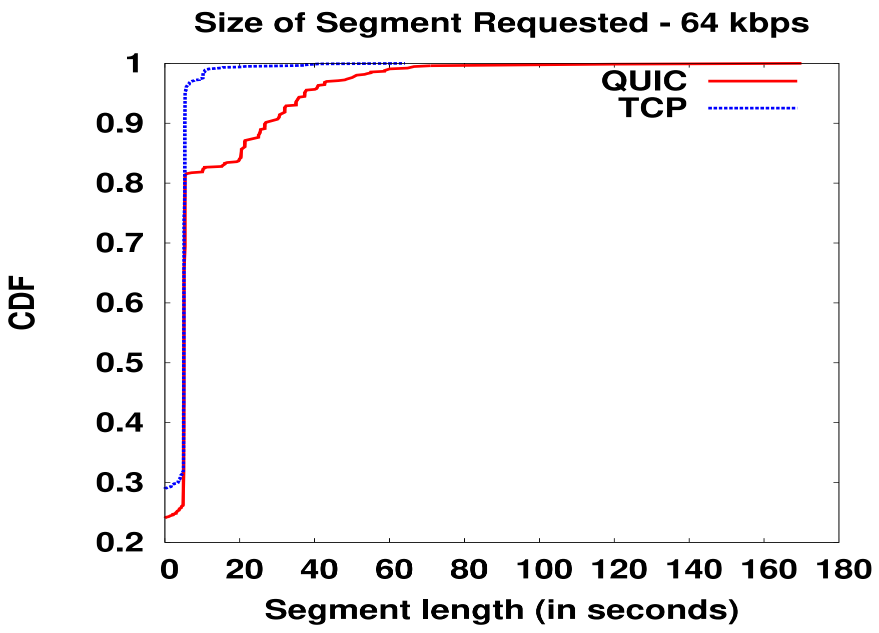


Figure 3.17: CDF of Segment Length at 64 kbps

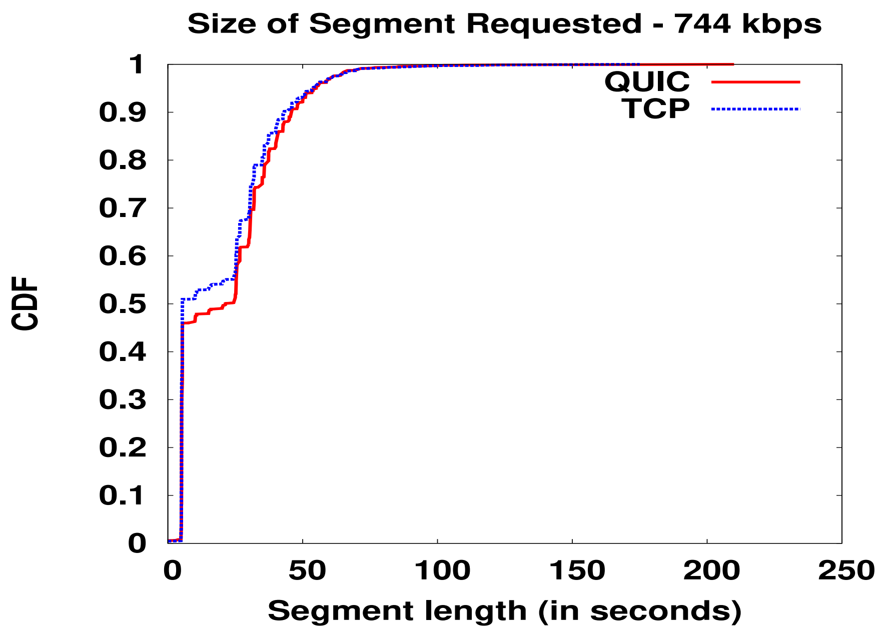


Figure 3.18: CDF of Segment Length at 744 kbps

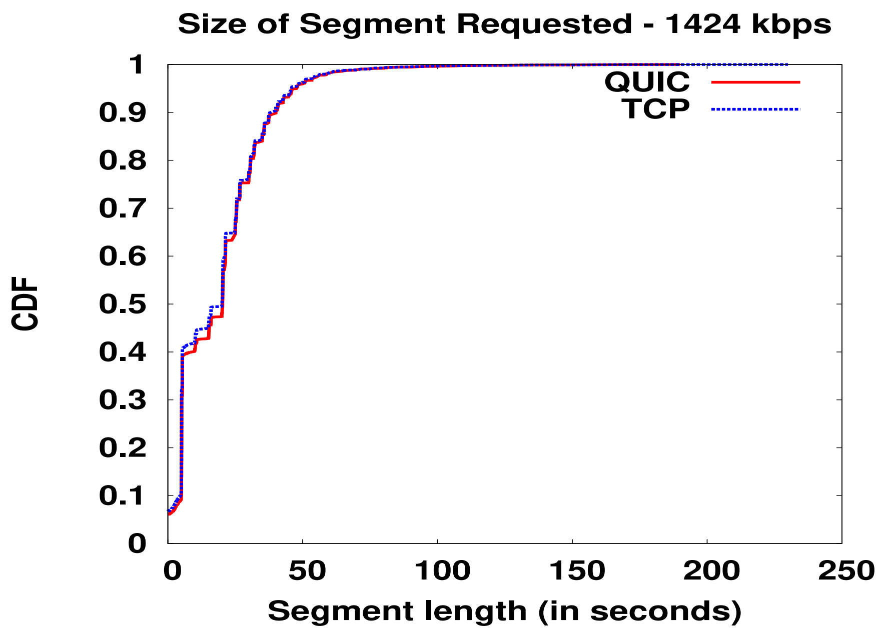


Figure 3.19: CDF of Segment Length at 1424 kbps

3.5.5 CDF for *itag* with respect to various *rbuf* ranges(buckets)

The two protocols does not differ much with respect to *rbuf*. When the buffer is smaller in amount they fetch all the itags but when the buffer is sufficiently large both the protocols fetch data only for higher itags as it is evident in the last two plots. QUIC makes more number of requests than TCP when the client has buffered large amount of data. This can be interpreted as QUIC's estimation that since buffer has large amount of data the rate of data consumption is less when compared to data fetched so bandwidth is sufficiently good and it can make further requests. It is also evident that most of the requests to server were made when the buffer is low in data as both the protocols interpret this as data depletion at a faster rate so they try to fetch data at a faster rate which means more number of requests made.

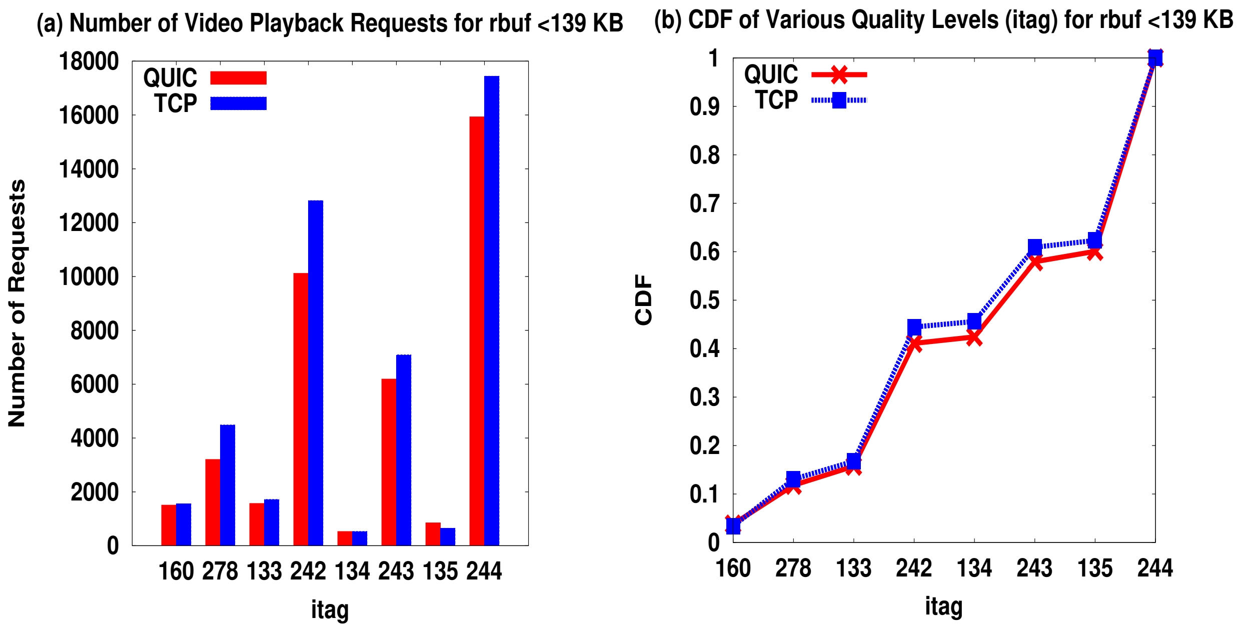


Figure 3.20: Number of requests and CDF of itag for rbuf <139 KB

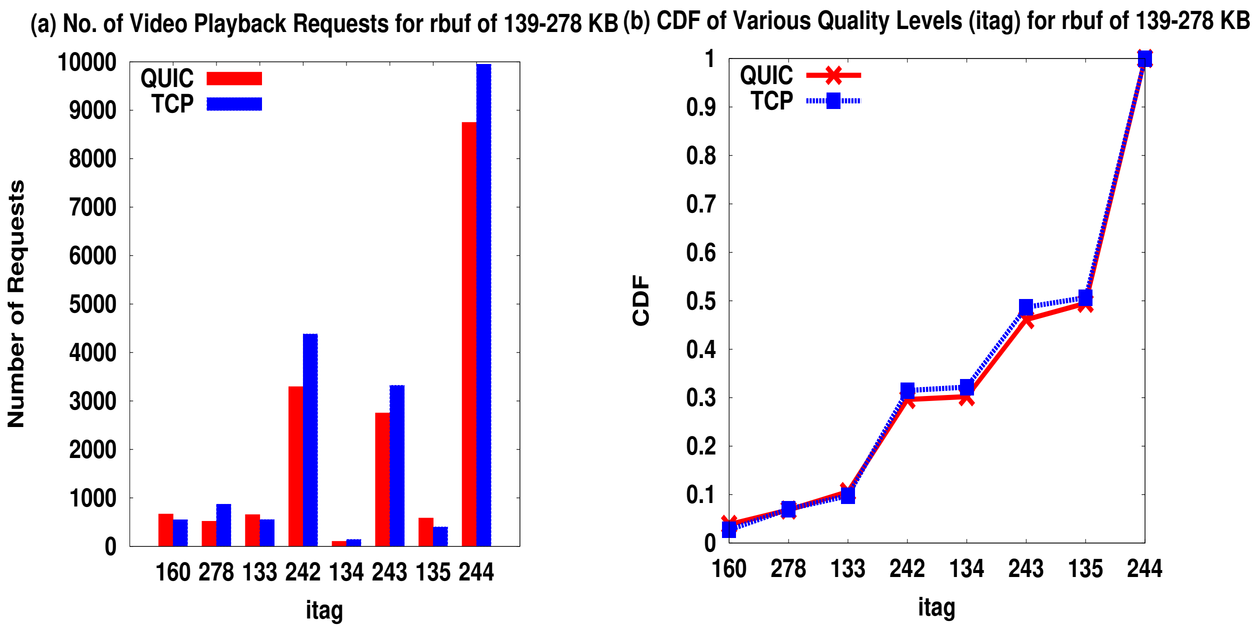


Figure 3.21: Number of requests and CDF of itag for rbuf 139-278 KB

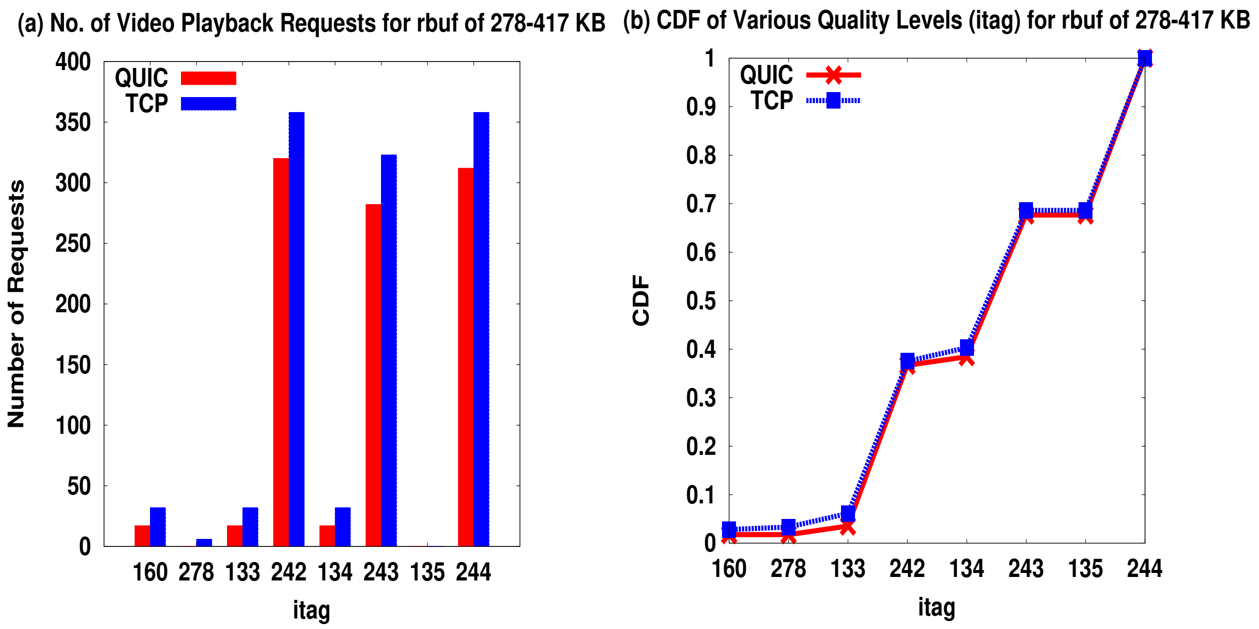


Figure 3.22: Number of requests and CDF of itag for rbuf 278-417 KB

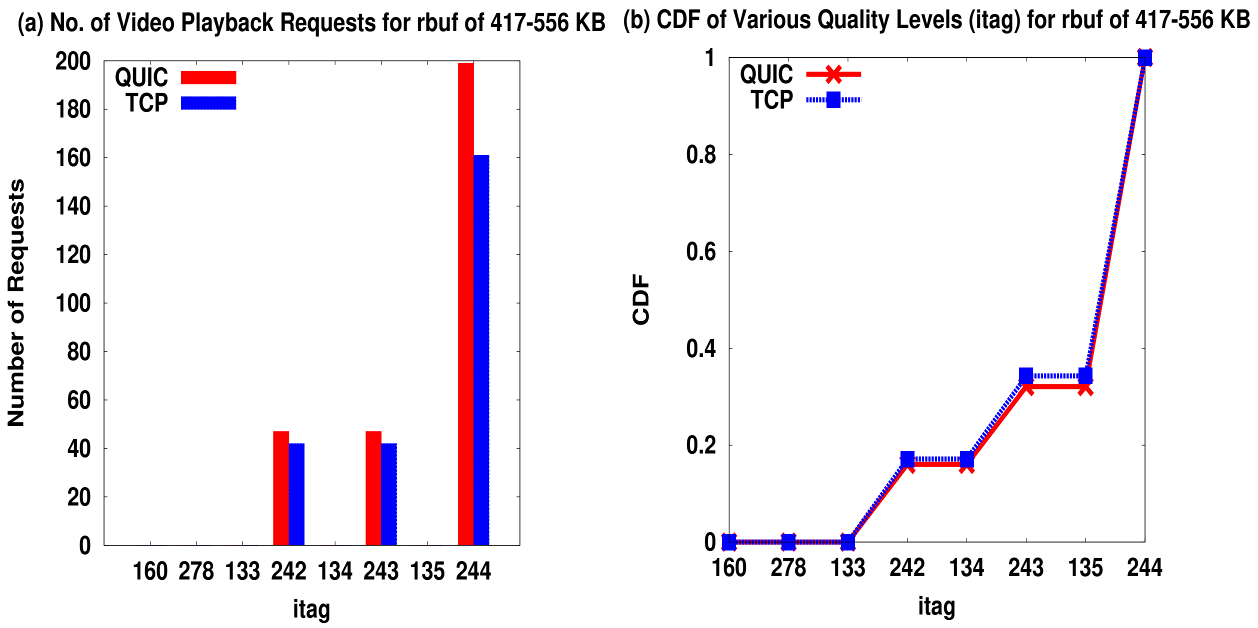


Figure 3.23: Number of requests and CDF of itag for rbuf 417-556 KB

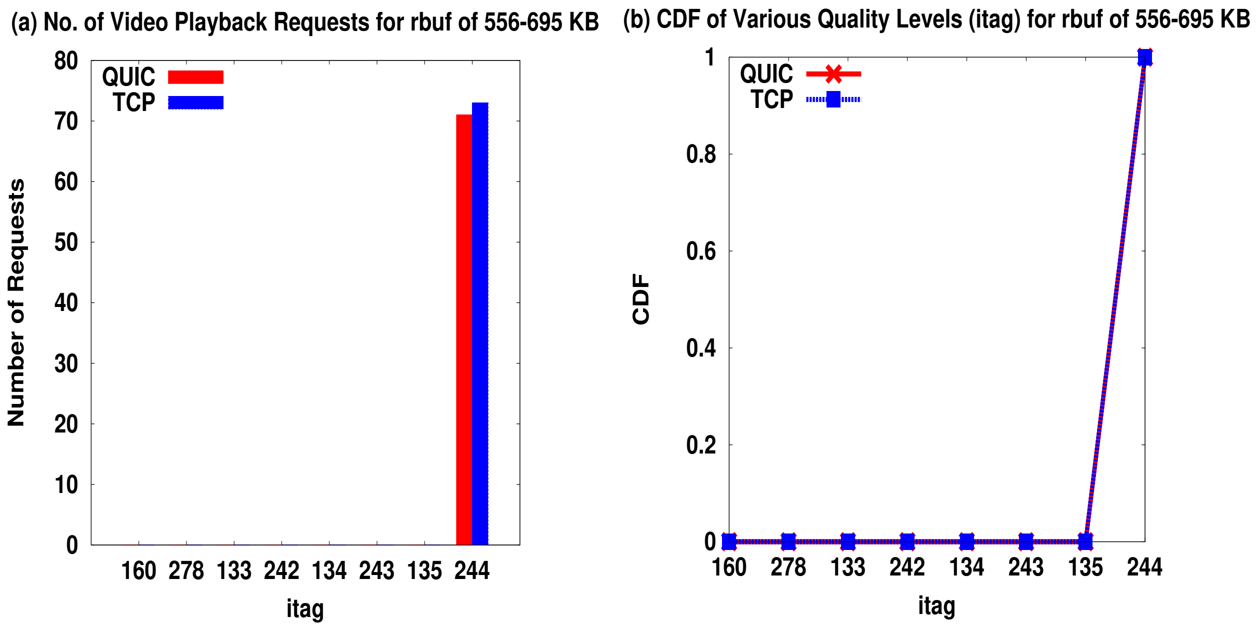


Figure 3.24: Number of requests and CDF of itag for rbuf 556-695 KB

3.5.6 CDF for *range* with respect to various *rbuf* ranges(buckets)

The behaviour of both the protocols is similar when the buffer is quite low or quite high but when the buffer is in the medium range QUIC and TCP show a different pattern. When the buffer is low both the protocols take a conservative approach and request the data in smaller chunks but QUIC requests the data in a slightly larger chunks. When the buffer is above a threshold value QUIC and TCP differ in that QUIC tries to download the data in larger chunks when compared to TCP. When the buffer is sufficiently high both the protocols doesn't differ much.

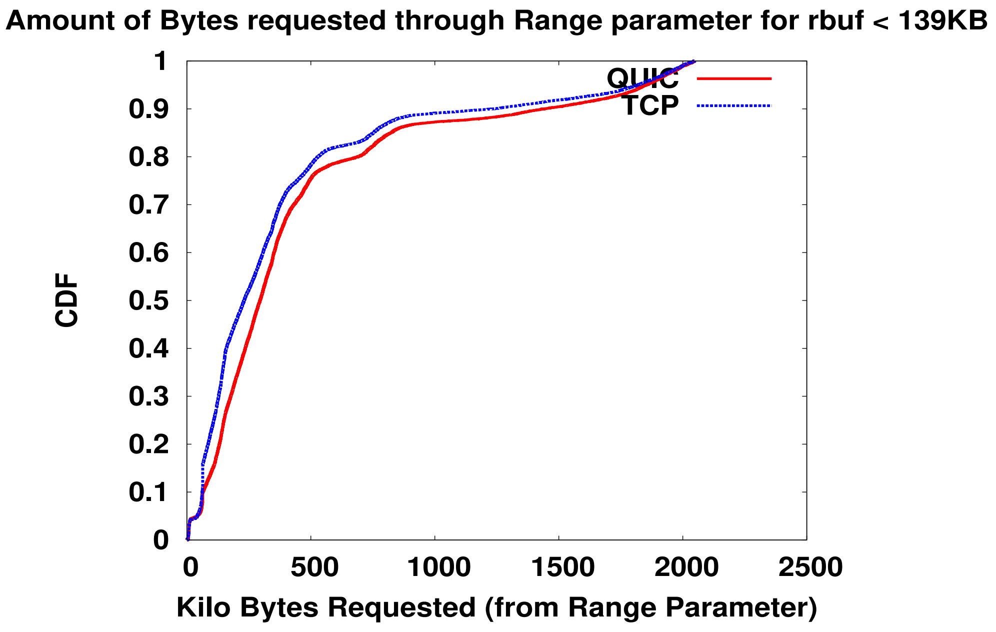


Figure 3.25: CDF of range for rbuf $<$ 139KB

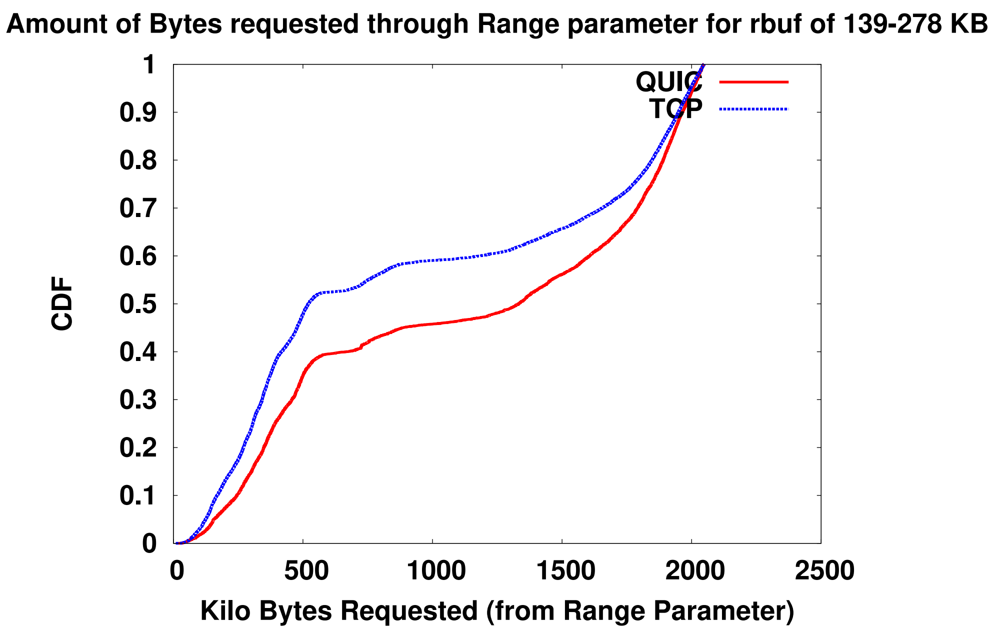


Figure 3.26: CDF of range for rbuf 139-278KB

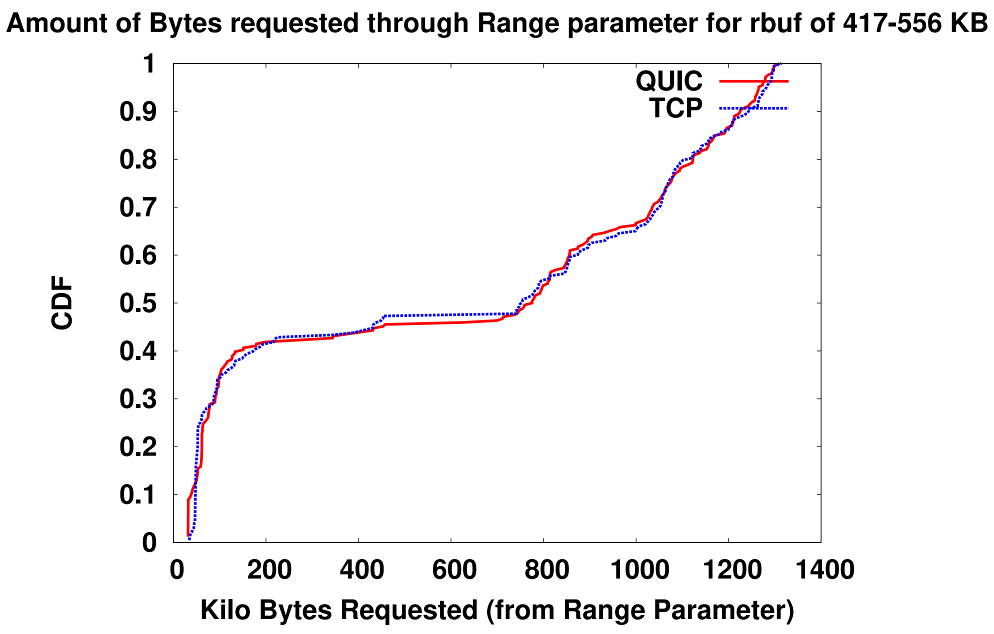


Figure 3.27: CDF of range for rbuf 417-556KB

3.6 Data Wastage by QUIC and TCP

Let us look at the total data downloaded by QUIC and TCP and the data wasted by both the protocols. Data wasted is computed by considering the fact that if there is lower resolution data when a higher resolution is being played then the lower resolution data is being wasted. The amount of data downloaded and data wasted has been calculated using the bit rates available for each itag in the HAR files.

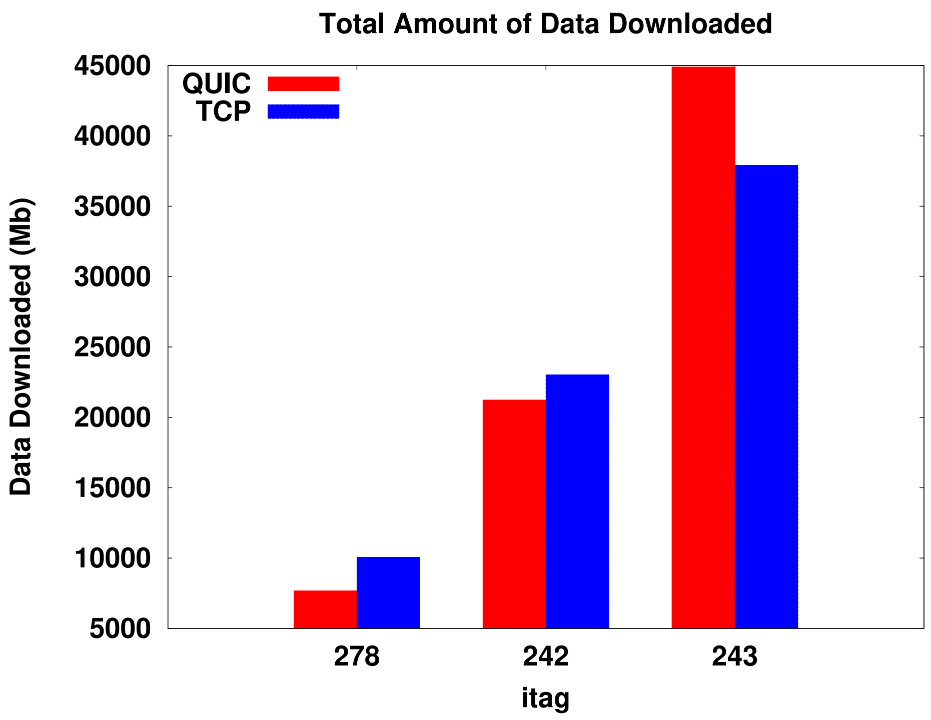


Figure 3.28: Amount of Data Downloaded

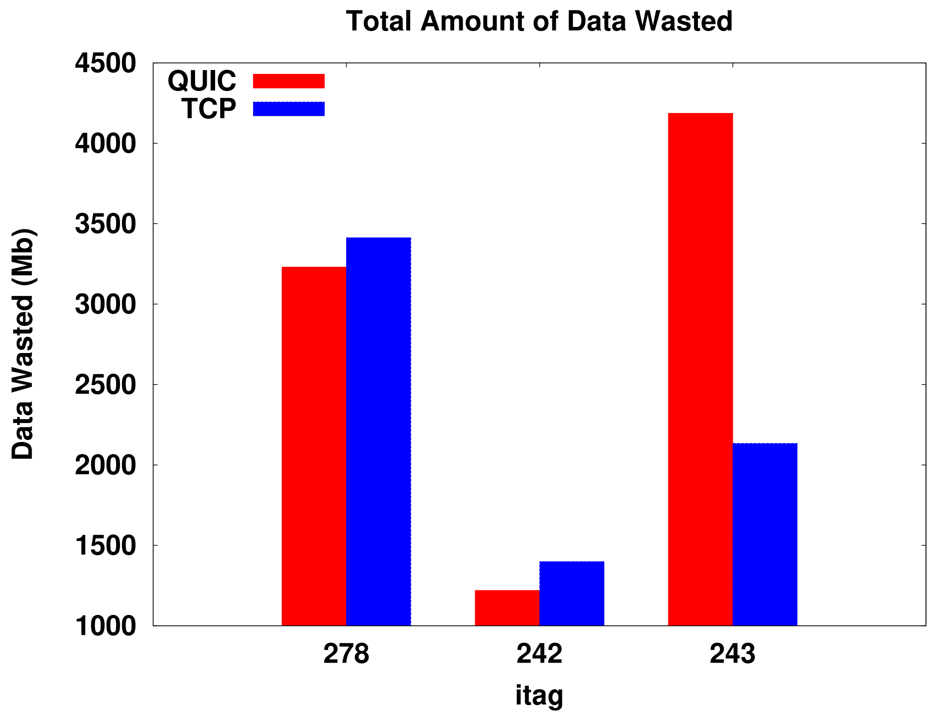


Figure 3.29: Amount of Data Wasted

Itag	Resolution	Data Downloaded (Mb)	DataWasted (Mb)	% of Data Wasted
278	256x144	7669	3230	42.12%
242	426x240	21232	1219	5.74%
243	640x360	44894	4187	9.33%
Total		73795	8636	11.7%

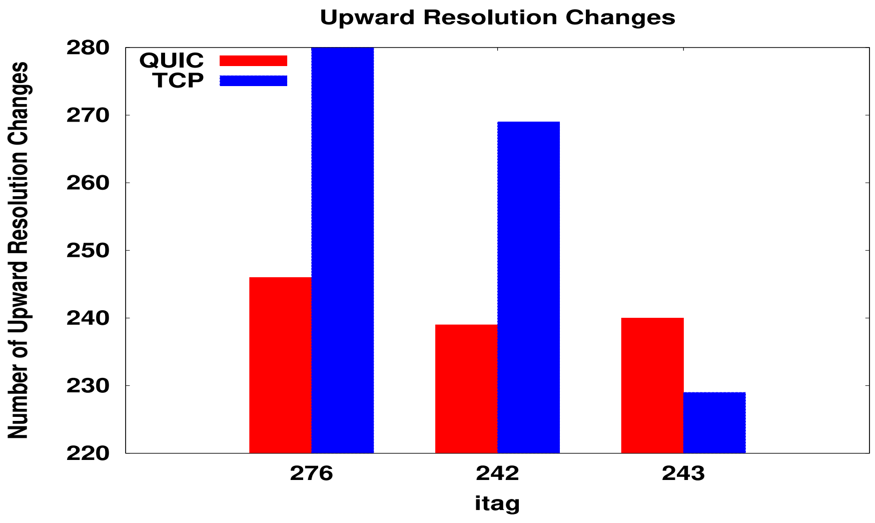
Table 3.3: Data Wastage for QUIC

Itag	Resolution	Data Downloaded (Mb)	DataWasted (Mb)	% of Data Wasted
278	256x144	10049	3412	33.95%
242	426x240	23028	1399	6.07%
243	640x360	37911	2134	5.63%
Total		70988	69456	9.78%

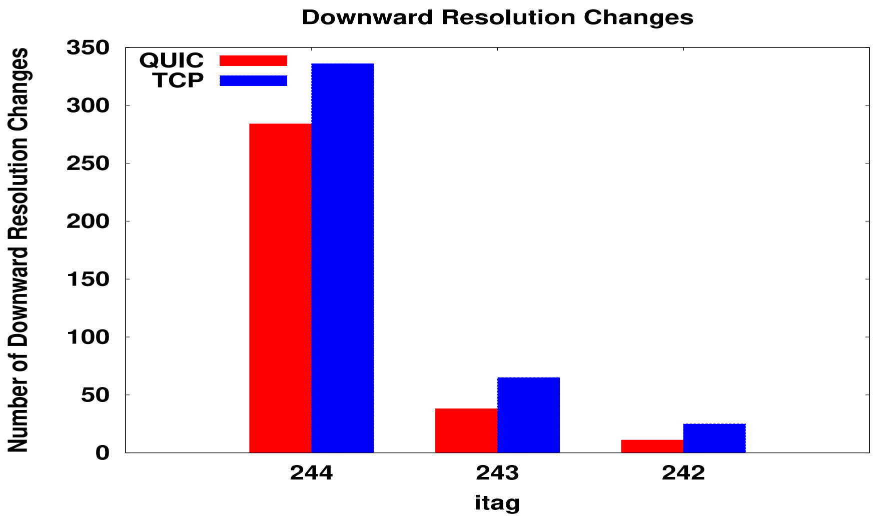
Table 3.4: Data Wastage for TCP

3.7 Number of Video Resolution Changes

For each protocol we counted the number of times Video resolution has changed and categorized it into upward resolution changes and downward resolution changes. It should not be confused that TCP is performing better with TCP making more number of upward resolution changes because we also need to look at the total number of resolution changes ans the number of downward resolution changes.



(a) Plot for Upward Resolution Changes



(b) Plot for Downward Resolution Changes

Figure 3.30: Plot for Number of times Video Resolution Changed

Chapter 4

4.1 Summary

From all these plots we can summarize that there is a trade-off between data wasted and the video resolution observed. From the CDF plots we can conclude that QUIC has a higher tendency to provide the user with better resolution. In order to avoid buffering QUIC is downloading the data in such a rate that rbuf is sufficiently good enough for the given bandwidth but at this point if the bandwidth increases and it is sufficient enough to jump to higher resolution then QUIC quickly jumps to fetch the higher video chunks wasting the data that is already present in the buffer. TCP as we know follows additive increase so it will play the data already present in the buffer for a good amount of time before fetching the data for higher resolutions. **There is a price we need to pay in terms of more data wastage to have better video resolution**. This wastage may be significant if the user has Internet connection with data limits. The main parameters that decide how the next segment has to be downloaded are *rbuf* and the bandwidth. YouTube uses a combination of these two parameters (may be some more unknown parameters) to decide which *itag* has to be requested next. This is the reason for plotting CDF with respect to bandwidth and *rbuf*. QUIC also provides the user with a better viewing experience as the number of resolution changes made by QUIC are lesser when compared to TCP.

4.2 Future Work

Motivated by the fact that QUIC provides user with a better viewing experience we would like to investigate ways in which data wastage can be reduced and try to check whether the behaviour is similar with other video streaming services if QUIC is supported in them. We would also like to explore the ways in which we can design a new streaming service in which both the advantages of TCP and QUIC are employed.

References

[1] Peter Megyesi, Zsolt Kramer, Sandor Molnar "How quick is QUIC?" in IEEE ICC 2016 - Communication QoS, Reliability and Modeling Symposium.

[2] Gaetano Carlucci, Luca De Cicco and Saverio Mascolo. "HTTP over UDP: an Experimental Investigation of QUIC." ACM SAC15, April 13 - 17 2015, Salamanca, Spain.

[3] (2016) Netfilter queue (last accessed: July 2016). [Online]. Available: `http://www.netfilter.org/projects/libnetfilterqueue`

[4] (2016) Format and resolution of YouTube videos (last accessed: July 2016). [Online]. Available: `http://www.genyoutube.net/formats-resolution-youtube-videos.html`

[5] D. K. Krishnappa, D. Bhat, and M. Zink, "DASHing YouTube: An analysis of using DASH in YouTube video service," in Proceedings of IEEE 38th LCN, 2013, pp. 407415.

[6] (2016) Introducing json (last accessed: July 2016). [Online]. Available: `http://www.json.org/`

[7] R.Hamilton et al. QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 draft-hamilton-early-deployment-quic-00.

[8] H. Li, H. Wang, J. Liu, and K. Xu, "Video sharing in online social networks: measurement and analysis," in Proceedings of the 22nd NOSSDAV, 2012, pp. 8388.

[9] Robert Lychev, Samuel Jero, Alexandra Boldyreva and Cristina Nita-Rotaru. "How Secure and Quick is QUIC? Provable Security and Performance Analyses", 2015 IEEE Symposium on Security and Privacy