Investigating "Taking a Long Look at QUIC"

Daniel P. M. de Mello[*]
Purdue University
West Lafayette, Indiana, USA
ddemello@purdue.edu

Taisuke Mori
Purdue University
West Lafayette, Indiana, USA
tmori@purdue.edu

Amit Roy
Purdue University
West Lafayette, Indiana, USA
roy206@purdue.edu

Yucheng Zhang
Purdue University
West Lafayette, Indiana, USA
zhan4332@purdue.edu

Jincheng Zhou
Purdue University
West Lafayette, Indiana, USA
zhou791@purdue.edu

ABSTRACT

QUIC is a transport layer protocol originally developed by Google to provide a fast and reliable internet connection. In this work, we evaluate the performance of QUIC by comparing the file download time of QUIC and TCP and investigate if fairness between QUIC and TCP is maintained when the two protocols share a network. Our experiments reveal that QUIC downloads files faster than TCP in every scenario and QUIC is very unfair to TCP (taking a much larger fraction of the bandwidth than TCP), especially in low bandwidth / high congestion scenarios and when the number of TCP connections is small.

KEYWORDS

QUIC, transport-layer performance, software-defined network

1 INTRODUCTION

QUIC (Quick UDP Internet Connections) is a transport layer network protocol, which was originally developed by Google [4] to provide a fast and reliable internet connection and was defined as an IETF (Internet Engineering Task Force) standard in 2021 [1]. Since its first release in 2013, QUIC has been growing in usage and now accounts for a significant amount of internet traffic. As of 2022, QUIC is the default protocol for Google Chrome and is implemented on many other browsers such as Microsoft Edge, Firefox and Safari. Built on top of UDP, QUIC is designed to reduce latency while keeping high security and congestion control comparable to the traditional protocol TCP.

Unlike TCP, QUIC is implemented in user space, which allows the protocol to be improved rapidly and, at the same time, makes it challenging to understand its current performance. To evaluate the performance of QUIC, Kakhki et al. [5] conducted experiments in 2017 where they built a testbed that emulates servers and clients to compare the latency and throughput between QUIC and TCP under various conditions, and found QUIC performs better than TCP in most scenarios. In 2020, Wong and Tieu [8] performed reproducing experiments and got similar results though there are some disagreements. Specifically, when downloading a large number of small objects, TCP is faster in Kakhki et al.'s experiment while QUIC is faster in Wong and Tieu's experiment. Wong and Tieu conjecture the reason for it is that QUIC's problem of updating the slow start threshold had been fixed in the newer version of QUIC they used for their experiments. Also, they get different results for the comparison of fairness. Kakhki et al. observe that, when the two protocols share a network, QUIC consumes more bandwidth than TCP regardless of the number of flows of TCP. On the contrary, Wong and Tieu found that the total bandwidth consumed by TCP is approximately the same as that of QUIC when there are two TCP flows and one QUIC flow in a network. Wong and Tieu argue that their result is reasonable given the fact that both TCP and QUIC employ the Cubic congestion control algorithm.

In this work, we aim to understand the performance of QUIC by reproducing Kakhki et al.'s and Wong and Tieu's experiments. We evaluate the latency by comparing the file download time of QUIC and TCP in various settings. Also, we analyze the fairness when QUIC and TCP share a single network. Similar to Wong and Tieu, we build a virtual environment on an EC2 instance with Mininet [1]. This allows us to emulate networks with desired configurations, and quickly conduct experiments in various conditions.

From the experiments, we found:

- QUIC outperforms TCP in every configuration,
- QUIC consumes much more bandwidth than TCP, but as the number of TCP connections increases, QUIC consumes less and less bandwidth. The extent of this behavior is also influenced by the total available network bandwidth.

2 RELATED WORK

Since QUIC was released, a number of studies on its performance have been done. Kakhki et al. [5] conducted experiments in 2017 where they built a testbed that emulates servers and clients to compare the latency and throughput between QUIC and TCP under various conditions, and found QUIC performs better than TCP in most scenarios. In 2020, Wong and Tieu [8] performed reproducing experiments and got similar results though there are some disagreements. Shreedhar et al. [7] also compared QUIC with TCP but, to understand its performance in real network settings, they measured the throughput of downloading contents from several major websites and internet services such as Google Drive cloud storage and YouTube.

Fairness analysis has been a major interest for a long time in TCP congestion control. In 1988, in one of the earliest studies on TCP congestion control, Jacobson [3] introduced some basic features such as slow-start and round-trip timing, although they mentioned

[*]Authors ordered by the last name

[1]http://mininet.org/

that those techniques did not necessarily guarantee fairness among connections. Hasegawa et al. [2] analytically investigated several major TCP congestion control techniques and argue that Tahoe and Reno cannot achieve fairness between connections with different propagation delays. After wireless Local Area Networks (LAN) became widely used, Pilosof et al. [6] investigated TCP fairness on wireless LAN through the simulation and found that different buffer availability of base stations could cause a significant unfairness.

3 EXPERIMENT SETUP

3.1 File Download Time Experiment

In this work, we aim to study two questions. First, how much faster is QUIC compared to TCP? Second, is QUIC fair to TCP, i.e., utilizing a similar amount of bandwidth as TCP, when both protocols share the same network resource? Besides answering these two questions categorically, we also want to study how the results change under different level of congestion in the network. Would TCP become less slower than QUIC when the available total network bandwidth is very high? Would QUIC become unfair to TCP under one congestion scenario, but changes its behavior in another scenario?

To answer these questions, we design two experiments to answer each question respectively. The first experiment is the File Download Time (FDT) experiment, in which we set up QUIC and TCP connections to download a same set of files from the server host to the client host. The QUIC and TCP connections are established sequentially, so when each individual protocol is running, it has the full network bandwidth at its disposal. We measure the ratio of QUIC's download time to TCP's download time, so if the measurement is smaller than 1, then QUIC downloads the files faster than TCP does, and the smaller the value, the faster QUIC is to TCP. Besides (1) the network's bandwidth, we also vary (2) the number of files to download and (3) the individual file sizes in order to study whether these two variations would affect QUIC and TCP's relative performance.

3.2 Fairness Experiment

The second experiment measures fairness between QUIC and TCP, where we establish multiple QUIC and TCP connections simultaneously to download a fixed-sized file. As these connections runs parallelly in the network, they must share the same restricted network bandwidth. A fair protocol should therefore not occupy too large a fraction of the available bandwidth, causing congestion for the other protocol.

We record and measure the throughput, or the data transfer rate, of each individual QUIC and TCP client-server connection throughout its lifetime. We then plot these throughput against time to analyze the fairness situation. The parameters we control in this experiment are (1) the number of QUIC connections, (2) the number of TCP connections, and (3) the network's bandwidth.

3.3 Experiment Environment

Computer networking experiments are tricky to set up because various factors and background processes in the operating system and in the network can influence the experiment results. To ensure a clean and reproducible environment with minimal influence from background processes, we conduct our experiments on an Amazon EC2 virtual machine, uses Mininet to set up the network environment, and run the client and server programs within a Mininet Docker image [2]. We set up a simple network topology as shown in Figure 1 with two hosts h_1 and h_2, one serving as the client host and the other one the server host, and a switch s_1 connecting the two hosts with an ONOS controller [3] running in the background to install forwarding rules into the switch s_1. The bandwidth of all the links in this network is set by passing a parameter when starting the Mininet Docker image.

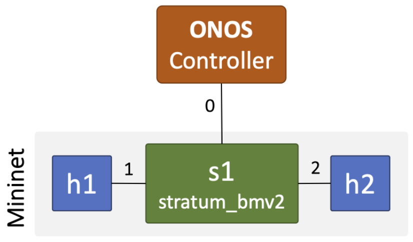


Figure 1: The Mininet network topology for the experiments

3.4 Server and Client Programs

In both experiments, the QUIC connection is implemented as a web browser accessing a QUIC server, and the TCP connection is implemented as the same web browser accessing a HTTPS server. Both type of servers reside on a server Mininet host of our choosing and serve the same ad-hoc website containing the same files to be downloaded. The web browser, on the other hand, runs on the other Mininet host.

We obtained a QUIC server program by building the sample QUIC binaries [4] from Google's official Chromium source [5]. Building this QUIC binary proved to challenging, as the entire Chromium source needs to be checked out and verified before the build command specific to the QUIC server can be run. We eventually set up a dedicated Google Cloud virtual machine to carry out this build, which took more than 100GB disk space and more than 3 hours to finish. Once the build process was finished, we extracted both the QUIC binaries and a list of self-signed SSL certificate files from the build target, the latter being a prerequisite for the sample QUIC server to run.

Compared to the QUIC server, the HTTPS server was easier to set up. We used the Python 2 `SimpleHTTPServer` module [6] to launch an HTTPS server. We chose to set up HTTPS connections instead of HTTP connections because the aforementioned QUIC server requires SSL certification, and we want to eliminate the possibility that the certification process could negatively impact the protocol's performance.

[2]We used the `mn-stratum` Docker image for the experiments: https://hub.docker.com/r/opennetworking/mn-stratum
[3]https://opennetworking.org/onos/
[4]Playing with QUIC: https://www.chromium.org/quic/playing-with-quic/
[5]Checking out and building Chromium on Linux: https://chromium.googlesource.com/chromium/src/+/main/docs/linux/build_instructions.md
[6]https://docs.python.org/2/library/simplehttpserver.html

For the client program, we chose the Google Chrome web browser [7]. The reason is three-fold. First, Google Chrome is a modern browser that implements both QUIC and HTTPS protocol, so we are able to use the same client program to connect to both the QUIC server and the HTTPS server, thus ensuring that the difference in the client program's performance would not be a factor that affects our experiment results. Second, Google Chrome is a production-ready web browser developed by Google, who also developed and released the QUIC protocol, so there is some level of performance guarantee for the QUIC connections. Finally, according to some market share report, Google Chrome takes up about 65% of the worldwide browser market [8]. Thus, conducting experiments via Google Chrome should reflect the real-life scenario to a great extent.

A final note to mention is that, in the fairness experiment, when we establish multiple QUIC and TCP connections, we launch a dedicated QUIC or HTTPS server process and a dedicated Google Chrome process for each connection.

4 RESULTS

4.1 File Download Time

The first experiment we have is about file download time. We want to compare the latency between QUIC and TCP under different bandwidths, file sizes, and file numbers.

For QUIC server, we set it up by the existing QUIC binaries. For HTTPS server, we set it up by Python SimpleHTTPServer library. There are two different ways to set up clients in our project. The recommended way is to use Google-Chrome as clients for both QUIC and TCP to guarantee fairness of competition. But we can also use QUIC client and a simple wget client for TCP. We run all our experiments in Mininet and start the Mininet environment with four different bandwidths: 1 MB, 10 MB, 50 MB and 100 MB respectively. For each transmission we use one server and one client. The files are transferred in sequence if there are more than one file to download. For each configuration, we run for three times and use the average time as the final results to make our data more stable and reliable.

For the first part of file download tests, we collect the download time under different file sizes from 5 KB to 10 MB. And the results are shown in 2c and 2a. In the pictures we show the (T_QUIC/T_TCP) ratio. For all the configurations QUIC is faster than TCP so the ration is always between 0 and 1. The darker the color is, the closer the ratio is to 0 and it means QUIC is more dominant. On the other hand, the light green means under this circumstance QUIC and TCP performances are close.

The different ways of setting up clients show similar patterns here. QUIC and HTTPS performance are similar when downloading small files (5 KB and 10 KB). However, QUIC dominates when downloading large files (500 KB and above). We guess this is because when the file size is small, the time spent on setting up connections will account for a larger percentage and the difference in transfer speed is not that obvious. And when the bandwidth gets smaller, QUIC becomes more dominant. We can draw the conclusion that QUIC is faster in transer speed.

[7]We used Chrome version 107.0.5304.110

[8]Browser Market Share Worldwide: https://gs.statcounter.com/browser-market-share

For the second part of file download tests, we fix the total size of files to be 1MB but there are different number of files from two hundred 5 KB files to one 1 MB file. And the results are shown in 2d and 2b. Same as above in the pictures we show the (T_QUIC/T_TCP) ratio. For all the configurations QUIC is faster than TCP so the ration is always between 0 and 1. The darker the color is, the closer the ratio is to 0 and it means QUIC is more dominant. On the other hand, the light green means under this circumstance QUIC and TCP performances are close.

The different ways of setting up clients still show similar patterns here. QUIC and HTTPS performance are similar when downloading small files of large number (5 KB * 200 and 10 KB * 100). However, QUIC dominates when downloading large files (500 KB * 2 and above) of small number. And when the bandwidth gets smaller, QUIC becomes more dominant. We guess this is because when transferring small file of large number, the time spent on setting up connections will account for a larger percentage and the difference in transfer speed is not that obvious. We can still safely draw the conclusion that QUIC is faster in transer speed.

In the comparison between QUIC and TCP, Kakhki et al. [5] concluded that "QUIC outperforms TCP in every scenario except in the case of large numbers of small objects". Whereas Wong and Tieu [8]'s experiments showed that QUIC is faster than TCP in most of the time except 500KB file download with 100Mbps bandwidth. However, in our experiments we have found that QUIC outperforms TCP in every configuration, the reason could be the not having head-of-line blocking for newer versions of both QUIC and TCP.

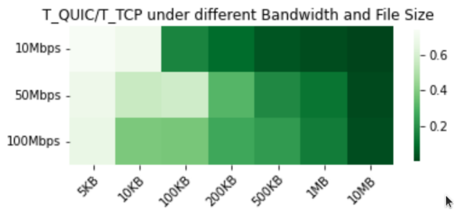
Our file download experiment results are consistent with the results of the reproducing experiments by Wong and Tieu [8]. We both see that the larger the file size is, the more dominant QUIC will be. However, they are not consistent in the file number experiment. Another thing interesting is that in their results the larger the bandwidth is, the more dominant QUIC will be, which is exactly opposed to our observations. We do not know the exact reason but we guess it is due to different versions of QUIC.

4.2 Fairness

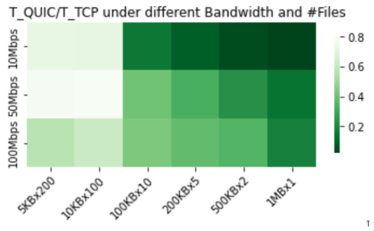
Fairness is a property of transport protocols, measuring how much of the bandwidth it uses in relation to other parallel transport protocols flows. It is desirable that each client-server in a connection uses the same share of the bandwidth, and the closer we are to this situation, the higher fairness we obtain.

In Figure 3 we display our main results for the fairness experiments. We use a single mininet link, with 1 QUIC client sharing it with varying numbers of TCP clients (1, 3 TCP clients), and the experiment is repeated for varying bandwidth sizes for the channel (1, 5, 10 Mbps). Numbers of TCP clients vary along the columns, and bandwidth sizes vary along the rows. Each client is downloading an 8 MB file. Each TCP client accesses a dedicated HTTP server sharing the channel.

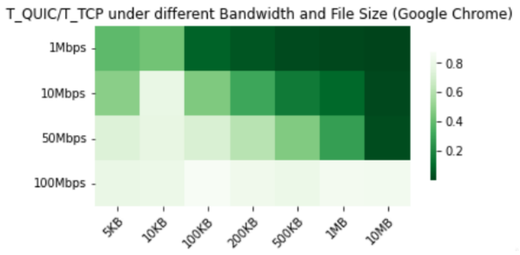
We can observe from this experiment that, for the 1 Mbps link, QUIC tends to dominate over TCP more than it does over 5 Mbps and 10 Mbps links, although QUIC can still dominate TCP in these cases too. To verify this claim, we note that QUIC took over 80 % of the bandwidth against 1 TCP in the 1 Mbps channel, which was the same fraction of the bandwidth for the 5 Mbps channel



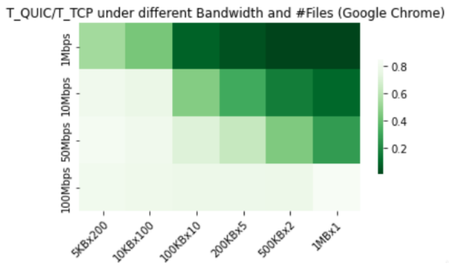
(a) QUIC vs TCP with different file size



(b) QUIC vs TCP with different #files



(c) QUIC vs TCP with the different file size (Google Chrome)



(d) QUIC vs TCP with different #files (Google Chrome)

against 1 TCP (also 80 %), but a smaller fraction than it did for the 10 Mbps against 1 TCP (between 60-70%). This trend is also noticeable when comparing 1 QUIC against 3 TCPs, as in the 1 Mbps it uses approx 50 % of the bandwidth, in the 5 Mbps it also uses approximately 50 % of the bandwidth, and in the 10 Mbps channel it actually gets dominated by the other TCP clients, using between 5-20 % of the bandwidth. When increasing the number o TCP clients and maintaining the bandwidth fixed, we can observe a decrease in the fraction of the bandwidth that TCP occupies: From 80 % to 50% in 1 Mbps, from 80 % to 50 % in 5 Mbps, and from 60-70% to 5-20% for 10 Mbps. The most fair situation we managed to obtain with these experiments was 3 TCP CLIENTS against 1 QUIC in the 10 Mbps channels. We can conclude that, for smaller bandwidths, a even higher ration of TCP clients to QUIC clients would be required to achieve fairness. The cause for this behavior, that is, QUIC being more unfair the lower the bandwidth, is still unclear, especially considering that according to the authors from [8], both QUIC and TCP are supposedly using the same Cubic congestion control protocol. [8] also mentions that, despite both implementing Cubic congestion control, it might be the case that QUIC implementation uses different parameters, such that it ends up emulating the congestion control equivalent to 2 TCP connections. However, this hypothesis doesn't explain the higher unfairness of QUIC for smaller bandwidths. And we can conclude that something in QUIC's implementation of congestion control is likely making it increase the congestion window size more agressively than TCP for lower bandwidths.

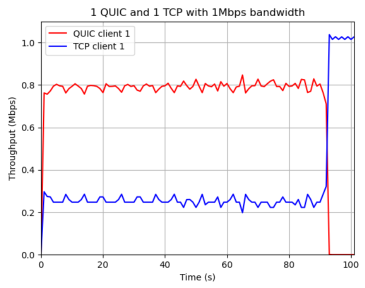
We also tried to reproduce the same Fairness results with the same specifications as [8], that is, 5 Mbps and 1 QUIC with 2 TCPs, which was the number of TCPs clients found to be necessary for fairness in this situation. The results are depicted in Figure 4. We can clearly observe that the fraction that QUIC occupies is far from being fair (about 60 % of the bandwidth). This result goes directly against the same experiment in [8], where QUIC was shown to occupy the same fraction as each TCP client. To obtain fairness with this 5 Mbps link situation, we needed 5 TCP clients sharing it with the single QUIC client, which is depicted in Figure 5.

5 CONCLUSIONS

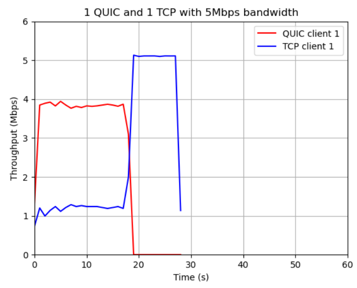
In this work, we performed experiments for understanding the performance of QUIC. Following Kakhki et al.'s and Wong and Tieu, we evaluated its latency and fairness between QUIC and TCP. To evaluate its latency, we compared the file download time of QUIC and TCP for different file sizes, number of files and bandwidth limits, and observed that QUIC outperforms TCP for all settings. For fairness, we measured the bandwidth each protocol consumes while varying the number of connections, and found that QUIC takes much more bandwidth than TCP, especially when the network link bandwidth is low and with less TCP connections.

ACKNOWLEDGMENTS

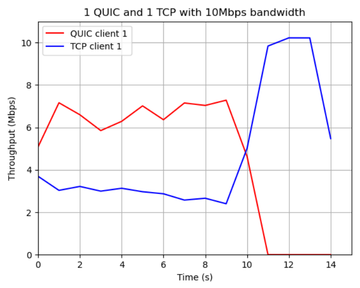
The project has been supported by the Purdue Department of Computer Science, CS 536: Data Communication And Computer Networks and also Professor Muhammad Muhammad Shahbaz and the Graduate Teaching Assistant, Sharuna Anandraj.



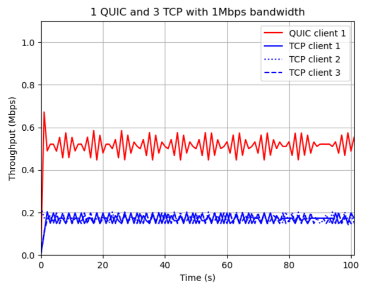
(a)



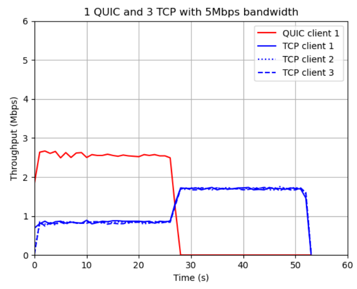
(b)



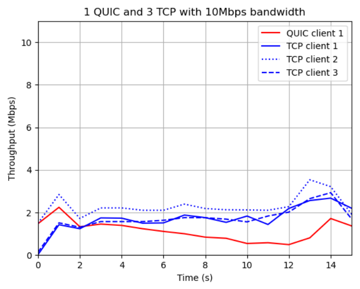
(c)



(d)



(e)



(f)

Figure 3: Fairness experiments with 1 QUIC client, varying number of TCP clients between 1 and 3, and varying channel bandwidth between 1, 5, 10 mbps.

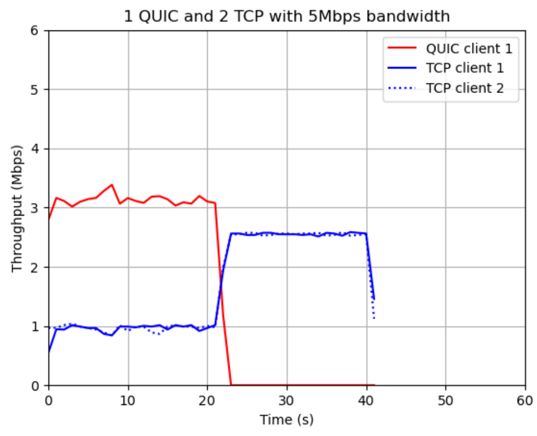


Figure 4: Reproduction of [8] fairness experiment, following the same specifications, that is, 5 Mbps channel, 1 QUIC client and 2 TCP clients

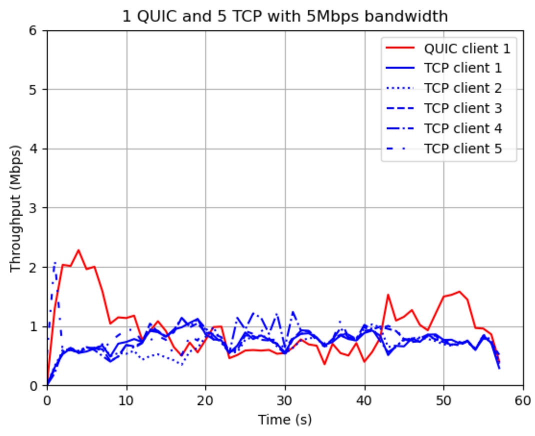


Figure 5: Reproduction of [8] fairness experiment, for 5 Mbps channel. It was necessary for us to use 5 TCP clients against 1 QUIC client, unlike [8].

REFERENCES

[1] Internet Engineering Task Force. 2021. Innovative new technology for sending data over the internet published as Open standard. https://www.ietf.org/blog/innovative-new-technology-for-sending-data/

[2] Go Hasegawa, Masayuki Murata, and Hideo Miyahara. 1999. Fairness and stability of congestion control mechanisms of TCP. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, Vol. 3. IEEE, 1329–1336.

[3] Van Jacobson. 1988. Congestion avoidance and control. *ACM SIGCOMM computer communication review* 18, 4 (1988), 314–329.

[4] Roskind Jim. 2013. Experimenting with QUIC. https://blog.chromium.org/2013/06/experimenting-with-quic.html

[5] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *proceedings of the 2017 internet measurement conference*. 290–303.

[6] Saar Pilosof, Ramachandran Ramjee, Danny Raz, Yuval Shavitt, and Prasun Sinha. 2003. Understanding TCP fairness over wireless LAN. In *IEEE INFOCOM 2003.*

Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428), Vol. 2. IEEE, 863–872.

[7] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2021. Evaluating QUIC performance over web, cloud storage and video workloads. *IEEE Transactions on Network and Service Management* (2021).

[8] Marc Robert Wong and Sarah Tieu. 2020. Reproducing "Taking a Long Look at QUIC".

A AUTHOR AND RESPECTIVE CONTRIBUTION

A.1 Daniel P. M. de Mello

- **Python HTTP server**: Small contribution at the beginning, researching and reproducing about how to use python SimpleHTTPServer to establish a connection in mininet and download a file, which was made concurrently with Yucheng.
- **Running QUIC on mininet**: At the beginning, made some elementary research about the possibility of using available QUIC implementations in C language to run directly on mininet, but this route ended up not being used, as the binary build was shown to be a simpler solution.
- **Wireshark**: Researched how to use Wireshark to explore packets from tcpdump output files. Used Wireshark to generate the plots for fairness experiments. Recorded a demo about how to use wireshark to generate plots by filtering packets by the port that each client uses to communicate with each server.
- **Fairness Experiments**: Helped to plan the fairness experiments and select the most informative results. Helped in creating the file for download in the fairness experiments, as well as necessary small modifications in the scripts to run the QUIC and TCP clients/servers on mininet. Wrote most of the fairness section in this report, and interpreted the results. Came up with the explanations regarding the possible differences in congestion window control between QUIC and TCP to explain the observed results. Also explored how to view congestion window information on Wireshark packets to explain the fairness results, but this study couldn't be finished on time for the work. Also identified some bugs in regards to saturation of bandwidth limit for experiments with bandwidth higher than 10 Mbps, which made us discard some results, opting for experiments with between 1-10 Mbps channels.

A.2 Taisuke Mori

- **System Architecture Design:** Surveyed techniques on mininet and QUIC (e.g. how we can install and run QUIC in mininet) and designed the system architecture that enables the comparison between QUIC and TCP (i.e. build 1-switch and 2-hosts by using mn-stratum).
- **Experimental Design:** Based on a thorough understanding of the target papers to reproduce, determined on which experiment we focus (i.e. comparison of the file download time and fairness evaluation based on consumed bandwidth), and design under what conditions we make a comparison (e.g. different file sizes, number of files and bandwidth limits).
- **Preliminary Experiment for File Download Time Comparison:** As a preliminary experiment, made a program that measures the file-download time of a QUIC client under different bandwidth conditions.
- **Writing Report:** Wrote the abstract, introduction, related work, and conclusion parts and created the references along with the survey of past related work.
- **Creating Presentation and Demo Materials:** Created the slide and demo video that explains the background and motivation of the project.

A.3 Amit Roy

- **Data collection for QUIC server and QUIC clients:** Written python scripts to run QUIC server and QUIC clients on the mininet environment and collected required time to download files for different configurations of bandwidths, file sizes, and the number of files.
- **Heatmap Generation for FDT:** Combined the results of FDT experiments of both QUIC and HTTP and plot them in matplotlib to produce the heatmaps for T_{QUIC}/T_{TCP} and added them in the report.
- **Compared the findings and differences with Kakhki et al.:** Pointed out how the performance of QUIC and HTTP differs under different configurations and also described how the findings of our experiments are different from the Kakhki et al [5].
- **Demo Video Preparation and Uploading:** Demonstrated the file download time experiment in the demo video. Also, merged different parts of experiments in the demo video and uploaded it to youtube.

A.4 Yucheng Zhang

- **Setting Up Environments for TCP File Download Time Experiments:** Writing Python scripts to run HTTPS server and HTTPS client on the Mininet environment. Helping solve the certificate generation process for HTTPS.
- **Data Collection for TCP File Download Time Experiments:** Writing scripts to collect and post-process data for file download experiments with different configurations of bandwidths, file sizes and number of files.
- **Writing Reports for File Download Time Experiments:** Writing the file download experiments part in the final report including environment set up, results, analysis and comparison with the Stanford reproduction paper.

A.5 Jincheng Zhou

- **Building the QUIC Toy Client/Server Binaries:** Set up the build environment by creating a Google Cloud virtual machine, found through trial and error and configured the correct parameters relevant to our experiment settings, and ran the building process to extract the QUIC binaries
- **Finding and Setting Up the Best Browser for the Experiments:** Experimented with different options (toy QUIC client, Linux's `wget`, Google Chromium, & Google Chrome), solved operating-system level problems that prevented the successful launch of Google Chrome (missing dependency and lack of system bus issue in the Mininet Debian Docker image), configured Chrome to work with the self-signed SSL

certificate (because by default it does not accept self-signed certificates),

- **Experiment Automation:** Written bash shell scripts that launched multiple QUIC & HTTPS server processes and multiple Chrome processes in the background to facilitate scaling up the fairness experiment.
- **Fairness Experiments:** Ran the Fairness Experiments with all the different combinations of parameter values multiple times to obtain stable results.
- **GitHub Repository Management:** Managed and Organized the Git flow and pull requests of our GitHub repository and performed multiple refactors at different stages of the code development process.
- **Presentation:** The speaker who presented this project during the last lecture.