

Performance Analysis of QUIC Protocol under Network Congestion

by

Amit Srivastava

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2017

APPROVED:

Professor Mark Claypool, Major Thesis Advisor

Professor Robert Kinicki, Major Thesis Advisor

Professor Craig Shue, Thesis Reader

Abstract

TCP is a widely used protocol for web traffic. However, TCP's connection setup and congestion response can impact web page load times, leading to higher wait times for users. In order to address this issue, Google developed with QUIC (Quick UDP Internet Connections), a UDP-based protocol that runs in the application layer. While already deployed, QUIC is not well-studied in academic literature, particularly QUIC's congestion response as compared to TCP's congestion response which is critical for stability of the Internet and flow fairness.

To study QUIC's congestion response, we conduct three sets of experiments on a wired testbed. One set of our experiments focuses on QUIC and TCP throughput under added delay, another set compares QUIC and TCP throughput under added packet loss, and the third set has QUIC and TCP flows that share a bottleneck link to study the fairness between TCP and QUIC flows. Our results show that with random packet loss, QUIC delivers higher throughput compared to TCP. However, when sharing the same link, QUIC can be unfair to TCP. With an increase in the number of competing TCP flows, a QUIC flow takes a greater share of the available link capacity compared to TCP flows.

Acknowledgements

I am very thankful to my two advisers Prof. Mark Claypool and Prof. Robert Kinicki for their time and patience. I would not have been able to complete my thesis without their guidance. I would also like to thank Prof. Craig Shue for his valuable feedback.

I would like to thank my friends at WPI for their support. Finally, I would like to thank my parents who have always worked harder than me and always supported me in every way they could.

Contents

1	Introduction	1
2	Background	5
2.1	Transport Control Protocol (TCP)	5
2.1.1	Connection Setup	6
2.1.2	Sliding Window	6
2.1.3	Additive Increase Multiplicative Decrease (AIMD)	7
2.1.4	Slow Start (SS)	7
2.1.5	Congestion Avoidance	7
2.1.6	Fast Retransmit	8
2.1.7	Fast Recovery	8
2.1.8	Evolution of Congestion Control in TCP	8
2.2	Fairness	12
2.3	Quick UDP Internet Connections	12
2.3.1	Features	14
2.3.2	Packet Header	15
2.3.3	QUIC Packet Types	16
2.3.4	QUIC Frame Types	17
2.3.5	Setting up a Connection	18

3	Experiments	21
3.1	Testbed	21
3.1.1	Network Topology and Components	22
3.1.2	Software Tools	22
3.1.3	QUIC Client and Server	24
3.1.4	TCP Client and Server	24
3.1.5	Test Script	24
3.2	Performance Metrics	25
3.2.1	Emulating Congestion	25
3.2.2	Control Parameters	26
3.3	Experiments	26
3.3.1	Impact of Delay	26
3.3.2	Impact of Packet Loss	27
3.4	Impact of Competing Flows	27
3.4.1	Internet-based Tests	28
4	Results and Analysis	30
4.1	Data Analyzed	30
4.2	Impact of Delay	32
4.3	Impact of Packet Loss	35
4.4	Impact of Competing Flows	38
4.5	Internet-based Tests	39
5	Conclusions	41
6	Future Work	43
6.1	QUIC with Competing Flows	43
6.2	Connection Migration	44

6.3	QUIC Streams - Request Multiplexing	44
6.4	QUIC over a Wireless Network	44
7	Appendix	46

List of Figures

2.1	TCP Connection Setup	6
2.2	QUIC Packet Header	15
2.3	QUIC Connection Setup	19
3.1	Testbed for offline tests	22
3.2	Testbed for Internet-based tests	28
4.1	These graphs show throughput versus time data from three iterations of tests conducted for QUIC and TCP at 4 and 16 Mbps link capacities with an added delay of 25ms. There is little difference in the performance of TCP and QUIC at low network latencies.	33
4.2	Four throughput versus time graphs at two link capacities and 200ms of added delay and 2% packet loss	34
4.3	Average throughput from QUIC and competing TCP flows plotted against the number of competing TCP flows. Each vertical set of dots represents the share of QUIC throughput, the combined TCP throughput, the average TCP throughput and the overall throughput when a single QUIC flow runs simultaneously with TCP flows indicated by the on X axis.	37
4.4	Jain's fairness index	38

4.5	Throughput from multiple iterations of experiments with QUIC and TCP on a 16 Mbps bottleneck, 200ms of added delay and 2% loss setting	40
7.1	Throughput from experiments at various bottleneck capacities and 200ms added delay and 0% loss	47
7.2	Throughput from experiments at various bottleneck capacities, 25ms added delay and 0.5% loss	48
7.3	Throughput from multiple experiments with a 4Mbps bottleneck, 200ms added delay and 0.5% loss	49
7.4	Throughput from experiments at various bottleneck capacities, 200ms added delay and 0.5% loss	49
7.5	Throughput from experiments at various bottleneck capacities, 25ms added delay and 1.0% loss	50
7.6	Throughput from experiments at various bottleneck capacities, 200ms added delay and 1.0% loss	51
7.7	Throughput from experiments at various bottleneck capacities, 25ms added delay and 2.0% loss	52
7.8	Throughput from experiments at various bottleneck capacities, 200ms added delay and 2.0% loss	53

List of Tables

2.1	QUIC Frame Types	16
3.1	Parameters with values used in our experiments	26
4.1	Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 4 and 8 Mbps and two latency values - 25 and 50ms	31
4.2	Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 12 and 16 Mbps and two latency values - 25 and 50ms	32
4.3	Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 4 and 8 Mbps and two latency values - 100 and 200ms	35
4.4	Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 12 and 16 Mbps and two latency values - 100 and 200ms	36
4.5	Average throughput from QUIC and TCP with Standard Deviation and Standard Error from experiments with two bottleneck bandwidths - 4 Mbps and 8 Mbps	39

Chapter 1

Introduction

The Internet is a shared resource used by people across the world as a medium for exchanging data. It is essential that the links of the Internet do not fail due to excessive traffic and that Internet flows get full allocation of the capacity. Internet Service Providers (ISPs) implement network management policies at devices such as routers to throttle traffic to match the available link capacity. Since traffic on the Internet varies with time, it is difficult to ensure fair sharing among the numerous connections on a given link at any given point of time. Thus, individual machines have to manage their communication and make sure their data does not overload the network.

One such means to use the network efficiently and fairly is to use TCP (Transport Control Protocol), a network protocol that allows applications to communicate reliably on the Internet. TCP comes built-in on most popular operating systems such as Windows, Linux and MAC OS and it implements a distributed congestion control mechanism where the endpoints manage the sending rates. Thus, TCP prevents congestion collapse on the Internet by allowing dynamic adjustment to sending rates based on network congestion. Any protocol that is an alternative

to TCP must provide acceptable congestion control for wider use on the Internet, including co-operating with existing TCP flows.

With the increase in capacity of wide area networks, web traffic has increased, changing from static HTML content to include high resolution images, streaming media and interactive applications. The needs of the such traffic are quite different from the traditional static pages. Users experience relatively greater page load times if a web page contains streaming media compared to pages without it. Moreover, with increase in the use of HTTP over SSL (HTTPS) to encrypt web traffic, servers now spend additional time setting up a secure channel over a TCP connection to send traffic.

To load web pages faster at end-user browsers, major content-hosting companies move the content geographically closer to users by means of Content Distribution Networks (CDNs) and distribute the content to multiple domain names or servers, allowing multiple connections to these servers to load the content concurrently. Another means to speed up the page load is to re-utilize an open TCP connection to get more content by pipelining requests. This approach has some limitations, which result from the way TCP works, that impede the performance of certain applications.

TCP's congestion control mechanisms, such as retransmission of packets and reduction of the sending rate after packet loss, are designed to add reliability and help prevent unfair sharing of network resources. But these mechanisms can prevent applications such as web browser from performing optimally under network congestion. Google developed SPDY [spd16], a protocol that behaves similar to HTTP1.1 and later became the basis for HTTP2.0 [htt16], in order to address some issues in

the browser to speed up web page loading. In 2013, Google introduced Quick UDP Internet Connections (QUIC) [qui17] as part of its Chrome browser. QUIC operates in the application layer and uses User Datagram Protocol (UDP) from the network stack of the underlying operating system.

Since the deployment of QUIC has been Google-wide, any user accessing Google services from a Chrome or Chromium browser defaults to QUIC. There have to be thousands, if not millions, of QUIC connections on the Internet everyday sharing network links with TCP. There is, however, no data available on how QUIC behaves in congested networks. The few studies which explore QUIC, Carlucci *et al.* [CDCM15] and Megyesi *et al.* [MKM16] focus on the page load times over QUIC to HTTP1.1 over TCP under lab conditions. Lychev *et al.* [LJBNR15] focused on cryptographic mechanisms used in QUIC. But there is an absence of publicly available data on QUIC's congestion response and behavior when compared to TCP. This topic can yield useful data for understanding QUIC's impact of TCP traffic. We assume the number of QUIC based applications will continue to grow.

Our experiments explore QUIC's performance from a network congestion standpoint. Our goal was to design and conduct experiments to analyze and compare the performance of QUIC to TCP under emulated network congestion on a wired testbed. A wired testbed limits the unknowns that can impact the experiments. This allows us to explore essentially the congestion control alone under controlled conditions on a wired testbed. This study conducted the experiments in Fuller Labs at Worcester Polytechnic Institute (WPI) in Worcester, MA. We use the QUIC server from the Chromium project and download files using it. We add delay, packet loss and competing flows to study the behavior of QUIC.

The results from our experiments show that QUIC performs similarly to TCP in the absence of competing traffic for a range of values for added delay. However, QUIC delivers better throughput than TCP at high random packet loss. In the tests designed to explore fairness towards competing flows, QUIC fails to reduce its share of the available bandwidth with an increase in the number of competing TCP flows. Our real world test show QUIC's throughput being capped and rather conservative in comparison to TCP, but the share of bandwidth does not change with increase in number of flows. Thus, the presence of higher number of competing TCP flows may allow a QUIC flow to get a greater share of available bandwidth than does conforming TCP flow.

Chapter two describes how TCP works and some of the significant changes suggested or made to TCP's congestion control algorithms, followed by an introduction to QUIC, its packet structure and features. Chapter three introduces the methodology, the testbed and our choice of parameters. Chapter four presents the results, chapter five the conclusions and in chapter six, the future work. The results section has a small subset of our tests results, a much larger set of results have been included in the appendix.

Chapter 2

Background

This chapter describes, in brief, TCP connection setup and flow control mechanism followed by a brief description of some of the important updates suggested for TCP's congestion control. In the later part of the chapter, we describe packet and frame structure of QUIC along with connection setup process. A brief description QUIC's response to congestion control and packet loss is part of the last section.

2.1 Transport Control Protocol (TCP)

All versions of TCP perform connection setup, data exchange and connection shutdown in almost the same manner. This section describes these basic operations of TCP Reno. We describe Reno because most TCP versions since Reno have made small modifications to the sender side congestion control parameters rather than changing the entire flow control logic of TCP.

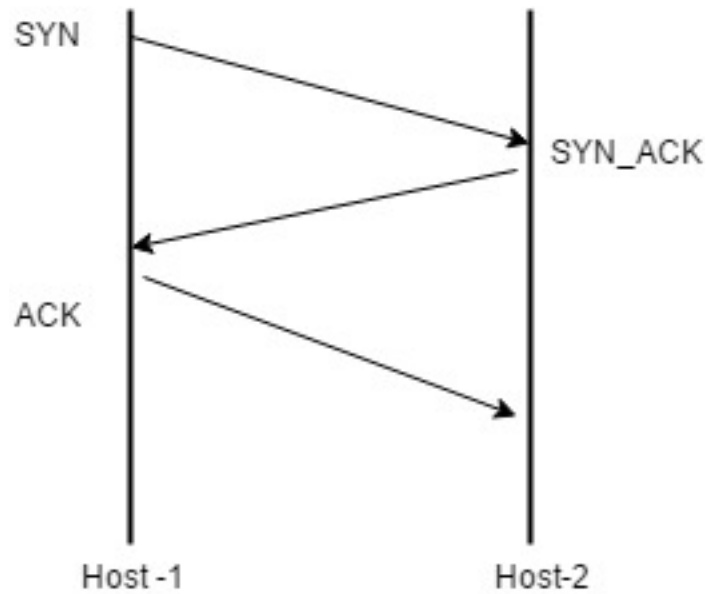


Figure 2.1: TCP Connection Setup

2.1.1 Connection Setup

The TCP connection begins with a three-way handshake. The host which initiates the connection sends a packet with SYN request to a remote host. The remote host responds with SYN-ACK packet acknowledging receipt of the SYN and agreeing to the connection. The initiator of the connection then sends an ACK acknowledging the receipt of SYN-ACK. The two end points can now exchange data.

2.1.2 Sliding Window

TCP implements a sliding window algorithm at the sender. A window refers to the amount of data, in bytes or packet count, that can be sent before an acknowledgment is received. The sender sends a window's worth of data and waits for an acknowledgment from the receiver of data before sliding the window forward and sending more data. This prevents the receiver and the network from getting overwhelmed by data. This window also allows the sender to keep track of the data successfully received by the

client.

2.1.3 Additive Increase Multiplicative Decrease (AIMD)

AIMD is the main algorithm for governing the sending rate on a TCP connection where the sender increases the size of the window by a constant linear value for every acknowledgment received and reduces the congestion window by a multiplicative factor in response to packet loss. This strategy may appear conservative but the actual behavior is modified by the choice of values for these two parameters.

TCP assumes a packet is lost if it is not acknowledged within a certain time frame, and the sender resends the packet after adjusting the congestion window to account for packet loss. We next describe TCP Reno and the New Reno update on handling retransmissions. Generally, all versions of TCP use similar mechanism for retransmissions and only vary in the AIMD parameters or the mechanism to react to congestion which could be loss-based, delay-based or a combination of both.

2.1.4 Slow Start (SS)

Slow start is a mechanism employed by TCP to prevent a connection from taking away bandwidth from other connections on the network. In slow start, the sliding window increases exponentially in size as well as moves forward with each acknowledgment received.

2.1.5 Congestion Avoidance

When the sender's congestion window ($cwnd$) grows beyond a value called *ssthresh* then the connection switches from the Slow Start to the Congestion Avoidance phase.

During congestion avoidance, the congestion window increases by an additive factor expressed in segment size and gets reduced to half the current size (in bytes or segments), if a packet loss is detected.

2.1.6 Fast Retransmit

If a packet is lost and the subsequent packets reach the receiver, then for each out of order packet received, the receiver sends an ACK with the next expected sequence number which did not arrive. The Fast Retransmit mechanism allows the sender to send the missing packet when three duplicate ACKs are received, rather than wait for the timeout to occur. Additionally, the *ssthresh* is reset set $(cwnd/2) + 3$ before the retransmission occurs as per RFC5681 [MAB09].

2.1.7 Fast Recovery

Sally Floyd *et al.* describe the Fast Recovery update to TCP Reno based on suggestions from Janey Hoe's [Hoe95] work and call it the New Reno update. Fast Recovery allows the *cwnd* to expand by one segment size for every duplicate ACK after the third received until we get a non-duplicate ACK. At this point the inflated *cwnd* is reset to a smaller value to represent the packet loss and the subsequent window decrease.

2.1.8 Evolution of Congestion Control in TCP

The way TCP responds to congestion on the network has undergone changes for over 25 years. The increase in the capacity of physical medium has forced TCP to update its congestion control mechanism to take advantage of high capacity on wide area networks and to share links with other TCP connections in a fair manner.

L. Brakmo *et al.* [BP06] proposed TCP Vegas, which aims to make a TCP connection more sensitive to the transient changes in the bandwidth by maintaining a 'correct' amount of packets on the wire, with the aim of not keeping packets queued at a bottleneck buffer along the path. TCP Vegas employs three techniques to improve throughput and reduce loss- one results in timely retransmits, another allows congestion detection and adjusts transmission rate, and the last technique modifies the slow start. For timely retransmissions the time elapsed since a packet was sent is calculated using ACKs and duplicate ACKs. Thus, even if less than three duplicate ACKs are received, a re-transmission can occur using the timeout. To detect congestion, Vegas uses the difference between the actual and expected bandwidth. This difference (Diff) if larger than a certain threshold. For this calculation Vegas defines a BaseRTT or the smallest RTT over the lifetime of the connection. Vegas defines alpha and beta as thresholds. If the $Diff < \alpha$ or $Diff < \beta$, the congestion window is increases linearly. If $\alpha < Diff < \beta$, the congestion window remains unchanged.

Sally Floyd *et al.* [Flo03] proposed HighSpeed TCP (HSTCP). HSTCP does not modify TCP's response under heavy congestion. It is a sender side modification to allow large congestion windows ($cwnd$) and prevent loss from shrinking $cwnd$ on high capacity links. The congestion window is related to loss rate by the formula $\sqrt{cwnd} = 2/2p$, where p is the loss rate. HSTCP defines three new variables, two for congestion window and one for loss rate. These are, High_Window, Low_Window and High_P. The value for the variables can be set based on the maximum throughput desired. The authors chose 83,000 as High_Window and High_P to be 10^{-7} , which meant throughput of 10 Gbps at a packet loss rate of 10^{-7} . The authors opine that for

compatibility with TCP the response of the new function should be similar to TCP Reno at loss rates between $10^{-3} - 10^{-1}$. For lower loss rates, such as 10^{-7} , HSTCP can deviate from Reno [Flo03].

Tom Kelly proposed Scalable TCP [Kel03] to improve TCP's performance on high capacity links. Scalable TCP updates the sender side congestion control algorithm such that the resulting TCP connection is compatible with TCP Reno flows on the network. The STCP adds a new parameter called the Legacy Window (*lwnd*). Legacy window denotes the congestion window size needed to achieve a given sending rate under given loss rate (called the legacy loss rate) by TCP Reno. The congestion control algorithm uses TCP Reno when $cwnd \leq lwnd$, and switches to scalable congestion window when $cwnd \geq lwnd$. With the value chosen for *lwnd* being 16 packets at a 1500 byte segment size. The AIMD parameter for window increase increments the window size by 1% for every acknowledgment when no congestion is detected and reduces the window size by 12.5% in the event of a congestion. Scalable TCP uses Legacy window to be fair to TCP at lower throughputs while allowing faster scaling of the window at higher link capacities.

FAST TCP [WJLH06] was developed to allow TCP to perform well at large window sizes. FAST TCP uses estimation of queuing delay and average RTT to adjust the congestion window. FAST TCP defines an equilibrium point and the congestion window growth is slower near the equilibrium and faster away from it. Here all the senders sharing a bottleneck try to maintain an equal number of packets in the queue. However, results from experiments show that FAST TCP connections with higher RTT experience higher queuing delays than connections with lower RTTs.

Kun Tan *et al.* proposed Compound TCP [TSZS06] as an improvement over TCP Reno so that TCP flows can utilize the higher capacity links on long distance connections such as optical fiber cables. The authors state that loss-based algorithms are aggressive as they fill up queues at network devices, only to slow down when packet drops occur at the bottleneck queue. The delay-based algorithms, however, respond to the increase in RTT when packets get queued by decreasing the sending rate. When used together, delay-based TCP flows lose bandwidth to loss-based TCP flows. In their opinion, the solution was to combine both by adding a delay window to the congestion window at the sender. The addition of the delay window allows more packets to be in flight. The delay window is added when the difference between actual and expected throughput is greater than a threshold value. At such a time the network is more congested.

E. Kohler *et al.* proposed Datagram Congestion Control Protocol (DCCP) [EKF06] as a UDP-based protocol designed primarily for applications that require timeliness over reliability. Examples of such applications include telephony and streaming media and other Internet-based applications. DCCP does not add reliability, only congestion control. It offers two mechanisms for congestion control denoted by Congestion Control IDs (CCID). The first is called TCP-like congestion control or CCID-2 and second mechanism is the TCP Friendly Rate Control (TFRC) or CCID-3. The acknowledgments used in DCCP use packet numbers and not data offset as used by TCP Reno.

H. Sangtae *et al.* [HRX08] proposed CUBIC, the current default congestion control algorithm on Linux. CUBIC is named because of a cubic function used to grow the congestion windows during congestion avoidance phase. This function

allows faster increments to the congestion window compared to New Reno when the difference between $cwnd$ size and $ssthresh$ is large and smaller increments when $cwnd$ size is closer to $ssthresh$. This behavior reduces sudden changes in sending rate close to a previous maximum rate but allows bigger increments to $cwnd$ if more capacity is available. CUBIC compares current $cwnd$ to TCP Reno's W_{TCP} to determine the current operating region. If $cwnd < W_{TCP}$ then the protocol is in the TCP region. A CUBIC connection is in concave region when $cwnd < W_{max}$ and convex region if $cwnd > W_{max}$. Concave region signifies window growth towards a maximum window size prior to a loss event, while convex region signifies probing for new maximum $cwnd$ in the absence of packet loss. In both concave and convex regions, the window increments depend on the RTT value along with absence of packet loss. For packet loss the window reduction uses $\beta = 0.2$ rather than 0.5 used in TCP Reno.

2.2 Fairness

Jain *et al.* [JCH84] proposed the Index of Fairness that was independent of the population size and the unit of measurement. The index would show small change in the the resource allocation.

$$f(x) = \frac{[\sum_{n=1}^N x_i]^2}{\sum_{n=1}^N x_i^2} \quad (2.1)$$

2.3 Quick UDP Internet Connections

QUIC stands for QUIC UDP Internet Connections [qui17b]. This is a new protocol in the application space that uses the UDP from the operating system below and adds its own set of features on top. These features mainly include reliability similar to TCP and congestion control where QUIC uses CUBIC similar TCP but also

supports other mechanisms.

This chapter describes the format of QUIC packet, components of the packet header with their purpose and how a QUIC connection is setup, used and torn down. QUIC has both regular and special packets. The special packets are used during the initial negotiation between a client and server on the version of QUIC that will be used on the connection and the encryption that will be used.

QUIC adds its own header to the QUIC payload and then encapsulated it inside a UDP datagram before sending it. The payload is encrypted thus it is not possible for anyone tracking the packets to know the contents of the payload.

QUIC is a multiplexed protocol which means that multiple requests can be sent over the same QUIC connection. To differentiate the payload based on the sender receiver pair QUIC adds frames. These frames have a unique stream-id that helps the receiver determine to which endpoint is the data in a QUIC frame is headed.

Our interest in QUIC stems from the desire to understand the implementation of QUIC's congestion control mechanism, which by default is said to be CUBIC, same as TCP's default mechanism on current Linux distributions. Since the only research work and publicly available data on QUIC comes from Carlucci *et al.* [CDCM15] and Megyesi *et al.* [MKM16] both of whom primarily analyzed web page load times.

2.3.1 Features

Some important features of QUIC include:

Connection Establishment Latency

QUIC combines the cryptographic and transport handshakes to reduce the number of round trips needed to setup a connection. QUIC introduces a client cached token that can be re-used to communicate with a server that has been seen before. This reduces the need for a new handshake.

Multiplexing

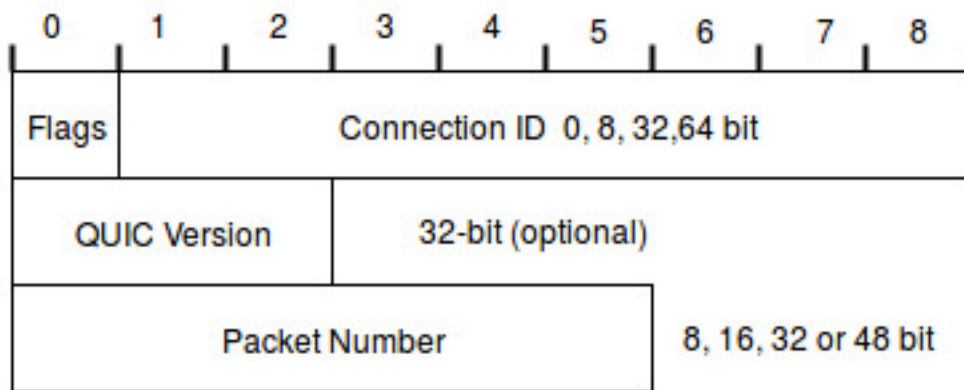
A QUIC connection consists of streams carrying data independent of one another. The data for each stream is sent in a frame identified by a stream ID. A QUIC packet can thus be composed of one or more frames.

Forward Error Correction (FEC)

QUIC supports FEC, where a FEC packet would contain parity of the packet that form the FEC group. This feature can be turned ON or OFF as necessary. This allows recovering contents of a lost packet in a FEC group.

Connection Migration

A QUIC connection is identified by a 64 bit connection ID rather than a 4-tuple of source and destination IP address and port numbers of the underlying connection. Thus, a QUIC connection can be reused if IP addresses or Network Address Translation (NAT) bindings change when, for example, a device changes its Internet connection. QUIC allows cryptographic verification of a migrating client and thus the client continues to use a session key for encrypting and decrypting packets



QUIC Header Structure

Figure 2.2: QUIC Packet Header

2.3.2 Packet Header

Figure 2.2 shows the public header of a QUIC packet. The header contains Public Flags of size 8 bits. The bits can be set to allow QUIC version negotiation between the two end-points, indicate the presence of Connection ID and indicate a Public Reset packet. The Connection ID is an unsigned 64 bit random number selected by the client. Its value does not change for the duration of a single connection. Connection ID can be omitted from the packet header when the underlying 4-tuple of IP address and port numbers for client and server do not change.

The sender assigns each regular packet a packet number, starting from 1. Each subsequent packet gets a number that is one greater than the previous packet. The 64 bit packet number is part of a cryptographic nonce. But the QUIC sender only sends at most the lower 48 bits of the packet number. To allow unambiguous reconstruction of the packet a QUIC end point must not transmit a packet whose number is larger by $2^{\text{bitlength}-2}$ than the largest packet acknowledged by the receiver. Therefore, there can be no more than 2^{48-2} packets in flight.

Regular	Special
PADDING RST STREAM CONNECTION CLOSE GOAWAY WINDOW UPDATE BLOCKED STOP WAITING PING	STREAM ACK CONGESTION FEEDBACK

Table 2.1: QUIC Frame Types

2.3.3 QUIC Packet Types

Here we describe the QUIC packet and types.

Special Packets

1. Version Negotiation Packet: The version negotiation packet begins with 8 bit public flags and 64 bit Connection ID. The rest of the packet is a list of 4 byte versions that a server supports.
2. Public Reset Packet: A public reset packet begins with 8 bit public flags and 64-bit Connection ID. The rest of the packet is encoded and contains tags

Regular Packets

Beyond the public header, all regular packets are authenticated and encrypted, and referred to as Authenticated and Encrypted Associated Data (AEAD). This data when decrypted consists of frames.

1. Frame Packet : It contains the application data in the form of frames that contain type information and payload.

2. FEC Packet : It contains the parity bits from XOR of null-padded payload from the Frame packets in a FEC group. QUIC frames are of two types- special frames and regular frames. We describe some important frames types that will help the reader to understand how a QUIC connection is setup and how data is sent from one end-point to another.

2.3.4 QUIC Frame Types

Here we describe the QUIC frames and types.

Stream Frame

This frame is used to initiate a new stream on an existing connection and also to send data for an existing stream. The header consists of 1 byte type field, stream id (1, 2, 3 or 4 byte long) a variable length offset stream of up to 8 bytes and data length of non-zero value.

ACK Frame

This packet is sent to inform the peer of the packets that have been received and those which are still considered missing. This is different from TCP's SACK in that it reports the largest packet number received followed by the a list of missing packet, or NACK, ranges.

Stop Waiting Frame

This frame is used to inform the peer that it should not wait for packet numbers lower than a specified value. This packet number can be encoded using 1, 2, 4 or 6 bytes.

Window Update Frame

This frame is used to inform the peer about increase in the flow control receive window size. The stream ID can be 0 for this frame indicating that the update is applicable for the entire connection rather than a particular stream. The header Window Update frame consists of a 1 byte Frame Type field and a up to 4 bytes of stream ID.

2.3.5 Setting up a Connection

A QUIC connection begins with a client sending a handshake request using a CHLO packet to the server. If the client and server have not previously communicated with one another, the server creates a cryptographic token for the client. This token is opaque to the client but, for the server, the token contains the IP address used by the client to send this initial request. The token is sent to the client in a REJ or reject packet. The client now uses the token to encrypt the HTTP request to the server and the server sends an encrypted response to the HTTP request of the client.

The next time the client contacts the same server it uses the token provided by the server to send an encrypted HTTP request. This saves time in setting up connections. The transport layer and encryption handshake are combined into one process. Reusing the key or token minimizes the need for a handshake to a given server. Further details about the token and QUIC crypto is available at [qui17c].

Loss Recovery And Congestion Control

The sequence numbers used in TCP increase the data offset in each direction. The sequence number for QUIC increases monotonically. When the packets are lost

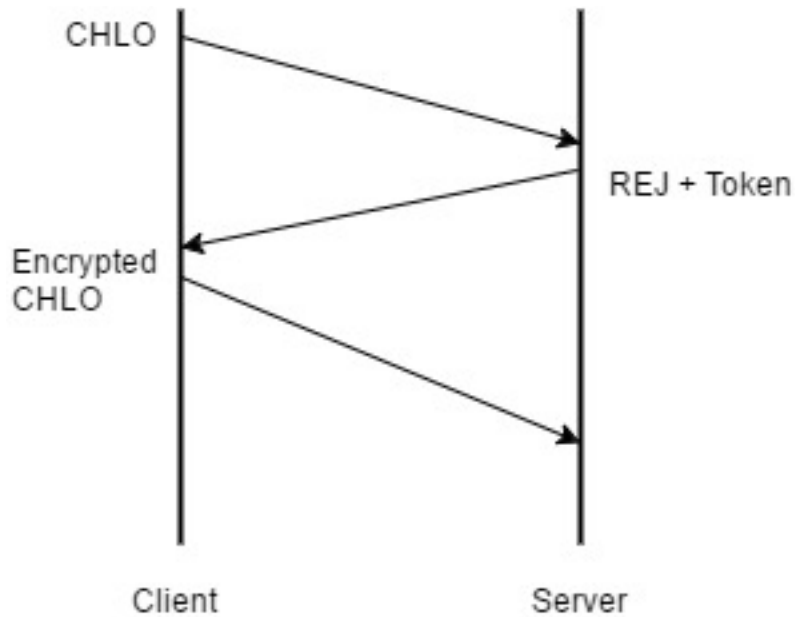


Figure 2.3: QUIC Connection Setup

for a TCP connection, keeping track of sequence numbers requires a non-trivial implementation. In case of QUIC, the sequence numbers are not repeated and the data is sent with new a sequence number. This allows easy loss detection.

After sending a packet a timer may be set based on:

- if handshake is incomplete, start handshake timer
- if there are packets that are NACKed, set loss timer
- if fewer than two Tail Loss Probes (TLP) have been sent, start TLP timer

On receiving an ACK the following steps are performed:

- Validate the ACK
- Update RTT measurements
- Mark NACK listed packets with sequence number smaller than the largest ACKed sequence number as missing

- Set a counter with threshold value 3 for each NACK listed packet
- NACKed packets with counter $>$ threshold are set for retransmission

Chapter 3

Experiments

This chapter describes the testbed and the experiments designed to study the congestion response of the QUIC protocol and to compare it with TCP. We first define the testbed used for our experiments. We then discuss the parameters used in the experiments in the following chapters. After an overview of the network topology and the parameters used, we describe the performance metrics. We then describe how to read the graphs.

3.1 Testbed

For experiments, we use the test setup shown in Figure 3.1. For experiments that require Internet access we use a part of the same testbed, shown in Figure 3.2. The topology consist of two Ethernet switches capable of gigabit speeds and five desktop PCs running Ubuntu 14.04 LTS. The desktop in the middle labeled as emulator has two network interfaces. The interface B is where we add delay and packet loss. Interface B is also the interface we use for traffic capture.

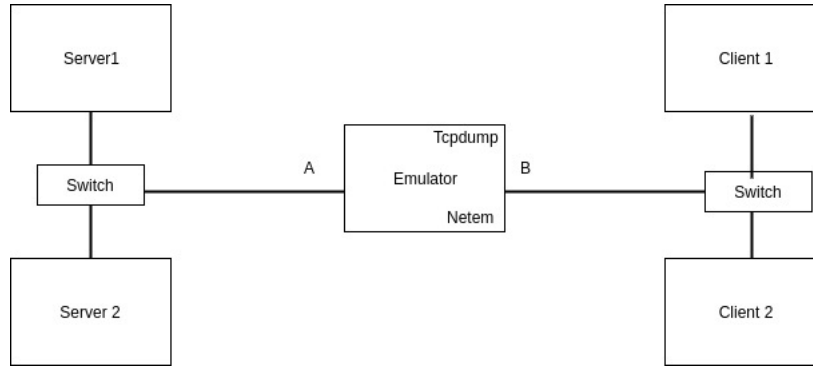


Figure 3.1: Testbed for offline tests

3.1.1 Network Topology and Components

The testbed has a dumbbell-shaped topology. This testbed simulates traffic arriving at a network device, such as a router, from different interfaces and leaving from one interface. Before running experiments on the testbed, we setup network addresses and routes on the desktop PCs to allow transfer of data from the servers to the clients via the emulator. We also setup the network parameters for each test.

3.1.2 Software Tools

This section describes the tools that we used to setup the routes, capacity on the bottleneck link and traffic capture.

Netem [net16] provides network emulation for testing purposes on Linux. Netem can add delay, drop packets, change order of packets and send duplicate packets. We use Netem to add delay and induce random packet loss.

Tcpdump [tcp16] is a Linux utility that monitors or records traffic passing through an interface. Tcpdump allows filtering of traffic based on network layer protocol, IP address or port number among many other features. We use tcpdump to capture

traffic and we do so using the verbose mode where the IP address is not resolved to a url. An example of the command we use is:

```
$tcpdump -c 80000 -nnvv '((tcp) or (udp))' -i eth7 -w capture.pcap
```

The command above captures only the first 80,000 packets of TCP or UDP traffic on interface eth7 and stores it in a pcap file format. 'nnvv' allows more details to be captured and prevents resolution of IP addresses.

Iperf [ipe16] is a tool for measuring the maximum achievable bandwidth on the networks. Iperf allows tuning the parameters such as timing, protocol and buffers. In our experiments, we use iperf to measure the achievable bandwidth that can be achieved after a bottleneck bandwidth has been set. We use iperf3, the latest version of iperf. We ran the iperf client on server1 and server2 and iperf server on client 1 and client2 desktops. By default, the iperf server listens on port 5001. An example of the commands we use:

```
$iperf3 -c 192.168.2.20
```

```
$iperf3 -s
```

Route [rou16] manipulates the Linux kernel's IP routing table. We use route command to add routes on the desktops to send data through the emulator interfaces. The route has been obsoleted by the new ip route. An example of the route command to set default routes is:

```
$route add default gw 192.168.1.1 eth0
```

Tc [ipe16] is used to configure traffic control in the Linux kernel. Tc is capable of traffic shaping, scheduling and policing. Tc uses queuing disciplines associated with network interfaces that hold packets before sending them out. Thus, most

modifications can only be done on outgoing traffic. For our experiments, `tc` helps to set the bottleneck capacity on the interface B. An example of `tc` command used to set maximum link capacity is:

```
$ tc qdisc add dev eth7 root handle 1: tbf rate 4mbit burst 1600 limit 50000b
```

3.1.3 QUIC Client and Server

The QUIC server and client application are part of the Chromium Project. This code is different from the code used on the Google servers and the Chrome browser. Further, [qui17a] adds a caveat that the code is for integration testing and not meant to perform at scale. Since we analyze congestion control, we assume that the algorithm and its parameters will not differ significantly from any real world implementation. Carlucci *et al.* [CDCM15] also instrumented the same code base in their study of QUIC, thus we consider it to be useful to derive insights into QUIC's performance in the absence of an open-source implementation.

3.1.4 TCP Client and Server

The client and server application for TCP flows are written in C. The server listens on a hard-coded port number and the client connects to the same port.

3.1.5 Test Script

Manually running the tests can be time-consuming and sometimes introduce errors and inaccuracies. Therefore, a Python script was used to set the network parameters and record traffic at the emulator and initiate file download at the two client desktops. The script initiates and terminates tests along with cleanup after each test run. This

script resides on the client-1 machine.

3.2 Performance Metrics

The focus of this study is to explore the response of QUIC to congestion on the network. We use throughput to measure the performance of QUIC when compared to TCP. We do not compare the congestion window size of QUIC and TCP in our experiments because the test code is different from the QUIC implementation on Google production servers and the congestion window is limited in the test code. Thus, using throughput of a flow at a middle box allows us to compare how QUIC and TCP flows differ under same network conditions.

3.2.1 Emulating Congestion

Network congestion is created when the amount of incoming packets on a network device is more than the rate at which packets can be sent out. This causes the buffers associated with network interfaces to fill up with packets waiting to be sent out. These buffers are limited in size and beyond a certain incoming packet rate at an interface, all packets get dropped. We emulate congestion by using a queue size equal to the bandwidth delay product using the bottleneck capacity and a delay of 100 milliseconds. Table 3.1 provides information about the values we chose for delay, loss, bottleneck link capacity and competing flows. The bottleneck link capacity, delay and loss values are similar to those used by Carlucci *et al.* [CDCM15] in their evaluation of QUIC.

While our experimental results may not be the same as an actual enterprise router, they provide an insight into QUIC’s congestion response. The results can be

Delay (ms)	25	50	100	200
Loss (%)	0.0	0.5	1	2
Capacity (Mbps)	4	8	12	16
TCP Flows	1	2	4	8

Table 3.1: Parameters with values used in our experiments

used to infer QUIC’s performance in the real world.

3.2.2 Control Parameters

To create congestion we vary the following parameters on our testbed.

- Latency : Add one-way delay to packets that are going from a server to client.
- Packet Loss : Drop packets that are going from a server to client.
- Bottleneck Capacity : Limiting the maximum link bitrate.
- Competing flows : Adding additional flows in the same direction as a QUIC flow.

3.3 Experiments

A Python script on Client-1 executes the test cases while the Internet based tests were not automated.

3.3.1 Impact of Delay

To understand the impact of queuing delay on QUIC we add one-way delay to the traffic from the server to the client machines. Since delay is added in one direction, we only add a buffer or queue of size $B = T \times C$ at the bottleneck interface rather

than $2 \times T$, where B is the buffer size in bytes, C is capacity of the link and T is the delay in seconds, needed to send packet one way. Thus, traffic from client to server takes an average time of about 350 micro seconds.

Each delay-based experiment begins by setting link capacity and one-way delay at the emulator interface B. After that, traffic capture is initiated using tcpdump followed by the file download. After the file transfer is complete, the script waits for five seconds and resets the parameters at the emulator. The traffic capture does not include all the packets that are part of the file transfer; we stop after about 90% of packets going from server to client have been captured. QUIC and TCP send a different number of packets, with QUIC sending more packets than TCP.

3.3.2 Impact of Packet Loss

Just as is the case with induced delay, each test with induced loss begins by setting the parameters for link capacity, delay and loss at the emulator. The packet loss is expressed as a probability in netem, such that every outgoing packet at interface B has a N% chance of being dropped, where N is the value selected using netem. We consider packet loss at all latency values we selected for the delay based tests.

3.4 Impact of Competing Flows

The use of competing TCP flows is of particular interest, since QUIC will need to coexist with TCP in the real world. Since most of our tests emulate congestion for a single flow they may be limited in their application to a larger and more diverse set of network conditions. Testing with competing flows allows us to study the response of congestion control in QUIC to TCP traffic under varied network load. In this

scenario TCP will also influence QUIC’s perception of available bandwidth, the resulting bandwidth share will determine the fairness of QUIC towards TCP.

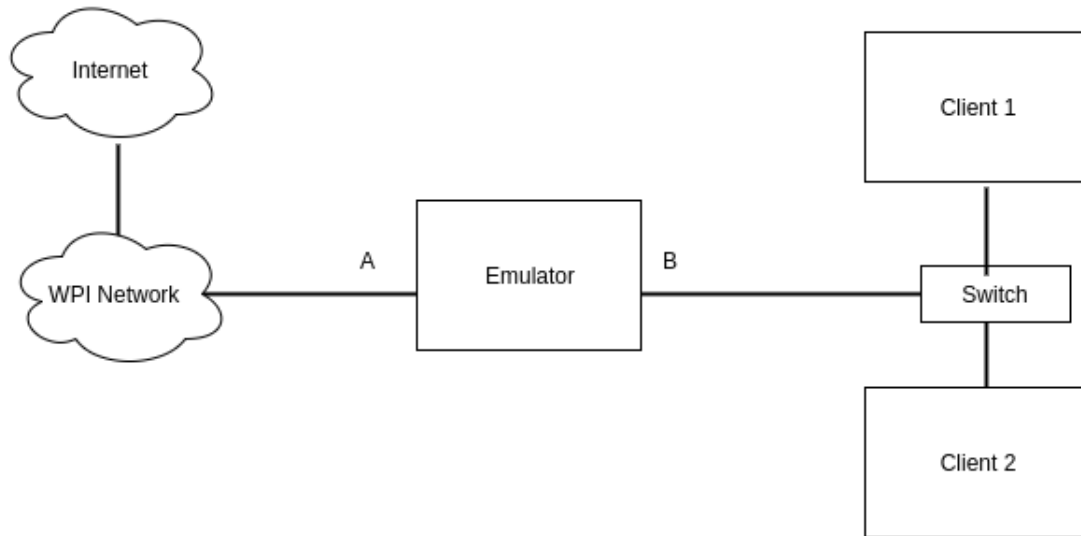


Figure 3.2: Testbed for Internet-based tests

3.4.1 Internet-based Tests

We want to test the QUIC protocol available on the Internet and to determine QUIC’s response to competing TCP flows. For this purpose, we download files from Google Drive. Thus, both the end-points use the latest version of QUIC (version 36) which is much more recent than the version we were using previously (version 25). When using TCP to download files we ensure that the delay experienced by TCP flow is same as that of QUIC flow. This ensures that the RTT value used by CUBIC in both TCP and QUIC is similar during bandwidth estimation. The only difference would be the packet loss rates. The bottleneck bandwidth was set to 16 Mbps and no packet loss was added at the emulator.

We modify the testbed by removing the Servers 1 and 2 and connect the re-

maintaining setup to the WPI wired network for Internet access as shown in Figure 3.2. Three different browsers are used in these experiments - Chrome, FireFox and Opera. Chrome provides a single QUIC flow between Google Drive and Client-1. And the other two browsers provide one TCP flow each to Google Drive.

We have to use different browsers in order to generate multiple TCP flows, otherwise, HTTP multiplexes multiple file download requests over a single TCP connection. In these test we could not achieve four or eight competing flows for this reason.

Only one or two competing TCP flows were used against a QUIC flow. Again, we emphasize that the scale of this experiment is not similar to an enterprise router running thousands of flows, but the underlying logic at the protocol level is the same. The browsers were downloaded from their respective websites. The same file is downloaded from all the browsers, no script was used to automate this experiment.

Chapter 4

Results and Analysis

This chapter provides results from our experimental evaluation of QUIC. The graphs depict the results that show major deviation from TCP-like behavior. This allows the reader to gain useful insight in QUIC's congestion response. The data from all the tests are summarized by the tables in this chapter.

The sections in this chapter follow the same sequence as their description in Chapter 3. We begin by describing the results from experiments with added delay, then added loss, competing flows and finally the results from Internet-based tests.

4.1 Data Analyzed

The graphs presented in this chapter are throughput versus time graphs. All experiments were repeated five times, but for readability, the graphs included in the results only show data from three runs under the same set of parameters. The X-axis represents time in seconds, with the data at least 20 seconds into a file transfer where possible. This is done in order to study the results at steady state. This helps to remove variability in the behavior of TCP or QUIC during the initial few seconds

when TCP or QUIC expand the congestion window to the maximum possible value permitted by the network conditions or the congestion control algorithms.

The complete information from the experiments such as average throughput, standard deviation and standard error of mean for throughput is available in the form of tables. The throughput values in Tables 4.1 through 4.4 are calculated 30 seconds after the start of file transfer and for a 20 second interval. Therefore, we do not include the start nor end of any file transfer in the graphs.

Bottleneck Bandwidth Mbps	One way Delay ms	Packet Loss %	QUIC Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean	TCP Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean
4	25ms	0	3,879,096	47.68	15.08	3,825,464	12.85	4.86
		0.5	3,878,496	52.13	15.05	3,773,696	2,443.81	736.84
		1.0	3,878,536	65.76	18.98	3,517,576	7,366.91	2,221.21
		2.0	3,866,544	1,299.27	391.74	2,697,104	9,788.43	3,095.37
8	25ms	0	7,758,360	55.90	18.63	7,650,944	24.33	8.60
		0.5	7,754,824	755.27	209.47	5,542,352	34,650.91	8,946.83
		1.0	7,670,056	14,735.02	4,086.76	3,961,408	25,873.86	8,624.62
		2.0	6,596,272	107,426.67	31,011.41	2,817,920	11,004.30	3,479.87
4	50ms	0	3,859,072	7,098.35	2,366.12	3,825,408	12.90	4.56
		0.5	3,878,640	61.79	17.14	2,850,752	8,668.31	3,876.59
		1.0	3,871,112	1,105.36	333.28	2,079,584	14,139.02	4,713.01
		2.0	3,573,936	39,897.73	12,029.62	1,725,344	99,259.85	35,093.66
8	50ms	0	7,758,256	56.60	21.39	7,651,080	9.07	4.05
		0.5	7,710,728	19,503.30	5,630.12	2,882,568	16,029.88	6,544.17
		1.0	7,054,712	107,708.44	28,786.29	2,048,808	5,104.05	1,929.15
		2.0	5,211,256	209,321.01	60,425.77	1,496,152	9,393.27	3,131.09

Table 4.1: Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 4 and 8 Mbps and two latency values - 25 and 50ms

Bottleneck Bandwidth Mbps	One way Delay ms	Packet Loss %	QUIC Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean	TCP Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean
12	25ms	0	11,637,664	54.00	19.09	11,476,680	29.06	11.86
		0.5	10,982,328	277,611.83	71,679.07	5,517,464	24,667.91	8,721.42
		1.0	11,169,384	111,782.29	29,875.07	3,979,040	19,857.92	8,880.73
		2.0	10,414,520	286,574.33	95,524.78	2,753,000	10,853.31	4,853.75
16	25ms	0	15,516,808	26.65	8.43	15,302,168	15.28	6.84
		0.5	15,360,648	59,970.22	17,311.91	5,815,536	51,609.90	25,804.95
		1.0	14,547,232	236,119.23	71,192.63	3,949,176	21,747.74	8,219.87
		2.0	12,730,592	486,936.33	140,566.41	2,739,704	7,183.25	2,932.55
12	50ms	0	11,637,496	69.78	26.37	11,476,496	11.71	5.24
		0.5	11,637,040	46.84	15.61	2,936,712	22,170.67	9,051.14
		1.0	10,208,680	272,519.02	86,178.08	2,114,544	6,038.02	2,134.76
		2.0	9,380,888	422,656.21	127,435.64	1,465,448	6,494.80	2,454.80
16	50ms	0	15,516,976	64.61	22.84	15,302,344	24.18	12.09
		0.5	15,090,864	150,824.41	41,831.16	3,047,072	21,015.97	7,430.27
		1.0	13,200,880	527,822.70	175,940.90	2,153,712	9,698.57	3,428.96
		2.0	13,448,920	456,912.71	144,488.49	1,488,560	4,493.49	1,420.97

Table 4.2: Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 12 and 16 Mbps and two latency values - 25 and 50ms

4.2 Impact of Delay

Figure 4.1 consists of four throughput versus time graphs that show the throughput achieved by TCP and QUIC flows at steady state for two bottleneck capacities of 4 and 16 Mbps with a fixed one-way delay of 25ms. Each line on the graph represents one iteration of the test. Figures 4.1(a) and 4.1(c) show an overlap throughput achieved at steady state by QUIC in multiple iteration of our delay-based tests. Thus, QUIC is consistent in its bandwidth estimation and similar to TCP in this case.

When we add delay to the traffic from the server to the client, both TCP and QUIC are able to utilize the available bandwidth. The payload for QUIC packets is smaller than the payload of TCP. Thus, in Figures 4.1(c) and 4.1(d), we observe that TCP flows terminate faster than QUIC flows.

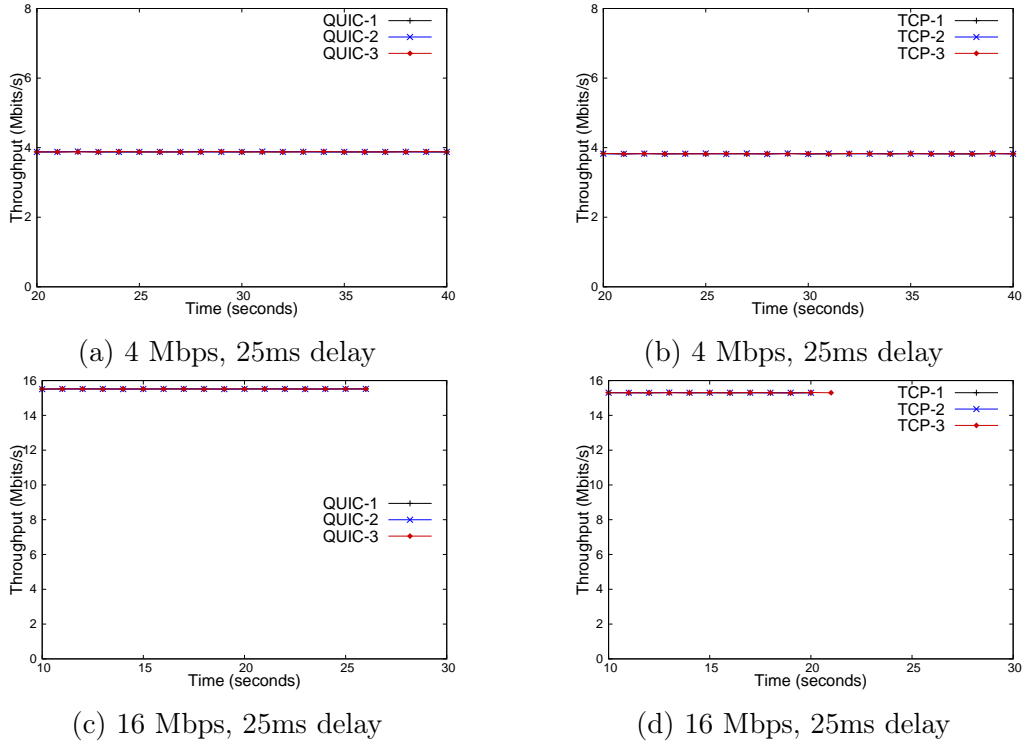


Figure 4.1: These graphs show throughput versus time data from three iterations of tests conducted for QUIC and TCP at 4 and 16 Mbps link capacities with an added delay of 25ms. There is little difference in the performance of TCP and QUIC at low network latencies.

From Figures 4.1(a) and 4.1(b), QUIC and TCP achieve similar throughput over small bandwidth links with low latencies such as 25ms. But QUIC delivers a higher throughput by a small fraction. Figures 4.1(c) and 4.1(d) have similar result, but at a higher bandwidth of 16 Mbps. Figures 4.1(c) and 4.1(c) also show a longer time to taken by QUIC to download a file compared to TCP. Our analysis is primarily concerned with the response to congestion alone.

From the data in Tables 4.1 and 4.2, we observe that QUIC achieves a greater throughput than TCP by about 1.4% across all bottleneck capacities with added delay of 50ms or less in the absence of induced packet loss on the testbed. The average packet size for a QUIC packet is smaller than a TCP packet. This implies

that the congestion window for QUIC would contain a higher packet count for the same window size in bytes as TCP. That is more packets in the queue at a bottleneck buffer and potentially higher packet loss rates.

Table 4.1 provides the results from experiments conducted using bottleneck bandwidths of 4 Mbps and 8 Mbps at various values for one way delay. As mentioned earlier, the queue size at the bottleneck buffer was kept at 1x times the bandwidth delay product, with delay being 100ms. McKeown *et al.* [BGG⁺08] show that a much smaller buffer size can be used even on enterprise devices but we set the bottleneck queue to $B = T \times C$ based on RFC 3439 [BM02].

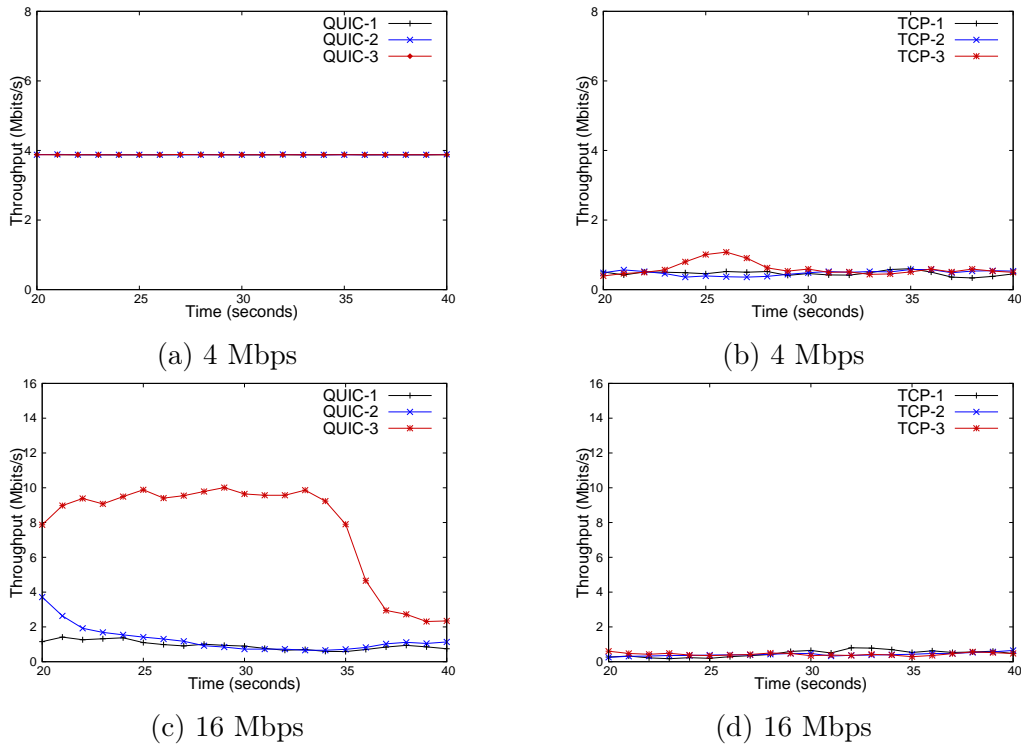


Figure 4.2: Four throughput versus time graphs at two link capacities and 200ms of added delay and 2% packet loss

Bottleneck Bandwidth Mbps	One way Delay ms	Packet Loss %	QUIC Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean	TCP Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean
4	100ms	0	3,879,016	67.15	25.38	3,825,504	12.27	7.09
		0.5	3,878,640	62.09	17.22	1,816,864	90,610.70	28,653.62
		1.0	3,759,264	49,404.83	14,261.95	1,160,784	6,443.06	1,942.66
		2.0	2,768,840	123,761.84	37,315.60	796,240	2,935.47	757.94
8	100ms	0	7,758,344	56.70	20.05	7,651,120	15.38	7.69
		0.5	6,023,072	228,616.12	61,100.23	1,586,200	12,306.70	3,177.58
		1.0	4,697,968	323,391.44	93,355.07	1,167,936	5,722.03	1,651.81
		2.0	3,363,360	310,265.75	98,114.64	1,292,088	220,502.31	58,931.72
4	200ms	0	3,879,016	61.02	23.06	3,825,464	13.07	5.34
		0.5	3,878,640	62.95	17.46	1,180,160	14,810.17	3,958.18
		1.0	3,279,112	110,400.79	31,869.96	730,976	4,526.84	1,255.52
		2.0	3,562,424	111,704.53	37,234.84	479,264	2,337.44	674.76
8	200ms	0	7,758,008	132.96	47.01	7,630,184	5,215.63	2,332.50
		0.5	6,218,776	257,683.68	71,468.59	1,191,760	16,463.83	4,250.94
		1.0	4,593,960	390,808.17	112,816.60	778,128	10,551.71	2,820.06
		2.0	3,855,760	388,395.16	112,120.03	464,976	1,614.94	466.19

Table 4.3: Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 4 and 8 Mbps and two latency values - 100 and 200ms

4.3 Impact of Packet Loss

In Figure 4.2(c), QUIC throughput shown in red, falls after what seem to be a multiple loss events within a single RTT. This response is not consistent. In Figure 4.2(c), the flows shown in blue and black at the bottom of the graph initially achieved a throughput value closer to the bottleneck bandwidth. However, for these two flows, the congestion window reduction happened much earlier and hence the reduced sending rates. TCP flows were more consistent in the observed throughput across tests for a given packet loss rate. QUIC flows ignored packet loss and used a greater share of uncontested bandwidth available despite the packet loss. This was true even for tests with high packet loss probabilities of 1 or 2% per packet.

Tables 4.1 through 4.4 shows the average throughput at steady for TCP and QUIC flows from all loss based tests. The data in Tables 4.1 and 4.3 shows that with increase in packet loss on a smaller link capacities (4 and 8 Mbps) at all values for delay, the decrease in the observed TCP throughput is greater than decrease for

Bottleneck Bandwidth Mbps	One way Delay ms	Packet Loss %	QUIC Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean	TCP Throughput Bits/s	Standard Deviation Mbps	Standard Error of Mean
12	100ms	0	11,637,496	69.78	26.37	11,281,896	48,662.80	21,762.67
		0.5	11,602,376	12,970.30	4,101.57	1,734,016	9,337.29	3,529.16
		1.0	11,288,888	129,656.30	41,000.92	1,143,208	8,318.51	2,941.04
		2.0	4,181,096	445,555.11	123,574.75	815,064	3,965.19	1,401.91
16	100ms	0	15,516,928	55.64	18.55	15,301,472	79.40	39.70
		0.5	14,060,168	413,702.72	124,736.06	1,678,544	19,506.34	6,896.53
		1.0	10,492,608	715,324.04	215,678.31	1,170,136	7349.23	2,598.35
		2.0	8,320,072	779,537.08	201,275.61	833,640	1,197.50	846.76
12	200ms	0	11,602,608	3,752.62	1,418.36	11,476,296	26.90	13.45
		0.5	8,679,672	407,857.21	113,119.24	1,155,672	15,360.85	5,430.88
		1.0	5,918,960	539,938.66	155,866.87	764,496	3,142.76	1,283.03
		2.0	3,679,272	498,810.93	143,994.31	480,352	3,409.88	1,392.08
16	200ms	0	12,705,400	222,211.69	83,988.12	14,628,904	188,161.02	76,816.42
		0.5	9,642,320	513,089.28	162,253.08	1280,312	16,449.94	5,815.93
		1.0	6,145,568	683,775.33	189,645.15	746,344	6,346.94	2,115.65
		2.0	4,931,296	672,731.75	186,582.22	482,136	2,664.30	1,087.69

Table 4.4: Average value, standard deviation and standard error for throughput from QUIC and TCP at two bottleneck bandwidths - 12 and 16 Mbps and two latency values - 100 and 200ms

QUIC which, in some cases, is negligible. The variation in the observed throughput is high in the case of QUIC as can be seen from the standard deviation values associated with QUIC throughput in Tables 4.1 through 4.4.

The increase in delay on the testbed for a given bottleneck capacity and loss results in lower throughput for TCP but the same is not always the case with QUIC, as seen in Tables 4.1 through 4.4 for 1% and 2% packet loss values. We observed this result from multiple iteration of the experiment for the same parameters. Each QUIC flow responds to congestion at different times, giving higher standard deviation values from the average throughput calculated at steady state and influences the average throughput figures. QUIC's throughput in the presence of induced packet loss becomes similar to TCP throughput under similar loss rates at high bandwidth delay product values. Tables 4.3 and 4.4 show this reduction in the observed throughput for QUIC at 12 Mbps and 16 Mbps with 200ms delay and 2% packet loss rate.

The results show a marked difference in how QUIC and TCP react to random packet loss. TCP treats these losses as an indication of congestion at a network device and decreases the congestion windows and hence the sending rate. QUIC, however, differs from TCP in its reaction to random packet loss.

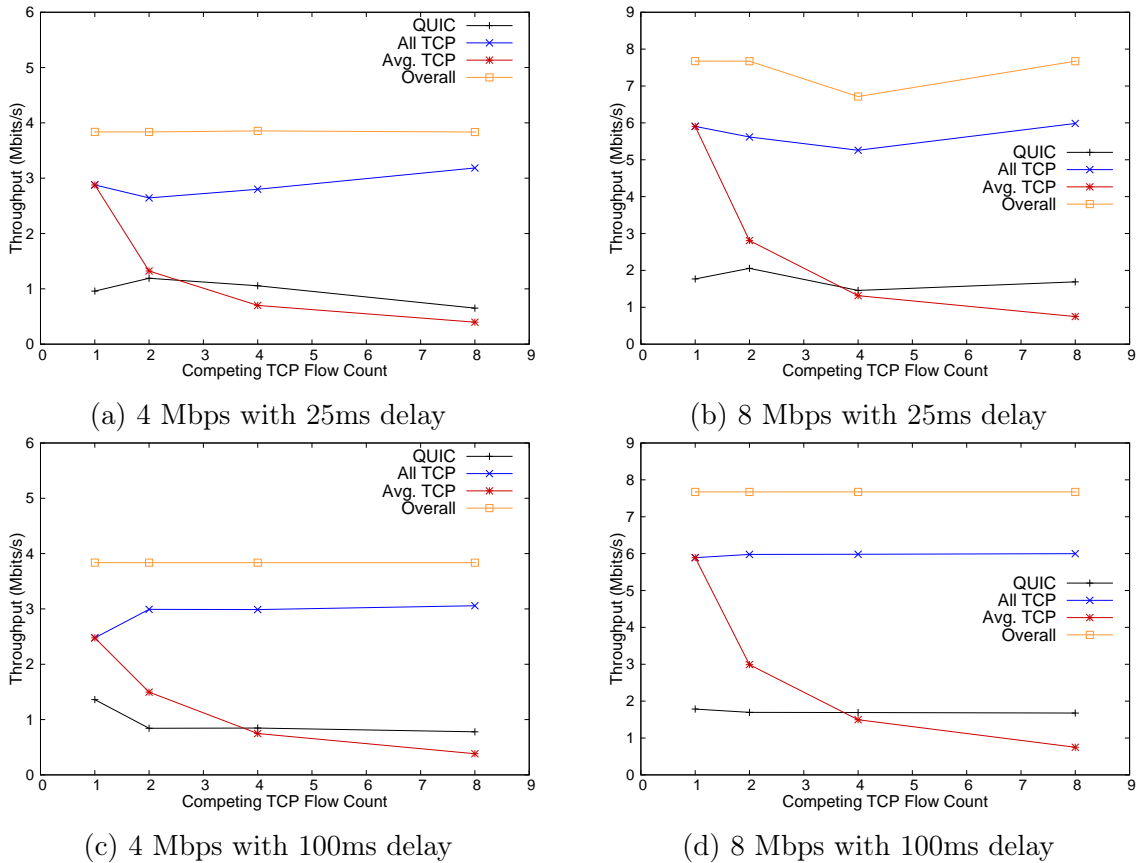


Figure 4.3: Average throughput from QUIC and competing TCP flows plotted against the number of competing TCP flows. Each vertical set of dots represents the share of QUIC throughput, the combined TCP throughput, the average TCP throughput and the overall throughput when a single QUIC flow runs simultaneously with TCP flows indicated by the on X axis.

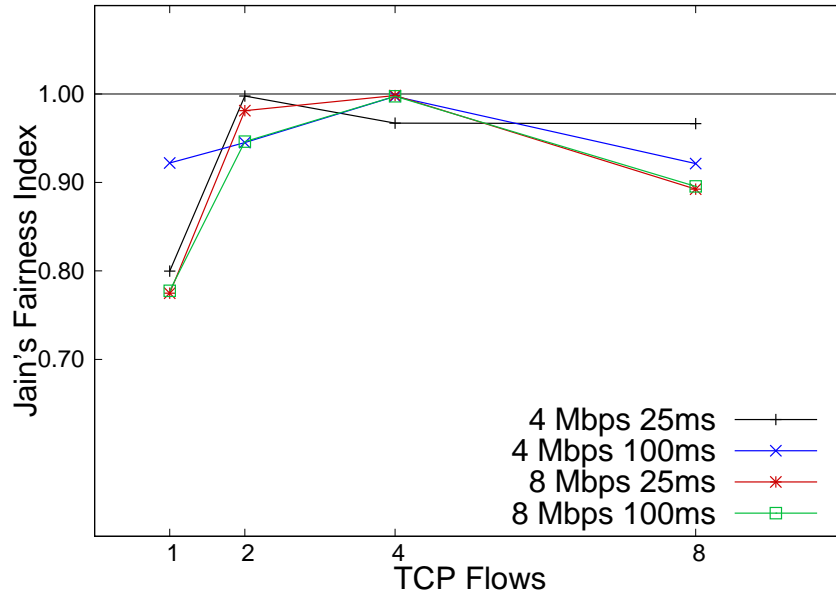


Figure 4.4: Jain's fairness index

4.4 Impact of Competing Flows

The results from experiments with competing TCP flows sending data in the same direction as a single QUIC flow can be seen in Figure 4.3 and Table 4.5. When a single TCP flow competes against a QUIC flow, the overall share of the bandwidth of QUIC is rather conservative at 25% of the link capacity. The TCP flow takes a larger share of the bandwidth. As the number of competing flows increases, QUIC's share of bandwidth becomes greater than any individual TCP flow. This indicates that against a higher flow count, QUIC can be unfair to TCP connections sharing a bottleneck with it.

Figure 4.4 contains the plot of Jain's fairness index calculated using the data from these tests. The straight horizontal black line at the top indicates the maximum fairness that can be achieved using this scale, one. As can be seen from Figure 4.3 the throughput for QUIC and TCP is most similar when one QUIC flow shares the link with four TCP links, the fairness of this experiment is highest. In case of two TCP

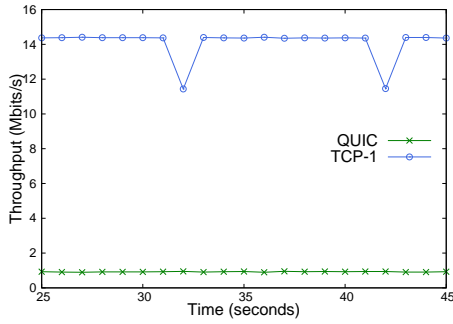
Bottleneck Capacity Mbps	Competing Flow Count	One way Delay ms	Average QUIC Throughput Bits/s	Average TCP Throughput Bits/s	Cumulative TCP Throughput Bits/s
4	1	25	119840	359863.0	359863.00
	1	100	170098	309585.0	309585.00
	2	25	148931	330523.0	165261.50
	2	100	105371	374122.0	187061.00
	4	25	131935	349984.0	87496.00
	4	100	105895	373623.0	93405.75
	8	25	81309	398121.0	49765.12
	8	100	97287	382221.0	47777.62
8	1	25	221088	738202.0	738202.00
	1	100	223036	736251.0	736251.00
	2	25	256956	702226.0	351113.00
	2	100	211875	747338.0	373669.00
	4	25	182050	657221.0	164305.25
	4	100	211385	747782.0	186945.50
	8	25	211192	748035.0	93504.38
	8	100	209482	749718.0	93714.75

Table 4.5: Average throughput from QUIC and TCP with Standard Deviation and Standard Error from experiments with two bottleneck bandwidths - 4 Mbps and 8 Mbps

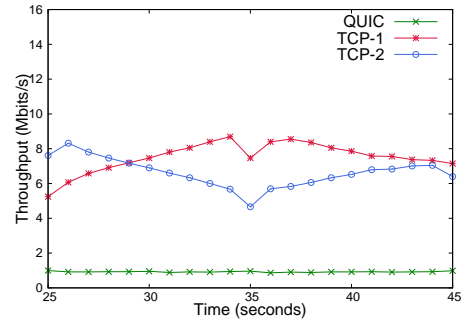
sharing the link with one QUIC at a bottleneck capacity of 4 Mbps at 25ms of added delay we saw fair sharing of the bandwidth and that result is visible in the fairness of that flow.

4.5 Internet-based Tests

Figure 4.5 shows the throughput from multiple iterations of experiments with QUIC and TCP on a 16 Mbps bottleneck, 200ms of added delay and 2% loss setting. The results from our Internet-based tests look different from the tests we conducted with emulated congestion. The QUIC connection to the Google Drive was conservative with an average throughput of roughly 1 Mbps on a bottleneck link with a maximum



(a) 4 Mbps with 25ms delay



(b) 8 Mbps with 25ms delay

Figure 4.5: Throughput from multiple iterations of experiments with QUIC and TCP on a 16 Mbps bottleneck, 200ms of added delay and 2% loss setting

bandwidth of 16 Mbps. The TCP connection from the same server dominated the available link capacity. This is observed by the straight line for QUIC throughput in Figures 4.5(a) and 4.5(b). It shows that the sending rate for the QUIC connection could possibly be capped at the server.

Chapter 5

Conclusions

QUIC is a new network protocol that resides in the application layer over UDP. Google developed QUIC as an alternative to TCP. Two browsers (Chrome and Opera) and Google servers are the only entities that support QUIC. When a user accesses Google's services such as Gmail over the aforementioned browsers, the data transfer will use UDP-based QUIC. This generates thousands of QUIC connections on a daily basis that share the links on the Internet with TCP.

Our goal is to study QUIC's performance vis-a-vis TCP under network congestion and its response to competing TCP flows. Given the scale at which QUIC is used by Google services on the Internet it is important to understand QUIC's congestion response to determine how QUIC impacts competing traffic. Our experiments are mainly conducted by controlling congestion on a wired testbed at Worcester Polytechnic Institute in Worcester, MA.

Our experiments show that QUIC and TCP achieve similar throughput values with for all values of added delay with QUIC doing slightly better than TCP by less

than 0.1%. However, with induced packet loss along with added delay, QUIC delivers higher throughput than TCP. QUIC flows also show a high standard deviation from mean values for throughput with increasing packet loss rates. TCP flows on the other hand had relatively lower standard deviations across experiments since TCP had a more consistent throughput for a given set of network parameters.

QUIC flows take a fixed share of the available bandwidth in the presence of competing TCP flows. The results show this fraction to be roughly 25% of the available bandwidth on our testbed. Irrespective of the bottleneck bandwidth, we get the same result with up to eight TCP flows. The bandwidth share of individual TCP connections goes falls below the throughput of the QUIC flow with 4 competing flows. This result demonstrates that QUIC flows can be unfair to TCP flows.

We conducted our Internet-based tests with the goal of verifying some of the results from our tests where QUIC and TCP compete for bandwidth on our wired testbed without Internet access. We used the latest available version of QUIC at both ends of the connection for the Internet-based tests. We find that QUIC takes a fixed but small share of the available bandwidth in the presence of competing TCP flows. This share of the overall link capacity does not change with increase in the number of competing TCP flows. Hence, we can postulate that if sufficiently high number of TCP flows would share a given link, they may receive a lower share of the bandwidth than a QUIC flow.

Chapter 6

Future Work

In this chapter, we briefly describe work that can be done using our current lab setup to further study QUIC. We can explore features such as Connection Migration, Multiplexing requests over QUIC Streams and QUIC's performance over wireless networks.

6.1 QUIC with Competing Flows

We found that a single QUIC connection can take unfair share of bandwidth from TCP flows sharing the same link. To further explore this behavior of QUIC in a real world setting, we need multiple flows of each protocol and a higher link capacity. We propose running our competing flows test with 10 or 20 parallel TCP connections on a 100 Mbps link sharing the network with up to 10 QUIC connections. We could also use a more recent version of QUIC, revision 36, which came out in late 2016. The goal would be to study QUIC flows react to other QUIC flows in the presence of TCP flows. The throughput at steady state for each flow would allow us to learn more about the current state of QUIC's congestion response fairness to competing flows.

6.2 Connection Migration

Connection Migration allows re-use of an existing QUIC connection when a device such as laptop or mobile phone changes its mode of network access. For example, a device is assigned a new IP address when it switches between wireless and wired networks. Our testbed can be supplemented with a pair of wireless routers to provide wireless connectivity between the emulator and the rest of the devices. We use a script to disable the Ethernet interfaces and enables wireless interfaces on the server and client devices. In this scenario, an application will detect the loss of connectivity when it fails to use an existing TCP connection and subsequently tries to re-connect to the remote server. The goal of this test is to understand how QUIC implements connection migration and how applications can benefit from it.

6.3 QUIC Streams - Request Multiplexing

QUIC is currently deployed inside a browser, but many other network-driven applications may appreciate an alternative to TCP. These include streaming and video chat applications that send audio and video components over a TCP connection via multiplexing. Such an application can use different QUIC streams for audio and video delivery. Our goal is to explore QUIC streams, and impact of stream throughput on the overall connection and the application using it. This test uses file downloads to study stream performance in the presence of delay and loss.

6.4 QUIC over a Wireless Network

Another useful test case would be to evaluate QUIC's performance over a wireless connection. This test would help study QUIC where losses occur in the wireless

(physical) layer and impact the upper layers in the form of timeouts as opposed to congestion-based loss.

Chapter 7

Appendix

This chapter contains some of the graphs from the experiments we conducted.

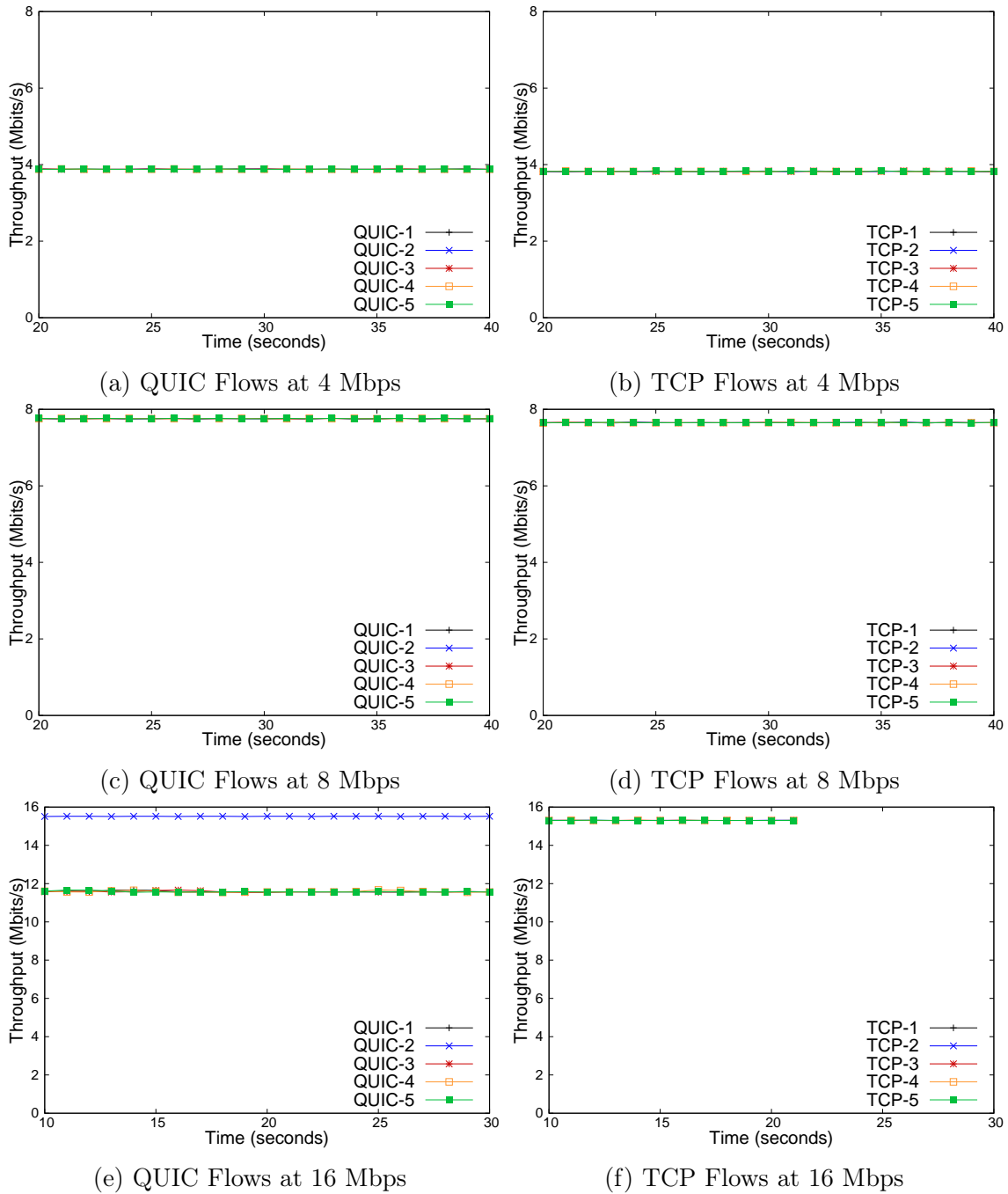
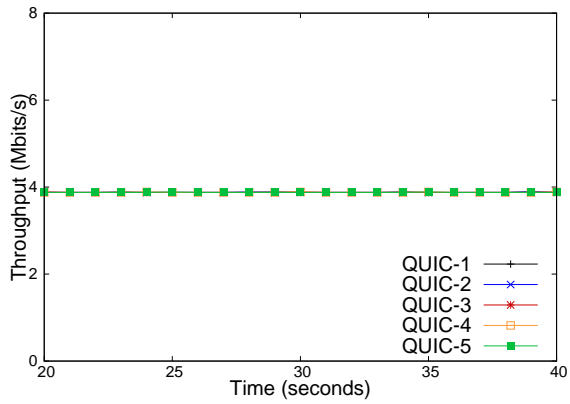
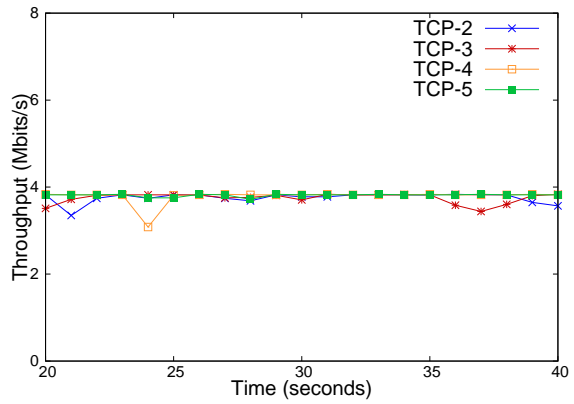


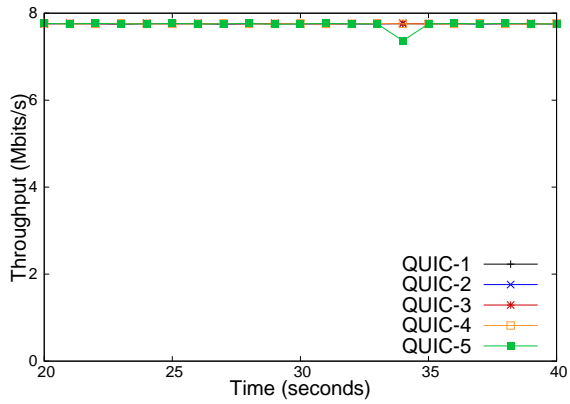
Figure 7.1: Throughput from experiments at various bottleneck capacities and 200ms added delay and 0% loss



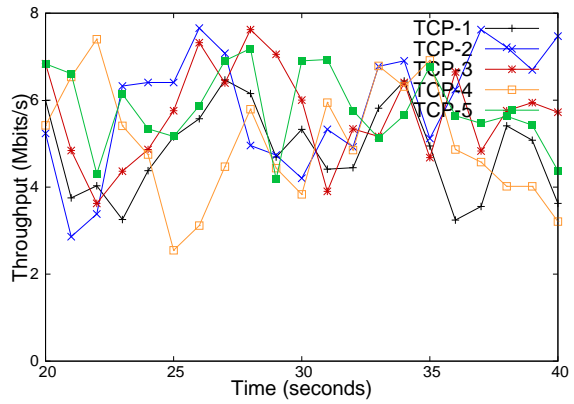
(a) QUIC Flows at 4 Mbps



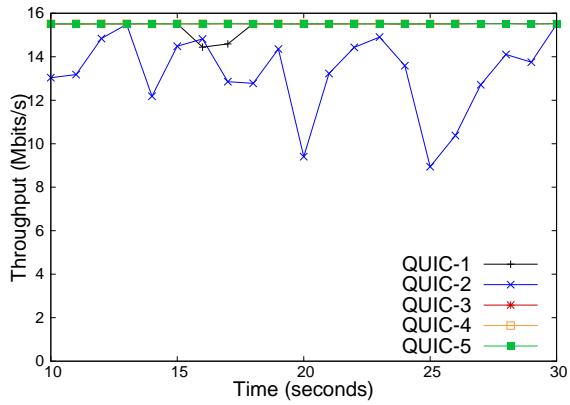
(b) TCP Flows at 4 Mbps



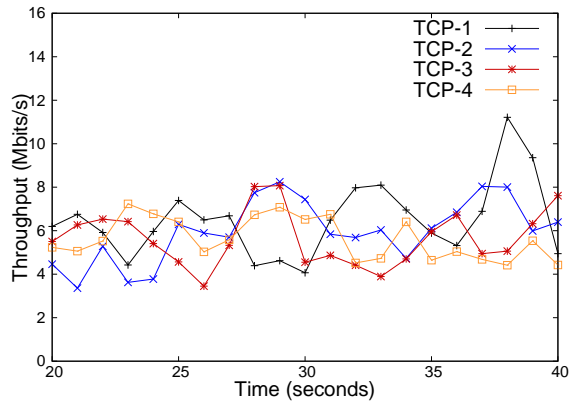
(c) QUIC Flows at 8 Mbps



(d) TCP Flows at 8 Mbps

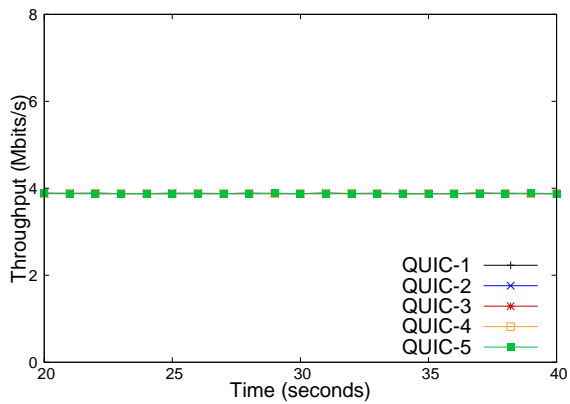


(e) QUIC Flows at 16 Mbps

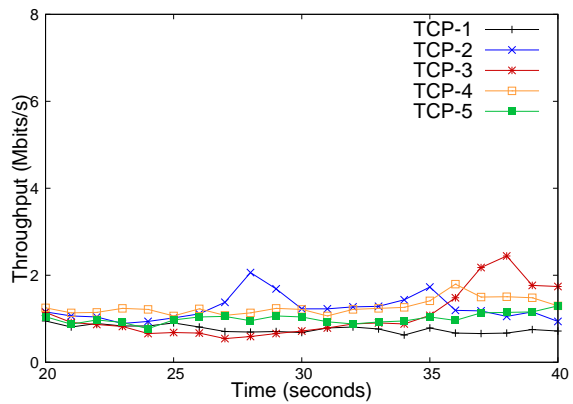


(f) TCP Flows at 16 Mbps

Figure 7.2: Throughput from experiments at various bottleneck capacities, 25ms added delay and 0.5% loss

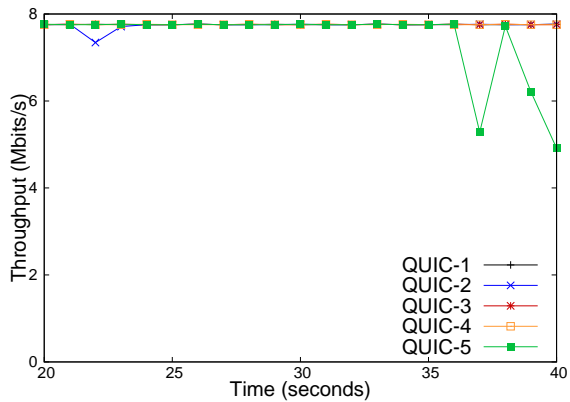


(a) QUIC Flows at 4 Mbps

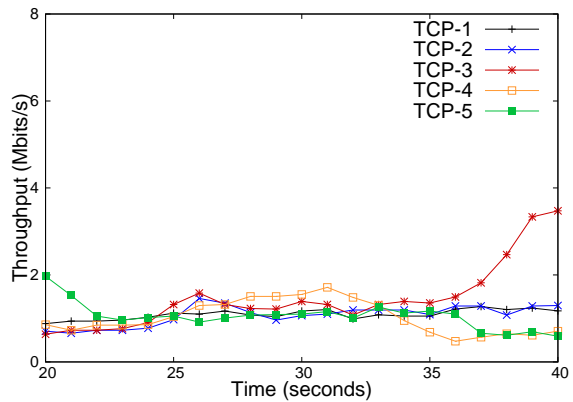


(b) TCP Flows at 4 Mbps

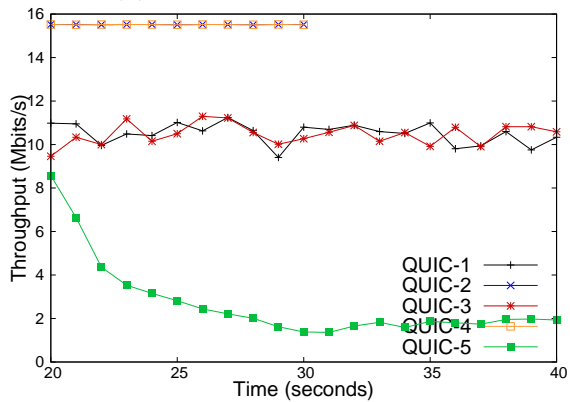
Figure 7.3: Throughput from multiple experiments with a 4Mbps bottleneck, 200ms added delay and 0.5% loss



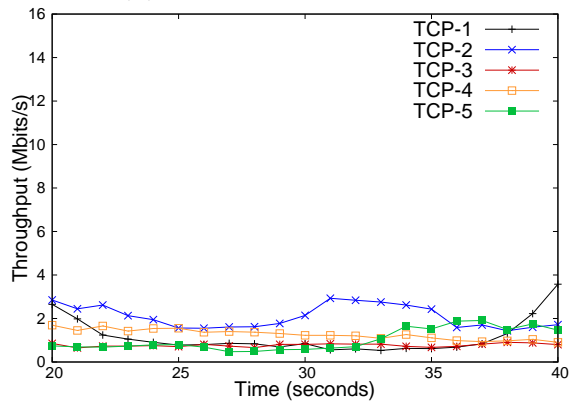
(a) QUIC Flows at 8 Mbps



(b) TCP Flows at 8 Mbps

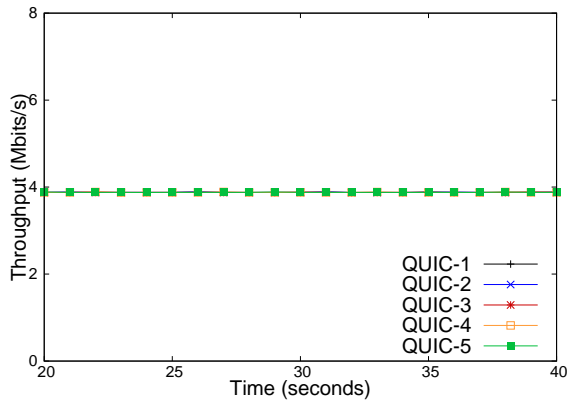


(c) QUIC Flows at 16 Mbps

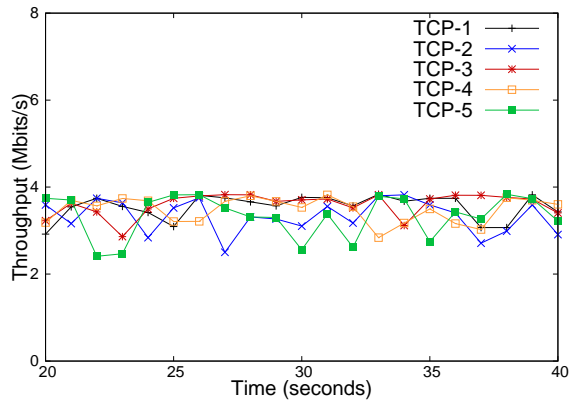


(d) TCP Flows at 16 Mbps

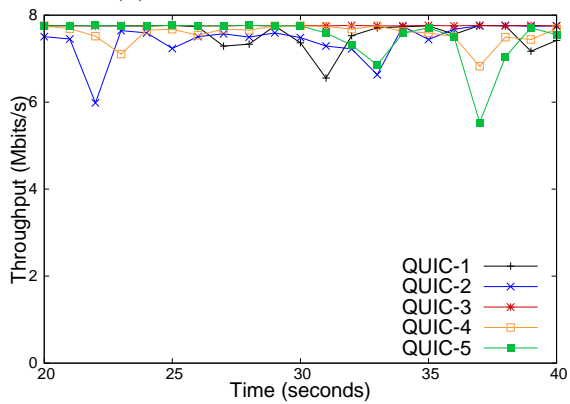
Figure 7.4: Throughput from experiments at various bottleneck capacities, 200ms added delay and 0.5% loss



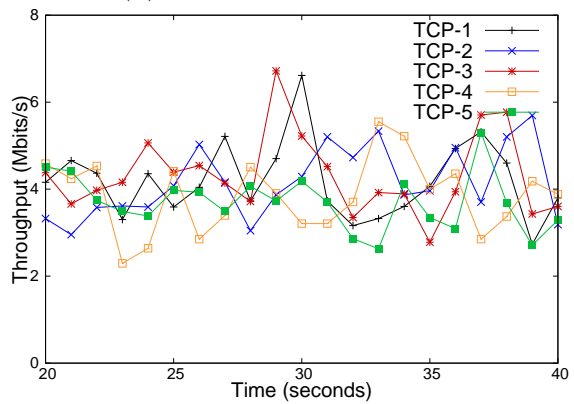
(a) QUIC Flows at 4 Mbps



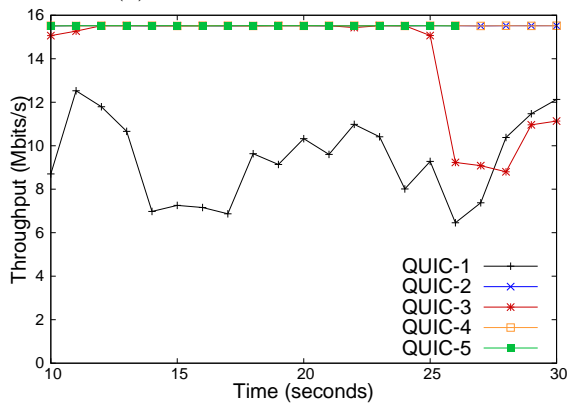
(b) TCP Flows at 4 Mbps



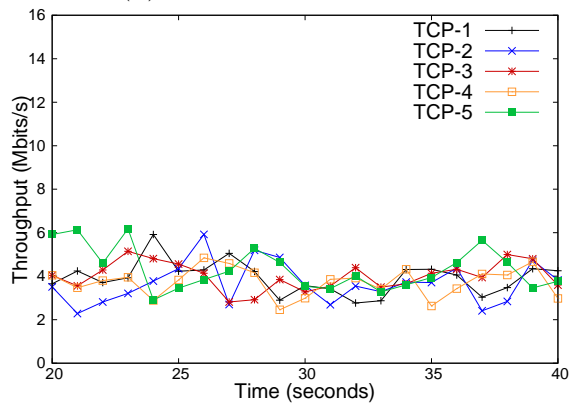
(c) QUIC Flows at 8 Mbps



(d) TCP Flows at 8 Mbps



(e) QUIC Flows at 16 Mbps



(f) TCP Flows at 16 Mbps

Figure 7.5: Throughput from experiments at various bottleneck capacities, 25ms added delay and 1.0% loss

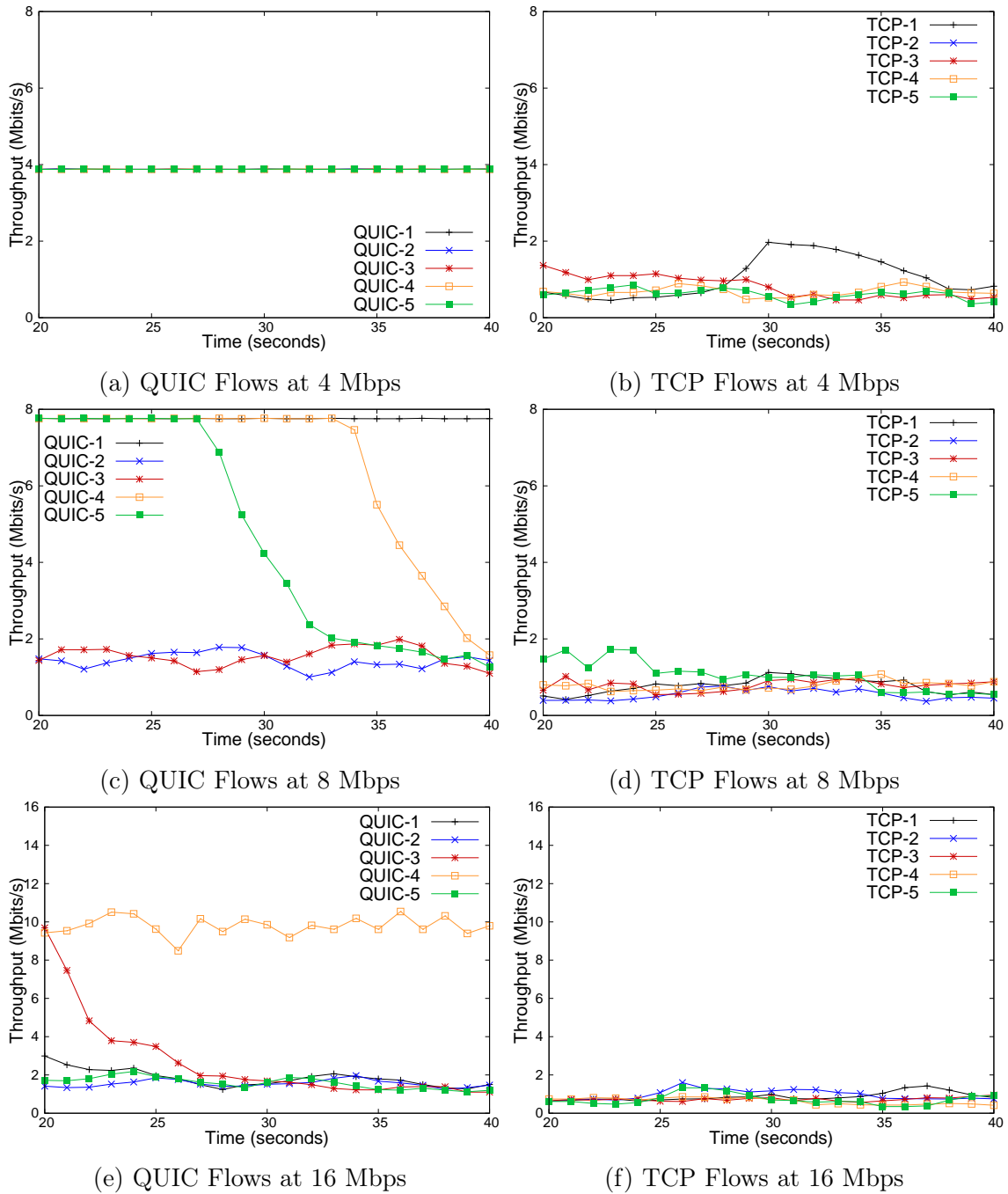


Figure 7.6: Throughput from experiments at various bottleneck capacities, 200ms added delay and 1.0% loss

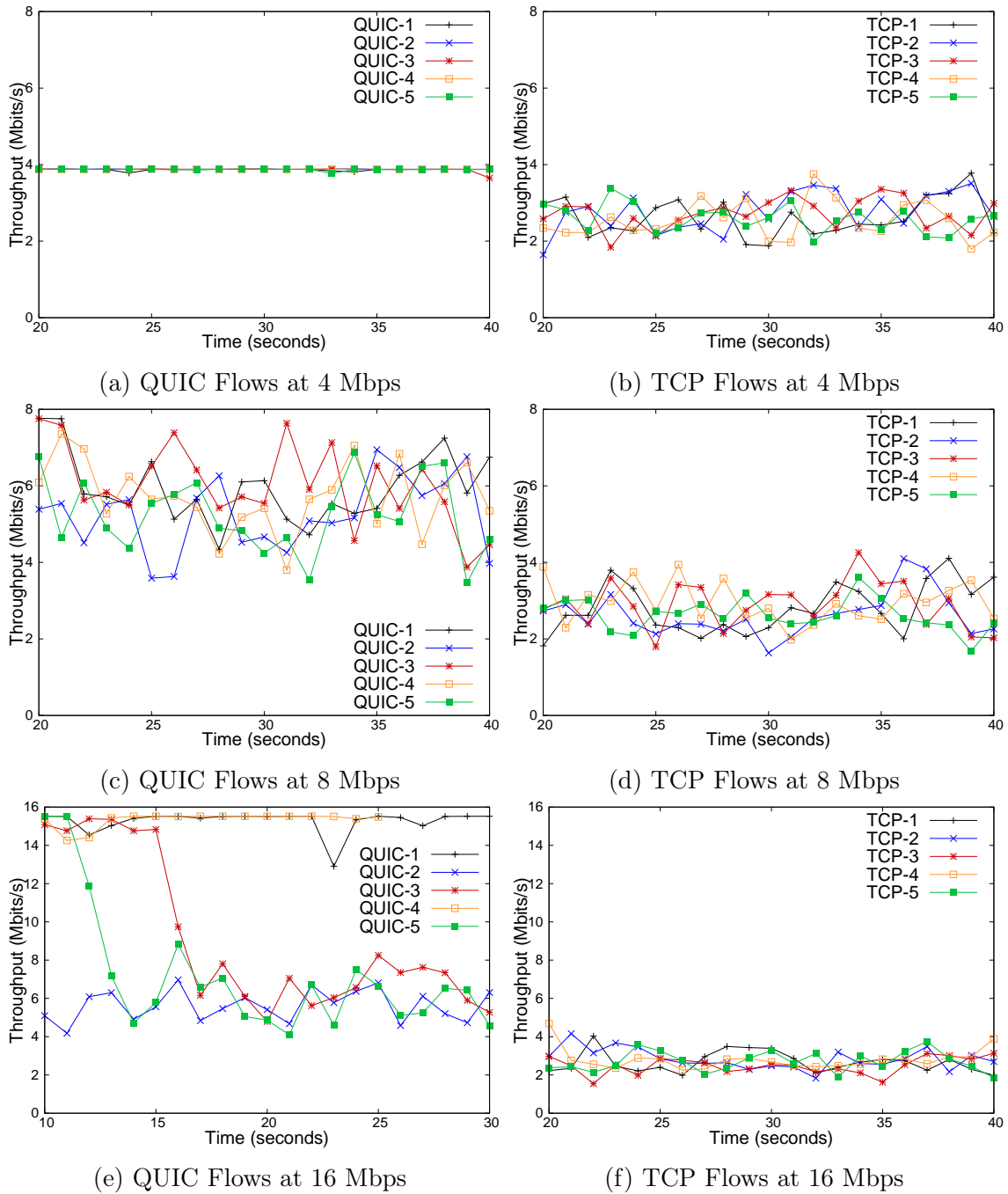
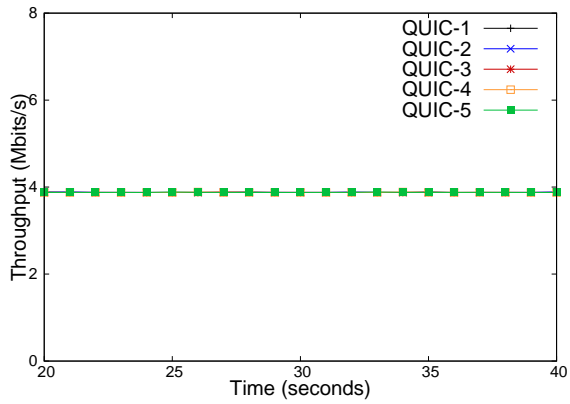
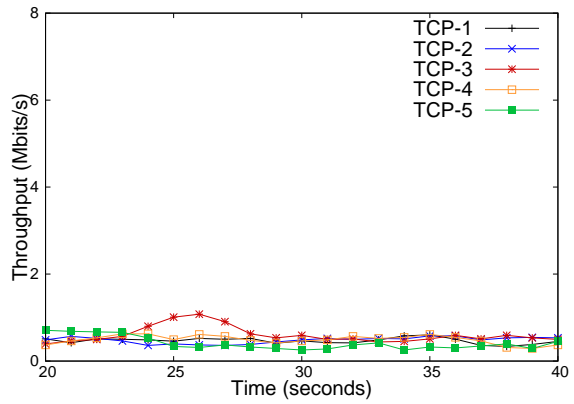


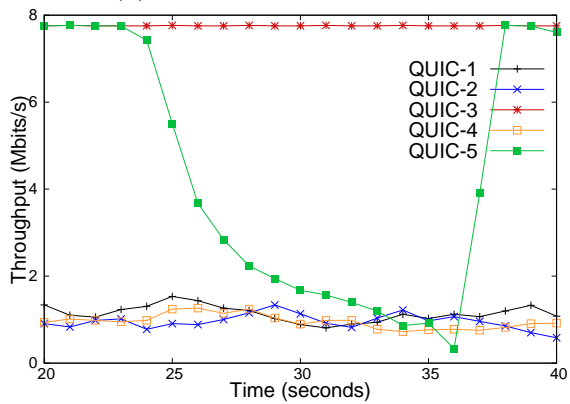
Figure 7.7: Throughput from experiments at various bottleneck capacities, 25ms added delay and 2.0% loss



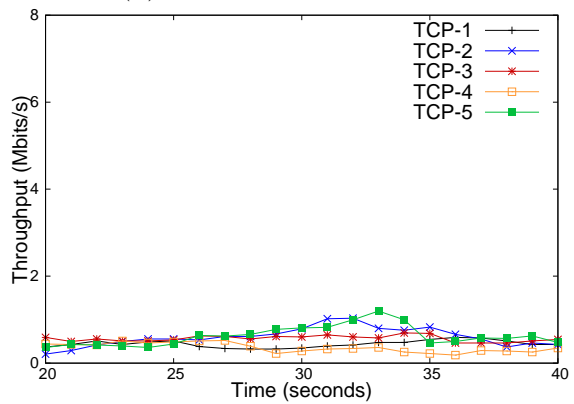
(a) QUIC Flows at 4 Mbps



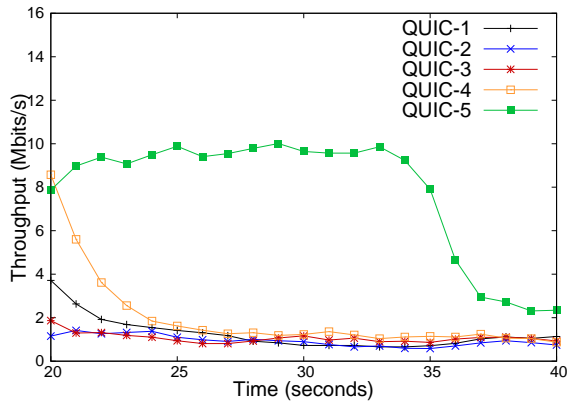
(b) TCP Flows at 4 Mbps



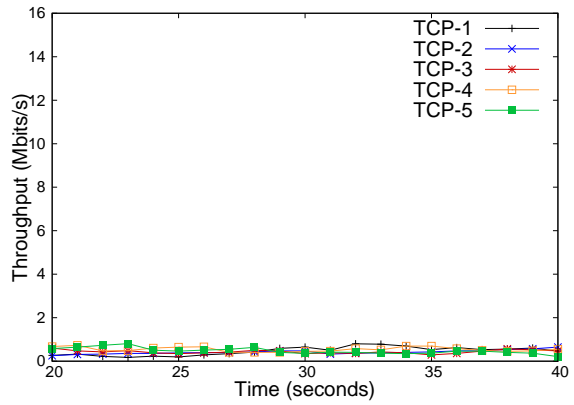
(c) QUIC Flows at 8 Mbps



(d) TCP Flows at 8 Mbps



(e) QUIC Flows at 16 Mbps



(f) TCP Flows at 16 Mbps

Figure 7.8: Throughput from experiments at various bottleneck capacities, 200ms added delay and 2.0% loss

Bibliography

- [BGG⁺08] Neda Beheshti, Yashar Ganjali, Monia Ghobadi, Nick McKeown, and Geoff Salmon. Experimental Study of Router Buffer Sizing. In *Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement*, IMC '08, pages 197–210, New York, NY, USA, 2008. ACM.
- [BM02] R. Bush and D. Meyer. Some Internet Architectural Guidelines and Philosophy. RFC 3439, RFC Editor, December 2002.
- [CDCM15] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. HTTP over UDP: an Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 609–614. ACM, 2015.
- [Flo03] Sally Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, RFC Editor, December 2003.
- [ipe16] iPerf - The TCP, UDP and SCTP network bandwidth measurement tool. <https://iperf.fr/>, 2016. [Online; accessed 01-January-2017].
- [Kel03] Tom Kelly. Scalable tcp: Improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, April 2003.
- [LJBNR15] Robert Lychev, Samuel Jero, Alexandra Boldyreva, and Cristina Nita-Rotaru. How secure and quick is QUIC? Provable security and performance analyses. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 214–231. IEEE, 2015.
- [MAB09] V. Paxson M. Allman and E. Blanton. TCP Congestion Control. RFC 5681, RFC Editor, September 2009.
- [MKM16] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. How quick is quic? In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.
- [net16] networking:netem Linux Foundation Wiki. <https://wiki.linuxfoundation.org/networking/netem/>, 2016. [Online; accessed 01-January-2017].

- [qui17a] Playing with QUIC. <https://www.chromium.org/quic/playing-with-quic/>, 2017. [Online; accessed 01-January-2017].
- [qui17b] QUIC, a multiplexed stream transport over UDP - The Chromium Projects. <https://www.chromium.org/quic/>, 2017. [Online; accessed 01-January-2017].
- [qui17c] QUIC Crypto. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45Ib1Hd_L2f5LTaDUDwvZ5L6g/, 2017. [Online; accessed 01-January-2017].
- [tcp16] Tcpcdump. <http://www.tcpcdump.org/>, 2016. [Online; accessed 01-January-2017].
- [TSZS06] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12, April 2006.
- [WJLH06] David X Wei, Cheng Jin, Steven H Low, and Sanjay Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.