Benchmarks and Analysis of QUIC Performance on Emulab

by

Naveenraj Muthuraj

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science
University of Alberta

Abstract

QUIC has been a fast-evolving protocol and, with its standardization as part of HTTP/3, it is an important part of the World Wide Web. Since its introduction in 2014, QUIC changed significantly from Google QUIC (gQUIC) to an IETF standard (2021). Understanding the performance of the current version of QUIC and how it compares to the older version is important, given its evolution and increased adoption.

We conduct a comprehensive performance evaluation of two versions of QUIC against TCP: Google QUIC version 37 from 2017 (gQUICv37) and IETF QUIC version 1 from 2021 (QUICv1). Following the parameters (e.g., latency, loss, and jitter) and methodology established by a notable QUIC paper from 2017, we validate the performance of gQUICv37 and extend our experiments to QUICv1. We leverage the Emulab testbed, which facilitates reproducible research.

We show that the performance advantages of QUIC over TCP, from core features like 0-RTT to reduce connection latency and multiple streams to avoid head-of-line (HOL) blocking, are consistent in gQUICv37 (2017) and QUICv1 (2021). There are also notable performance advantages due to the (1) new BBR congestion control algorithm and (2) updated loss detection strategy, that improve QUICv1 over gQUICv37 under packet reordering scenarios by (1) 60% to 80% and (2) 46% to 48% (using CUBIC), respectively, particularly for 10 MB objects. By utilizing Emulab and sharing our scripts and code, we enable other researchers to replicate and extend our study for future versions of QUIC.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Paul Lu, whose guidance, mentorship, and unwavering support have been invaluable throughout this journey. Without his assistance, my thesis writing process would have been far more challenging, and I am truly grateful for his patience and insights. I would also like to thank Dr. Ivan Fair and Dr. Hamidreza Anvari for serving on my committee, and Dr. Nilanjan Ray for being the exam chair.

Special thanks are due to the Flux group at the University of Utah for providing and supporting Emulab, which has facilitated the execution of my experiments. The financial support of the Department of Computing Science through a Teaching Assistantship is gratefully acknowledged.

I would like to thank my parents, Muthuraj and Kamala for deviating from the rural Indian norm of creating assets for children, and instead, making us assets through education. I am grateful to my brother, Amruthraj, for his unwavering support and encouragement throughout my academic journey.

Finally, I would like to thank my friends for their support and encouragement, which has been a source of strength and motivation throughout my time at the University of Alberta.

Table of Contents

List of Tables

List of Figures

Abbreviations

ACK Acknowledgment.

BBR Bottleneck Bandwidth and Round-trip propagation time.

BDP Bandwidth Delay Product.

CCA Congestion Control Algorithm.

CWND Congestion Window.

DNS Domain Name System.

gQUICv37 Google QUIC version 37.

HAR HTTP Archive format.

HOL head-of-line.

HTML Hyper Text Markup Language.

HTTP Hyper Text Transfer Protocol.

HTTP/3 Hyper Text Transfer Protocol version 3.

IETF Internet Engineering Task Force.

MASQUE Multiplexed Application Substrate over QUIC Encryption.

NACK Negative Acknowledgment.

PLT Page Load Time.

QUIC Quick UDP Internet Connections.

QUICv1 IETF QUIC version 1.

QUICv2 IETF QUIC version 2.

RACK-TLP Recent ACKnowledgment with Tail Loss Probe.

RTT Round-trip time.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

Chapter 1

Introduction

Over the years spanning from 2017 to 2021, the QUIC transport protocol has undergone significant changes, evolving from its inception by Google to its standardization by the IETF (Figure 1.1). With the emergence of HTTP/3, which adopts QUIC as its underlying transport layer protocol, understanding the performance of the current version of QUIC is paramount. Equally crucial is establishing a reproducible experimental framework to facilitate the evaluation of future iterations of QUIC. Since QUIC operates in user space rather than relying on the underlying operating system, it is easier to evolve and deploy. The rapid evolution of this transport protocol underscores the necessity of an evaluation and analysis framework to discern the impact of design decisions made along the way.

To address these imperatives, this thesis aims to achieve three primary goals. Firstly, it seeks to comprehensively understand the performance of the recent version of QUIC (i.e., QUICv1) within a modern and reproducible testbed called Emulab. Secondly, it aims to provide an easily reproducible experimental framework that can be used to evaluate future versions of QUIC. Lastly, by benchmarking two different versions of QUIC, we aim to investigate the impact of changes in the QUIC protocol over the years.

QUIC is a general-purpose transport layer protocol introduced by Google around 2014 [1] and later adopted by the IETF in 2016 [2]. The timeline of QUIC evolution is

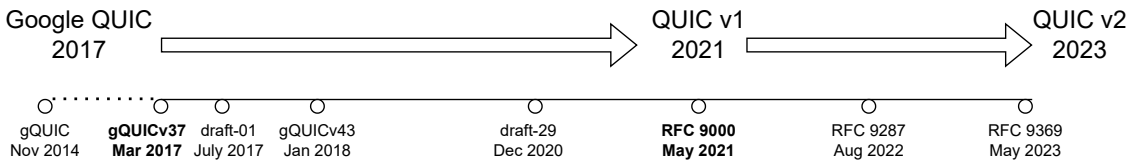


Figure 1.1: QUIC evolution timeline showing major QUIC versions and extensions. The versions in bold are the ones we studied in this thesis.

shown in Figure 1.1. QUIC has evolved significantly while transitioning from Google QUIC (gQUIC) to an IETF standard, which was released as RFC 9000 QUIC version 1 (QUICv1) in 2021 [3]. After which, it has continued to evolve with the release of RFC 9369 QUIC version 2 (QUICv2) in 2023 [4].

In this work we study the performance of QUIC in two different versions, Google QUIC version 37 (gQUICv37) and IETF QUIC version 1 (QUICv1), which are both implemented in Chromium. While numerous studies have benchmarked QUIC performance against TCP, this thesis takes a unique approach by benchmarking two different versions of QUIC (gQUICv37 and QUICv1) from the same implementation stack (Chromium), each compared against TCP. These versions are distinguished by two standards (Google QUIC and IETF QUIC) and separated by five years of protocol changes (2017 to 2021).

We conduct a comprehensive performance evaluation of QUIC in two stages, as described in Section 3.2.4 and Table 3.2. Initially, we replicate experiments conducted by Kakhki *et al.* [5] on gQUICv37. In the second stage, we reproduce the same experiments using QUICv1 and draw comparisons with the results obtained from gQUICv37. A direct comparison of gQUICv37 and QUICv1 is not the focus of this work; rather, we compare each version of QUIC to the TCP implementation of its respective period (i.e., 2017 and 2024). All our experiments are conducted on Emulab, which enables reproducible research. Section 3.2.1 provides an overview of Emulab and the experimental setup used in this thesis.

As presented in Chapter 4, our benchmarking results show that QUIC's perfor-

Chromium QUIC Changes			
Test parameter	**gQUIC v37**	**QUIC v1**	**Findings**
Latency	QUIC Crypto	TLSv1.3	Different Library
	0-RTT	0-RTT	Same functionality
Loss	HOL	HOL	Same, N = 2 has impact
Bandwidth	Depends on CCA	Depends on CCA	Same, N = 2 has impact on fairness
Jitter	NACK = 3	NACK = Dynamic	Dynamic NACK improves QUIC performance
CCA	Cubic	Cubic,	Cubic with same parameters are fair
		BBR, BBRv2	QUIC is effective with BBR under jitter

Table 1.1: Overview of QUIC changes and their impact on performance

mance advantages over TCP, given by fundamental features like 0-RTT (Section 4.3) and head-of-line (HOL) blocking (Section 4.4), have remained consistent across the two versions. However, the introduction of new Congestion Control Algorithms (CCA) like Bottleneck Bandwidth and Round-trip propagation time (BBR) (Section 4.7) and loss detection strategies under packet reordering (Section 4.6) have notably improved QUICv1 in comparison to its predecessor, gQUICv37. Additionally, we show that the CUBIC parameter (Emulated connection, N) in Chromium QUIC implementation impacts its fairness with TCP (Section 4.1.1).

Our work makes the following contributions:

1. **Benchmarking**: We present a parameter sweep benchmark of QUIC vs. TCP, using two versions of QUIC (gQUICv37, QUICv1), across a variety of web pages (object sizes, and object counts) under different network bandwidths, latencies and loss rates.

2. **Reproducibility**: We document the process for supporting our benchmarking methodology on Emulab, which makes it easier for other researchers to directly reproduce our results and extend them for future versions of QUIC.

3. **Root cause analysis**: We perform a root cause analysis of the performance differences between QUIC and TCP, and the key changes in QUICv1 that led to

the performance improvements. Our three analyses include evaluating QUIC's 0-RTT advantage, explaining CUBIC unfairness in QUIC, and the impact of the updated loss detection strategy in QUICv1 under jitter.

In summary, this thesis conducts the performance evaluation of the recent version of QUIC (i.e., QUICv1) using the Emulab testbed. Specifically, we benchmark QUICv1 and gQUICv37 in an extensive parameter sweep. By using two different versions of QUIC, we investigate the impact of changes in QUIC protocol over the years. We share a reproducible Emulab experiment profile to enable easier replication of our work and extension to future QUIC versions. For every experiment, we try to explain the root cause of the performance difference between QUIC and TCP, and the key changes in QUICv1 that led to the performance improvements over gQUICv37.

Chapter 2

Background and Related Work

In this chapter, we discuss the design of QUIC and its features that can make it a better transport protocol than TCP. We relate the features of QUIC to the problems of TCP and how QUIC addresses them. For example, while TCP uses approximately 3 RTTs (TCP and TLS handshake, Figure 2.2a) to establish a secure connection, QUIC uses 0-RTT connection establishment to reduce connection latency (Section 2.1.1). Where applicable, we also note the differences between gQUICv37 and QUICv1 in relation to the features being discussed. For instance, while gQUICv37 uses Google Crypto for 0-RTT connection establishment, QUICv1 uses TLSv1.3 for the same (Figure 2.3). Subsequently, we survey various related works that have studied the performance of QUIC (Section 2.2).

2.1 QUIC

QUIC, originally an acronym for "Quick UDP Internet Connections" was initially designed by Jim Roskind at Google in 2012. It was made public and described at an IETF meeting in 2013 [6]. QUIC aimed to address three main problems of SPDY (predecessor of HTTP/2) : Reduce connection latency and improve security, avoid TCP head-of-line (HOL) blocking, and enable faster evolution and deployment than TCP.

As seen in Table 2.1, the newly designed protocol addresses the issues with TCP

TCP Problems	QUIC Solutions
Connection Delay	1-RTT / 0-RTT Connection
HOL blocking	Independent Streams
Protocol Entrenchment	End-to-End Encrypted
Implementation Entrenchment	User-space implementation

Table 2.1: Addressing TCP issues with corresponding QUIC features [7].

by including new features. These feature ideas were a culmination of lessons learned from previous protocol designs [8]. They are as follows:

1. QUIC combines transport and security handshakes to enable 1-RTT/0-RTT connection establishment, therefore reducing connection latency.

2. QUIC supports multiple independent streams over a single connection, to avoid HOL blocking.

3. QUIC uses the User Datagram Protocol (UDP) as the underlying protocol and implements TCP functions like congestion control, flow control, and reliability in the user space, enabling faster evolution and deployment.

Although we do not discuss all the features of QUIC, we will discuss three of QUIC's features, 1-RTT/0-RTT connection, HOL blocking, and congestion control, in Section 2.1.1, Section 2.1.2 and Section 2.1.3, respectively.

While Google adopted and reported the benefits of QUIC to improve transport performance for HTTPS [7], the QUIC working group at IETF was formed in 2016 to standardize the protocol. The working group published QUIC version 1 (RFC 9000) in 2021 [3], making QUIC a secure, general-purpose transport protocol. Along with RFC 9000, additional specifications were also introduced. RFC 9001 [9] details the integration of TLS to secure QUIC, and RFC 9002 [10] outlines the loss detection and congestion control mechanisms for QUIC.

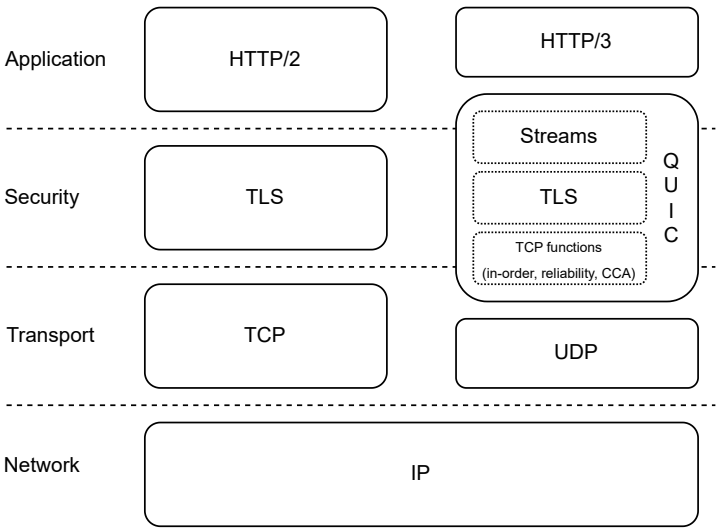


Figure 2.1: QUIC in relation to TCP and TLS [7].

IETF also defined HTTP/3 [11], the mapping of HTTP to QUIC as a transport layer protocol.

> "HTTP/3 relies on QUIC to provide confidentiality and integrity protection of data; peer authentication; and reliable, in-order, per-stream delivery. While delegating stream lifetime and flow-control issues to QUIC, a binary framing similar to the HTTP/2 framing is used on each stream. Some HTTP/2 features are subsumed by QUIC, while other features are implemented atop QUIC" RFC 9114 [11, Section 1.2]

As seen in Figure 2.1, QUIC is a transport layer protocol that is built on top of UDP and uses TLS for encryption. QUIC adds TCP-like features such as congestion control, flow control, and reliability in the user space. Similar to HTTP/2, QUIC provides multiple streams over a single connection, which the application can use to send data.

Due to the performance benefits of QUIC over TCP, QUIC saw an increase in adoption by the large providers [12], even before standardization. The trend of increasing QUIC adoption has continued with the usage of HTTP/3, which has generally increased for web traffic [13]. As of today, QUIC is supported by major web browsers

like Chrome, Firefox, Safari, and Edge [14].

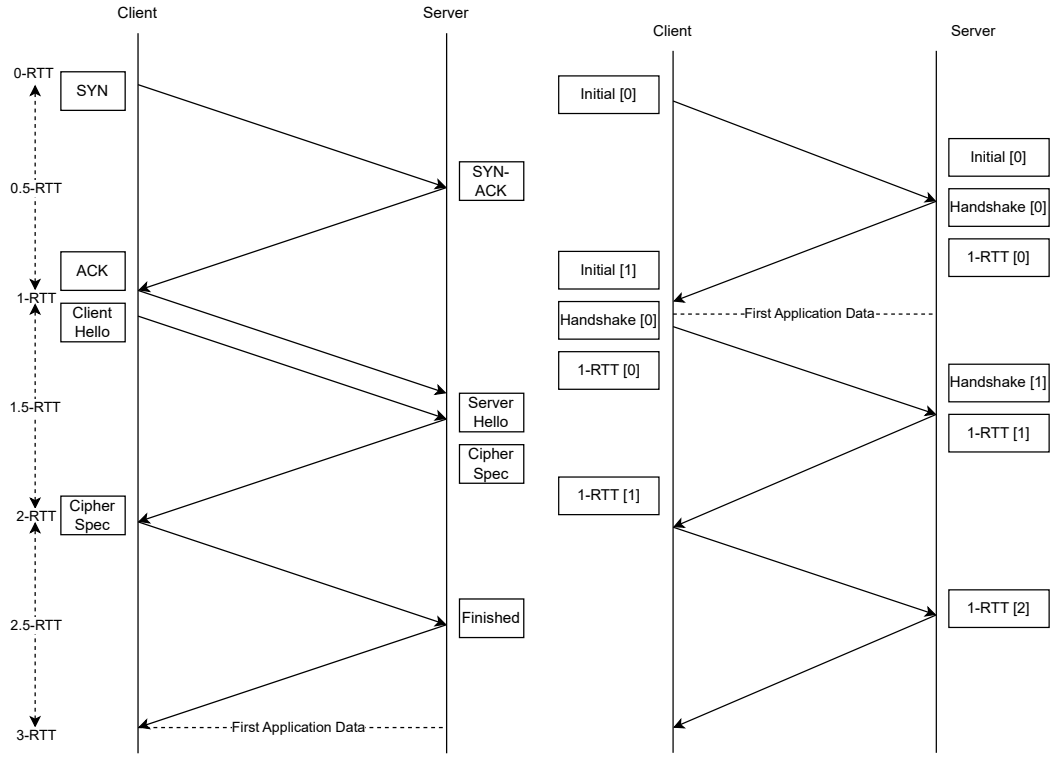
2.1.1 1-RTT / 0-RTT Connection

While most web traffic on the Internet evolved to require secure connections, the unencrypted TCP connection required TLS to make it secure (e.g., HTTPS). This increased connection establishment latency, as the client and server had to perform both transport (TCP) and security (TLS) handshakes. QUIC aims to improve upon TCP by combining transport and security handshake, therefore reducing the connection establishment latency.

This section explores QUIC's 1-RTT (Figure 2.2b) and 0-RTT (Figure 2.3d) connection establishment, and how it compares to TCP's 3-RTT connection (Figure 2.2a). Initial QUIC connection will incur 1-RTT (Figure 2.3b) for secure connection establishment, while subsequent connections will incur 0-RTT (Figure 2.3d) for data transmission. We now look at initial connection establishment in TCP and QUIC, as shown in Figure 2.2, where 0-RTT, 1-RTT, 2-RTT, and 3-RTT along the vertical axis refer to the timeline.

From Figure 2.2a, TCP uses a total of 3x RTT (from 0-RTT to 3-RTT in the timeline) before it can send the data. Initially between 0-RTT and 1-RTT in Figure 2.2a, the client sends a SYN packet (SYN) to the server, which responds with a SYN-ACK packet (SYN-ACK), acknowledging the connection request and sending its synchronization information. Next, between 1-RTT and 2-RTT, the client receives the SYN-ACK packet and sends an ACK packet (ACK) to acknowledge the server's information. This completes the TCP three-way handshake (3WHS), which incurred 1.5x RTT. The 3-way TCP handshake connection corresponds to unencrypted HTTP traffic. To encrypt the connection, the client and server perform a TLS handshake, which corresponds to HTTPS (secure) traffic.

At the same time after 1-RTT, the client and server perform a TLS handshake to secure the TCP connection, which incurs another 1.5x RTT (from 1.5-RTT to 3-RTT



(a) 3-RTT TCP + TLS Connection (b) 1-RTT QUIC Connection

Figure 2.2: Connection establishment in TCP and QUIC. TCP incurs 3-RTT for secure connection establishment, while QUIC incurs just 1-RTT.

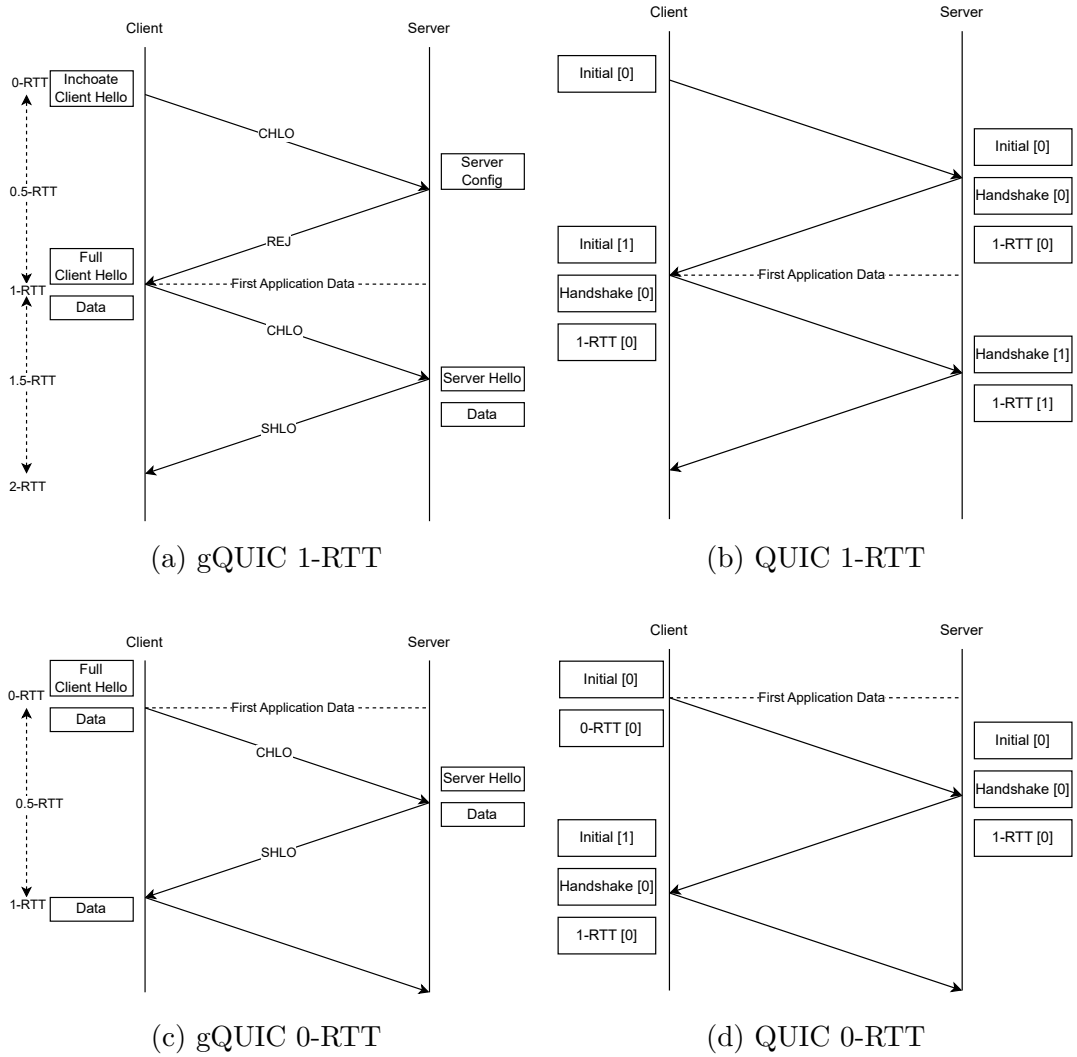
in the timeline). The TLS handshake starts with the client sending a "Client Hello" message to the server (after 1-RTT, at the start of the 2-RTT timeline), which includes the TLS version and the cipher suites supported by the client. The server responds with a "Server Hello" message, which includes the server's SSL certificate and cipher suite. At the end 2-RTT in the timeline of Figure 2.2a, the client verifies the server's SSL certificate and obtains the server's public key. For the next 1-RTT, the client then sends a pre-master secret to the server, which is encrypted using the server's public key and can only be decrypted by the server's private key. The server decrypts the pre-master secret and both the client and server generate the session key from the pre-master secret. The server sends a "Finished" message to the client, which is encrypted using the session key, and the application data can be sent encrypted using the session key.

In total, TCP incurs 3x RTT for secure connection establishment, which includes 1.5x RTT for 3-way TCP handshake (3WHS) [15] and another 1.5x RTT for its TLS handshake [16] (Total 3x RTT). Since TCP relies on TLS for encryption, and these protocols operate at different network layers, an encrypted TCP connection always incurs both transport and encryption handshake costs before data transmission begins.

However, QUIC which is encrypted by default, uses UDP for transport and integrates with TLSv1.3 for encryption. As shown in Figure 2.2b, the first QUIC packet from the client (Initial[0]) includes both the client's TLS "Client Hello" message and the QUIC transport parameters. As a result, the initial QUIC handshake combines the typical TCP 3WHS and TLS handshake [3]. After the server receives the Initial[0] packet, it responds with a packet that includes the server's TLS "Server Hello" message (Initial[0]). The server also sends a Handshake[0] packet that includes its transport parameters. At this point (0.5-RTT in the timeline), the server, with its private key and client's public key (from "Client Hello") can derive the session key to send application data with 1-RTT packets (1-RTT[0]).

After 1-RTT in the timeline of Figure 2.2b, the client receives the server's Initial[0] and Handshake[0] packet and can derive the session key to send application data with 1-RTT packets (1-RTT[0]). Hence, QUIC will complete connection establishment which includes authentication (3WHS) and encryption (TLS) using just 1x RTT.

Furthermore, QUIC due to its 0-RTT feature can send data over the network without spending extra RTTs for handshake in subsequent connections. QUIC clients can reuse the negotiated parameters from the previous connection to the same server, to send data in the first packet of the connection. As shown in Figure 2.3d, QUIC uses 0-RTT packets to send data in the first packet of the connection. Figure 2.3a and 2.3b show the initial 1-RTT connection establishment for gQUIC and QUIC, respectively. Whereas, Figure 2.3c and 2.3d show the subsequent 0-RTT connection establishment for gQUIC and QUIC, respectively. While both versions of QUIC achieve the same



(a) gQUIC 1-RTT

(b) QUIC 1-RTT

(c) gQUIC 0-RTT

(d) QUIC 0-RTT

Figure 2.3: Connection establishment in gQUIC and QUIC for initial 1-RTT and subsequent 0-RTT connection. gQUIC uses QUIC Crypto for securing QUIC connection, while QUIC uses TLSv1.3 for the same.

functionality, they use different cryptographic protocols to achieve the same.

While Google developed QUIC, it needed a cryptographic protocol to secure QUIC connections. As a result, QUIC Crypto [17] was developed, which helped to combine the cryptographic and transport handshakes to minimize connection RTTs. Figure 2.3a and 2.3c show the initial and subsequent connection establishment for gQUIC, which uses QUIC Crypto for securing QUIC connection.

In Figure 2.3a, initially, the client "Client Hello" message with the tag CHLO, are in their inchoate form. In response to an incomplete CHLO message, the server

will send a rejection message with the tag REJ. The rejection message will include information that the client can use to complete the CHLO message, in subsequent handshake attempts. Upon receiving a complete CHLO message, the server will send a "Server Hello" message with the tag SHLO, which completes the handshake.

In Figure 2.3c, the 0-RTT connection starts with a Full Client Hello message, as the client contains server config from the previous handshake. Since the client has the server config, it can send the 0-RTT data in the first packet of the connection. As the QUIC Crypto [17, Section: Client handshake] specifies, "Conceptually, all handshakes in QUIC are 0-RTT, it's just that some of them fail and need to be retried."

However, when QUIC became an IETF specification, QUICv1's cryptographic handshake was based on TLSv1.3 [16]. Figure 2.3b and 2.3d show the initial and subsequent connection establishment for QUICv1, which uses TLSv1.3 for securing QUIC connection. Figure 2.3b is the same as Figure 2.2b, as it shows the initial 1-RTT connection establishment for QUICv1, and was discussed previously in this section. Figure 2.3d shows the subsequent 0-RTT connection establishment for QUICv1, where the client will use the previous session ticket along with other connection parameters to send 0-RTT data in the first packet of the connection.

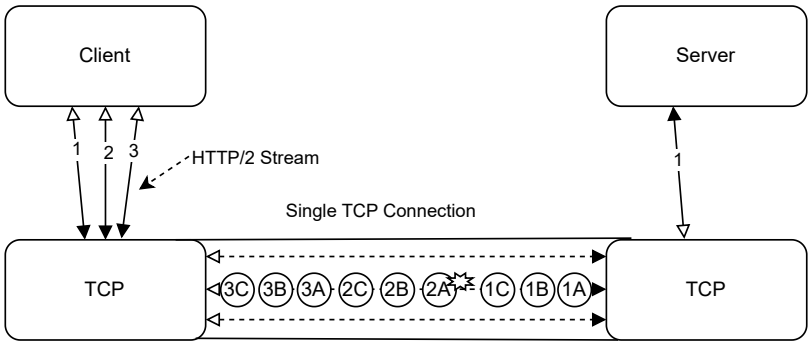
In Figure 2.3d, the client initiates the handshake process by sending an Initial packet (Initial[0]), similar to that in the 1-RTT handshake process, with a few additional TLS extensions. Specifically, the client utilizes the "pre_shared_key" extension to communicate the PSK identity it possesses and employs the "early_data" extension to indicate that it has 0-RTT data to transmit. The application data is contained within a 0-RTT packet (0-RTT[0]), which is protected with the resumption secret. This packet, along with the Initial packet, can be encapsulated into a single UDP datagram.

Upon verifying the 0-RTT packet using the TLS stack, the server will send an Initial and Handshake packet similar to the 1-RTT handshake process (Figure 2.3b, with few changes). In the "Server Hello" message, the server will include the "pre_shared_key"

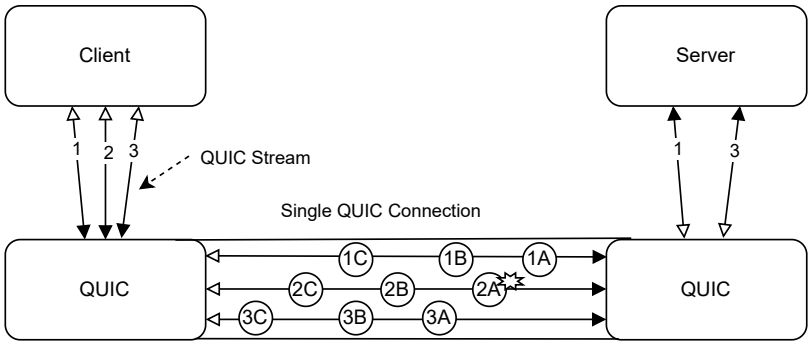
extension to indicate its acceptance of the PSK identity. In the Handshake packet, the TLS extension "early_data" will be appended, signaling the acceptance of the 0-RTT data.

In summary, QUIC's 0-RTT connection establishment allows it to send data in the first packet of the connection, without incurring extra RTTs for handshake in subsequent connections. QUIC's combined cryptographic and transport handshake for setting up a secure transport connection gives it an advantage over TCP when there is a noticeable latency in the network, as we will see in Section 4.3 and 4.5.

2.1.2 Head-of-Line Blocking



(a) TCP HOL blocking problem



(b) QUIC solving HOL blocking problem

Figure 2.4: QUIC processes each stream independently, avoiding blocking when there is packet loss in a single stream. Whereas, TCP waits for the retransmission of a lost packet, blocking the transmission of other streams.

The combining of the security layer and transport layer in QUIC resulted in a 1-RTT/0-RTT connection, a significant improvement over TCP. In addition, QUIC

also adopted features from the application layer such as multiplexed streams from HTTP/2, which allows it to avoid HOL blocking.

HTTP/2 introduced the concept of multiplexed streams (Figure 2.4a) [18, Section 5], which allows multiple independent exchanges of data over a single connection. The multiplexed stream allowed HTTP/2 to avoid HOL blocking, which was a problem in HTTP/1.1 [19]. However, HTTP/2 still uses "Single TCP Connection" as the transport layer, which is unaware of the streams in the application layer. Hence, TCP processes the data in the order it receives, which causes TCP to wait for the retransmission of lost packets, causing all other packets in different streams to be blocked.

For example, Figure 2.4a depicts a scenario where TCP's HOL blocking disrupts HTTP/2's multiplexed streams. While HTTP/2 utilizes three streams (1,2 and 3) to send data over a "Single TCP Connection", TCP itself is unaware of the streams within the data. When a packet 2A from stream 2 is lost, TCP's in-order delivery mechanism stalls all subsequent packets (Packet 2B onwards, including packets from other streams. e.g., Packet 3A) until the missing packet (Packet 2A) is retransmitted and received. As a result, all streams experience a delay because TCP waits to resolve the missing packet in the sequence, hindering the potential for parallel data transmission offered by HTTP/2's multiplexed streams.

With QUIC adopting multiplexed streams called "QUIC Streams" (Figure 2.4b) [3, Section 2], it can avoid TCP HOL blocking. As a result, when a packet is lost in a QUIC connection, only the stream that the packet belongs to will be affected, and other streams will continue to operate normally. In other words, only the stream where the data is lost will need to wait for the retransmission of lost packets, and it will not block other streams from sending data.

Figure 2.4b, shows how QUIC avoids HOL blocking issues. When the application protocol (e.g., HTTP/3) sends data divided into three streams (1, 2, and 3), QUIC processes them independently, differentiated by stream ID [3, Section 2.1]. As a

result, when packet 2A from stream 2 is lost, QUIC will not stall the transmission of packets from the other two streams i.e., packets 1B, 1C, 3A, 3B, and 3C from stream 1 and 3 can continue to flow. QUIC maintains in-order delivery within each stream, ensuring data integrity for each independent flow. QUIC's ability to process streams independently allows other streams to progress concurrently uninterrupted even in case of packet loss.

In general, QUIC's ability to process each stream independently allows it to avoid HOL blocking, which results in better performance compared to TCP when there is packet loss in the network. The performance benefits of QUIC over TCP due to HOL blocking will be further explored in Section 4.4.

2.1.3 Loss Detection and Congestion Control

As seen previously in Figure 2.1, while QUIC borrowed streams from HTTP/2 and secure connection from TLS, it uses UDP as the underlying transport protocol. QUIC uses existing UDP to be able to deploy on the current Internet. However, UDP [20] does not guarantee packet delivery, ordering, or duplicate protection. Hence, QUIC has to implement its own TCP-like features of in-order delivery, reliability, and congestion control mechanisms.

As per RFC 9002 [10], QUIC implements loss detection mechanisms similar to TCP. However, due to protocol differences between QUIC and TCP, the loss detection and congestion control mechanisms vary slightly. For example, TCP conflates transmission order at the sender with delivery order at the receiver, resulting in the retransmission ambiguity problem [21]. QUIC separates transmission order from delivery order: packet numbers indicate transmission order and delivery order is determined by the stream offsets in STREAM frames. QUIC uses a monotonically increasing packet number to identify each packet, which is used to detect packet loss and reordering.

Furthermore, due to the lack of a reference implementation and the user-space

nature of QUIC, the CCAs in QUIC have deviated from the TCP implementation in the kernel [22]. As a result, QUIC performance can vary depending on the loss-detection mechanism employed, which is further explored in Section 4.6.

2.1.4 Other features

Apart from the features discussed above, QUIC has other features that make it a better transport protocol than TCP. For example, QUIC uses a 64-bit Connection ID to uniquely identify a connection, unlike TCP which uses a 5-tuple (source IP, source port, destination IP, destination port, and transport protocol). This allows QUIC to migrate connections between different IP addresses and ports, surviving changes to the underlying network (e.g., NAT timeout and rebinding).

Further, QUIC as a transport protocol has seen applications beyond HTTP/3. For example, DNS over QUIC (DoQ) [23] provides privacy and latency benefits over DNS over UDP and DNS over TLS (DoT). Similarly, QUIC has been used for proxying via MASQUE (Multiplexed Application Substrate over QUIC Encryption) [24][25], which allows for secure and efficient proxying of application data over QUIC.

2.2 Related Work

QUIC has garnered significant attention from the research community, leading to numerous studies benchmarking its performance against TCP since 2015. The literature on QUIC performance is extensive, covering various aspects of its implementation [26], adoption [7], and effects on network performance [5]. In this section, we will discuss a few notable studies in this domain.

Google introduced QUIC, a new protocol designed from the ground up to improve the performance of HTTPS traffic. The paper by Langley *et al.* [7] provides motivation, design, and deployment experience of QUIC at Google. It presents the motivations behind the development of QUIC, highlighting the limitations of existing transport protocols as key drivers. These limitations include protocol entrenchment

(due to the middle-box), and implementation entrenchment (due to the coupling with OS). These entrenchments make it slow and challenging to evolve TCP. Moreover, the initial handshake delay in TCP due to TLS and HOL blocking issues, arising from the layered structure of protocols, further necessitated the need for a new transport protocol.

Introducing QUIC, Langley *et al.* [7] details its features, including connection establishment, 1-RTT and 0-RTT handshakes, version negotiation, stream multiplexing, encryption, and other features. They describe how each of these features addresses issues in TCP, including deployability, security, and reduction in handshake and HOL blocking delays. For example, QUIC protocol combines its cryptographic and transport handshakes to minimize connection RTTs. This was achieved by integrating the cryptographic layer into the transport layer (Figure 2.1), reducing the number of round trips required to establish a secure connection (handshake latency), something that was not possible in TCP. Similarly, QUIC was designed to support multiple streams of an application layer, e.g., HTTP/2. Making QUIC aware of multiple streams over a single connection, allowed it to avoid HOL blocking.

Langley *et al.* [7] then describe a sophisticated experimentation framework built within Chromium, enabling comprehensive testing and refinement of QUIC features. This framework facilitated the gradual deployment of QUIC across Google services. The performance evaluation of QUIC reveals significant improvements over TCP, particularly in handshake latency. They see that with increasing RTT, average handshake latency for TCP/TLS trends upwards linearly, while QUIC stays almost flat (c.f., Figure 7 [7]).

The rigorous testing and iterative improvements resulted in the wide adoption of QUIC across Google, resulting in 7% of Internet traffic being QUIC (2017), marking a significant milestone in the evolution of Internet transport protocols. The protocol's overall performance gains and scalability have been demonstrated convincingly through empirical testing and deployment. The protocol used by Google before stan-

dardization is known as Google QUIC (gQUIC).

Following the deployment experience of QUIC at Google, an IETF working group was formed to standardize QUIC, resulting in RFC 9000, also referred to as IETF QUIC version 1 [3]. The paper 'Quick look at QUIC' by Dellaverson *et al.* [8] serves as an RFC explainer, aiming to elucidate the core ideas behind QUIC's design. Apart from explaining features of standardized QUIC, they look into the insights gleaned from the design of other transport protocols such as T/TCP, SCTP, RTP, TLS, and DTLS which has helped QUIC. The paper underscores that the concepts embodied in QUIC are a culmination of insights gained from decades of networking experimentation and prior protocol designs.

For instance, inspired by T/TCP, QUIC preserves and leverages connection states to enable 0-RTT communication. Drawing from RTP, QUIC operates over UDP to remain outside the kernel. Similarly, akin to SCTP and HTTP/2, QUIC employs multiple substreams to alleviate HOL blocking and the streams to facilitate multiple independent exchanges of packets. Adopting these ideas and synthesizing them into a single protocol allows the QUIC protocol to minimize latency (0-RTT) and address problems encountered in other transport protocols (e.g., HOL and entrenchment of TCP).

While Google introduced QUIC and presented its performance results for search and video, the paper by Kakhki *et al.* [5] focuses on analyzing the performance of gQUIC across a variety of network conditions. The work presents a methodology for generating inferred state machine diagrams for the transport protocol. The authors use this methodology to analyze the root cause for the performance benefits of QUIC over TCP.

The experimental setup involved an EC2 instance hosting a Chrome QUIC server and an Apache server, with clients running Chromium in a desktop environment (similar to Figure 3.1 from Section 3.2, but over a live network). They use simple web pages with single and multiple objects, to isolate the impact of parameters such

as the number and size of objects. They measure performance using page load time (PLT), representing the time taken to download all objects on a page.

Notably, we use the same methodology, metrics, and measurement scripts from Kakhki *et al.* [5] to measure the performance of QUIC in our work. In their experiments, the authors compare gQUICv34 with TCP and also acknowledge that gQUICv37 had a similar performance to gQUICv34. Our work benchmarks gQUICv37, before extending the study to QUICv1, to assess the impact of changes in the protocol over time. Kakhki *et al.* [5] observe that QUIC generally outperforms TCP on desktop environments, except in the jitter scenario (c.f., Figure 8 [5]). This superiority is attributed to QUIC's 0-RTT capability and efficient loss recovery mechanisms, driven by its accurate RTT estimation for different network conditions. However, QUIC exhibits sensitivity to out-of-order packet delivery, treating such occurrences as losses. We validate these results in Sections 4.3, 4.4 and 4.5. Also, examining the impact of the updated loss detection mechanism under jitter in QUICv1 in Section 4.6.

Kakhki *et al.* [5]'s findings reveal that QUIC's performance advantage over TCP persists across a range of bandwidths, although the improvement diminishes at higher bandwidths. Notably, on mobile devices (c.f., Figure 12 [5]), QUIC's performance is affected by application-layer packet processing and encryption, leading to a slowdown attributed to high resource utilization in the application space. Additionally, the authors discover that QUIC consumes more than twice the fair share of bottleneck bandwidth compared to TCP, indicating unfairness (c.f., Figure 4 [5]). We look at this issue of fairness in Section 4.1.

Yu and Benson [27] studied the performance of QUIC and TCP against production endpoints hosted by Google, Facebook, and Cloudflare. They find that QUIC's performance is largely dependent on the server's choice of CCA. As an example, they show that Cloudflare's H3 (QUIC) lagged behind H2 (TCP). This is attributed to the use of BBR CCA in H2, while H3 uses CUBIC. The authors also share that the QUIC

client configurations play an important role in optimizing QUIC performance. For example, ngtcp2's [28] default P256 TLS cipher group, which is incompatible with Facebook Proxygen's default X25519 TLS cipher group, resulted in ngtcp2 needing an extra RTT to resend the correct cipher group. Their observations show that QUIC's performance is inherently tied to implementation design choices, bugs, and configurations, and QUIC measurements are not always a reflection of the protocol.

Besides the performance studies of QUIC, some studies have focused on variations in QUIC implementation and their impact. For example, Marx *et al.* [26] compares 15 IETF QUIC and HTTP/3 implementations and finds that there is large heterogeneity between QUIC stacks. Additionally, they introduce qlog and qvis tools to generate QUIC logs and visualization QUIC connections, to facilitate root-cause analysis of different QUIC implementation behaviour. Mishra *et al.* [22] study the speciation in QUIC CCA implementations with respect to the reference (kernel) implementation. They find significant deviation between the existing QUIC implementation of standard CCAs (CUBIC, BBR, Reno) from the reference implementations. However, in the following work [29] on 11 popular open-source QUIC stacks, they find that most QUIC CCA implementations are conformant to kernel implementation in shallow buffers, but less so in deep buffers.

While most studies either only benchmarked Google QUIC (gQUIC) [5] [7] or the early draft versions of IETF QUIC [27][26] or both [30]. To the best of our knowledge, there is no prior work comparing the older gQUIC and standardized IETF QUIC [3], to assess the changes in a single implementation over 5 years. Our study of gQUICv37 and QUICv1 compares the performance of the two versions of QUIC against the corresponding TCP version to assess the impact of the changes in the protocol over time.

Understanding how QUIC behaves in different environments is valuable, but reproducing such studies can be challenging due to the variability in methodologies and setups provided. For example, Kakhki *et al.* [5] used a local server over a live

network, while Yu and Benson [27] used a production server, making replication difficult. Although Yu and Benson [27] provided setup information for reproduction, the workloads were on a production server that no longer exists. Moreover, relying on servers not controlled by the researchers introduces uncertainty due to the constant evolution of QUIC implementations.

With our work in Emulab, we have configured the profile to enable the same experiment to run effortlessly while still maintaining pluggable aspects such as workloads, protocol versions, and network conditions. We provide all the necessary components including workloads, servers, clients, testbed configurations [31], and automation scripts [32] to facilitate the complete reproduction of the experiments. Further, during this work, we found that Chromium is dropping support for gQUIC versions [33]. This makes it the right time to measure the performance of older gQUIC before moving on to the future IETF version of a continuously evolving QUIC protocol.

2.3 Concluding remarks

In this chapter, we discussed the few important features of QUIC and how they address the problems of TCP. We also surveyed the related works that have studied the features, performance, and implementation of QUIC. In the next chapter, we will discuss the methodology and setup used to measure the performance of QUIC and TCP.

Chapter 3

Experimental Methodology

In this chapter, we present the methodology, tools, and workloads used in our experiments. Building upon the work of Kakhki *et al.* [5], but in a different environment (i.e., Emulab), we extend their experiments to include IETF QUIC version 1 (QUICv1). For simplicity and clarity, we will refer to Google QUIC as gQUIC (e.g., gQUICv34, gQUICv37), IETF QUIC (as defined in RFC 9000 [3]) as QUICv1 (i.e., version 1 as QUICv1) and just QUIC to refer to the protocol. To enable easier replication of our work and extension of this methodology to future QUIC versions, we share the reproducible the Emulab experimental profile.

3.1 QUIC Versions

We now explore the two QUIC versions of the QUIC protocol utilized in this study: Google QUIC version 37 (gQUICv37) and IETF QUIC version 1 (QUICv1). While both share the fundamental QUIC design, they possess distinct features (e.g., different crypto protocols for encryption) and functionality (e.g., QUICv1 is general-purpose transport, while gQUICv37 was only used for the web) stemming from their separate development paths. Previous work conducted in 2017 [5] did not include QUICv1, which was finalized in 2021 [3]. Therefore, we benchmark QUICv1 to assess its performance in relation to the TCP in 2024. To understand how QUIC has evolved since 2017, we compare each QUIC version (gQUICv37 and QUICv1) to the TCP imple-

mentation of its respective period. QUIC version 2 (QUICv2) [4] was standardized in May 2023. As per RFC 9369, "QUIC version 2 is meant to mitigate ossification concerns and exercise the version negotiation mechanisms." [4]. Since this does not define or enhance any performance features, we do not include QUICv2 in our performance benchmarking.

3.1.1 Google QUIC (gQUIC)

Since Google introduced the QUIC protocol to the world, Google's implementation of the protocol (gQUIC), formed the majority of QUIC traffic on the Internet [7]. However, the IETF QUIC working group subsequently took the lead in standardizing the protocol, resulting in a version distinct from gQUIC (Figure 1.1). Post-IETF adoption of the protocol, Google then started to converge Google QUIC to IETF QUIC. This convergence began with gQUIC version 44 [34], which incorporated all changes outlined in the IETF invariants draft [35]. While subsequent versions of Google QUIC were based on the IETF QUIC standard, Google QUIC was still different from IETF QUIC in many ways. For example, Google QUIC used Google's QUIC Crypto library, while IETF QUIC used TLSv1.3 for packet encryption (Table 3.2 QUIC crypto row).

For our experiments, we employed gQUICv37, because it represents the version of gQUIC before the transition to IETF QUIC, and there is prior work that established the performance of gQUICv37 [5]. We accessed gQUICv37 of 2017 by leveraging an older Chromium codebase and browser that supported this earlier version. The specific Chromium versions used are detailed in Section 3.2.3. During our experimentation with Google QUIC, we found that the latest Chromium (i.e., version 112) is deprecating gQUIC in favor of IETF QUIC, by dropping support for all pre-IETF versions of QUIC (gQUIC and non-TLS code paths) [33]. As a result, starting from Chromium version 112, we found that the Chromium browser did not connect to the server running gQUIC. Hence, our work is useful in capturing the performance difference between these two versions of QUIC (gQUICv37 and QUICv1) before gQUIC is

fully phased out.

In our evaluation, we only benchmark version 37 of Google QUIC (gQUICv37). Previous work by Kakhki *et al.* [5] demonstrated that gQUIC versions 34 and 37 exhibited similar performance characteristics. They reported that the primary difference between these versions was an increase in the Maximum Allowed Congestion Window (MACW) from 430 to 2000 in version 37.

3.1.2 IETF QUIC (QUIC)

IETF QUIC [3] represents the standardized and evolved version of the Google QUIC [1]. While retaining the core design principles of QUIC, numerous significant changes have been implemented. Notably, IETF QUIC leverages TLSv1.3 for packet encryption, differing from gQUIC's use of its own crypto library.

While Google QUIC was started as an alternative to TCP+TLS+HTTP/2, primarily to improve web page load times, IETF QUIC defined a clear boundary between the transport layer and the application layer. RFC 9000 [3] established QUIC as a general-purpose transport layer protocol. HTTP/3, as described in RFC 9114 [11] was the successor of HTTP/2 to use QUIC as its transport layer.

We utilized QUICv1, implemented within Chromium, as our second QUIC version for the experiments (see Table 3.2, column 'Extended'). While the fundamental design of QUIC remained the same, there were significant changes in the protocol. As we will see in Section 4, the performance benefits of QUIC over TCP, given by fundamental features like 0-RTT (Section 2.1.1) and HOL blocking (Section 2.1.2), have remained consistent across the two versions (Section 4.3 and 4.4). However, the performance of QUICv1 has improved over gQUICv37 in scenarios with jitter, where there were changes to loss detection strategies under packet reordering (Section 4.6).

3.2 Experimental Setup on Emulab

In this section, we detail the components of our experimental setup (Figure 3.1). Starting with the Emulab as our hardware component, we then discuss Dummynet, which we use to shape the network traffic. Next, we discuss the software involved, such as Chromium as our QUIC implementation and other tools for traffic measurement (e.g., chrome-har-capturer for HAR file.). Finally, we present the differences in experimental setup and protocol versions compared to the original work by Kakhki *et al.* [5].

3.2.1 Emulab testbed

Emulab is a network emulation testbed facilitating repeatable research [36]. It provides researchers with complete control over the experiment testbed, allowing them to develop, debug, and evaluate their system in a controlled environment. With the help of an Emulab profile [37], you can specify an arbitrary network topology, giving you a controllable, predictable, and repeatable environment. While there are other installations of Emulab, we use the primary installation [38] run by the Flux Group, part of the School of Computing at the University of Utah.

The Emulab profile allows users to define the specific configuration of their experiments, including network topology, node setup, and software environment. The profile can be created using either a graphical user interface (GUI) or through Python scripts using the `geni-lib` library. Once created, profiles can be shared with others, allowing researchers to collaborate and reproduce experiments. By having the experimental setup and configuration defined in a script, it is easy to have different versions of the same experiment. For example, the "Replicated" and "Extended" experiments of Table 3.2 are two different versions of the same profile. The profile also facilitates the automation of software installation.

With reproducibility as one of our goals, all our experiments are conducted on the

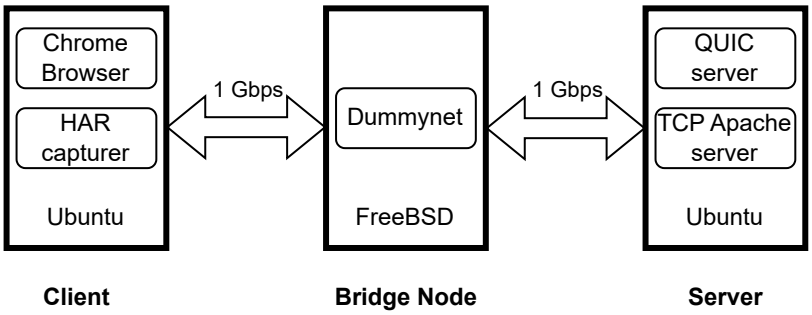


Figure 3.1: Experiment Topology in Emulab. The server and client are connected via a link bridge node, which shapes the traffic.

Emulab testbed [36]. As shown in topology Figure 3.1, the setup comprises three physical nodes: a server, a client, and a link bridge node. The server and client are designated as Emulab nodes of type d430 and d710, respectively [39]. The d430 node has 64 GB memory and two Intel E5-2630v3 8-Core CPUs at 2.4 GHz. The d710 node has 12 GB memory, 8 Core Intel(R) Xeon(R) CPU E5530 @ 2.40GHz. Both the server and client node run Ubuntu 22.04, while the link bridge node, connecting them, operates on FreeBSD 13.2. Since Dummynet comes pre-installed on FreeBSD, the Emulab "Bridge Node" (Figure 3.1) runs FreeBSD.

The server hosts both the TCP and the QUIC server, serving HTTP/2 and HTTP/3 requests, respectively. Since both servers are on the same hosts, we make sure that both TCP and UDP reach the desired bandwidth and that all system configurations are the same to ensure there is no unfair advantage for either protocol. We present the precise socket buffer sizes and other system configurations in Section 3.2.4. Apache is used as the HTTP/2 server (with default Linux TCP stack configuration), and the QUIC sample server from Chromium [40] is used as the QUIC server. Additionally, Apache is configured to serve HTTPS requests using TLS, to make a fair comparison with QUIC, which has encryption built in. Throughout this thesis, we refer to measurements that include HTTP/2+TLS+TCP as "TCP" and HTTP/3 as "QUIC".

Our client has Chromium as the browser and chrome-har-capturer tool [41] for

request timing measurement. Both Chromium and HAR capture tool are further discussed in Section 3.2.3 and 3.2.3 respectively. Apart from the browser, the client also has the necessary network tools to measure and validate the network parameters. For instance, we use iperf [42] to measure the bandwidth of the link between the server and client and ping to measure the latency and packet loss. tcpdump [43] is used to capture the network traffic between the server and client, for measuring the throughput and packet inspection for further analysis.

To shape traffic between the server and client, we utilize Dummynet [44], a widely used link emulator. Dummynet, running on the link "bridge node", enables us to precisely control network parameters such as bandwidth, latency, and packet loss. We expand on the Dummynet usage in Section 3.2.2.

The two key benefits of using Emulab for our work are: Firstly, it allows us to define the network topology we need in the form of a Python script using geni-lib [45], a Python library for defining Emulab resources. Secondly, The Emulab experiment profile that supports geni-lib can be shared with other researchers to replicate the same experiments in Emulab. We have shared our experiment profile [46], which when instantiated in Emulab or any testbed that uses components of the GENI Software Architecture [47] produces a ready-to-use experimental setup. The shared profile, workloads combined with methodology, and updated scripts [48] of Kakhki *et al.* [5] our entire setup can be recreated with minimal effort and with configurable options like choice of server, client, workloads, and network parameters. With this precise network testbed, the results presented can not only be replicated but also used as the basis for evaluation of future QUIC versions (Figure 1.1).

3.2.2 Dummynet traffic emulator

Dummynet [44] is a commonly used link emulator that relies on ipfw [49] for packet classification. Upon interception in the network stack, packets are directed based on ipfw rules to their respective pipes. Pipes are the core unit of Dummynet, responsible

for shaping the traffic after receiving packets from their corresponding ipfw rules.

Within a pipe, incoming packets are queued if capacity allows, with the queue being emptied at a rate dictated by the link's bandwidth. After exiting the queue, packets undergo a specified propagation delay before being reintroduced into the network stack. Additionally, Dummynet can perform non-congestion-related drops based on the given loss probability, effectively emulating links with uniform random loss patterns.

Emulab [36] uses Dummynet to emulate WAN links between nodes. The Emulab resource mapper automatically inserts Dummynet-enabled Bridge Nodes [50] between two physical nodes to shape traffic based on the specified network parameters. Bridge nodes act as Ethernet bridges, so they are transparent to experimental traffic.

We use Link Bridge Nodes [50] (Figure 3.1) for traffic shaping between server and client, to avoid the pitfalls of enforcing traffic shaping at endpoints as noted in Kakhki *et al.* [5]. The Bridge Nodes [50] also provides a Dummynet wrapper script called `delay_config` which helps us to easily update the network parameters like BANDWIDTH, DELAY, LOSS (PLR). The `delay_config` script accepts the Emulab project name, experiment name, and the link name (bridge) as arguments, and then applies the specified network parameters to the link.

```
delay_config -b QUIC-BENCH EXP1  \\
link_bridge BANDWIDTH=10000 DELAY=36 PLR=0.01
```

Listing 3.1: Sample Delay config command to set link to 10 Mbps, 36 ms RTT and 1% loss.

In Listing 3.1, we show Emulab `delay_config` traffic shaping command to limit link bandwidth to 10000 Kb/s (10 Mbps), with 36 ms RTT and 1% loss (Example parameter of Table 3.3, setting of 10 Mbps row of Figure 4.5b), for the link named `link_bridge` in the experiment `EXP1` of `QUIC-BENCH` project.

3.2.3 Chromium QUIC Stack

We use Google's Chromium QUIC as the QUIC stack for our experiments. Chromium is an open-source browser project behind the Google Chrome browser and was the sole browser to support QUIC in 2017 [5]. The Chromium project is one of the few implementations that support both early versions of Google QUIC and standardized IETF QUIC (see row "Chromium browser", Table 3.2), both of which are essential for our experiment. Additionally, the Chromium project provides both server and client implementations of QUIC [40].

However, working with Chromium's large codebase can be cumbersome. Downloading the complete codebase and its history of tags and branches results in a sizable 100 GB of data and requires several hours. Furthermore, Chromium lacks support for building older versions of the code, specifically those predating version 68 (release branch 3420)[1]. This presented a challenge as we needed access to the Chromium 60, which supports gQUICv37. Consequently, we sought an alternative approach to building the entire Chromium codebase.

Fortunately, the Chromium project offers a standalone QUIC implementation, facilitating experimentation with QUIC without compiling the entire Chromium codebase. Leveraging this, we built the QUIC sample server (Figure 3.1) [40] from the standalone repository. Whereas, we obtained pre-built Chromium browsers as clients. The smaller codebase of the standalone repository enabled us to instrument the QUIC server for logging essential information during our experiments. For example, we added logging to get packet loss count and congestion window size for the QUIC server, which aided our analysis of QUIC fairness and performance.

Before Chromium version 68, proto-quic [51] served as a standalone repository for QUIC code. However, following 2019, the proto-quic project was archived in favor of Google quiche [52], an upstream repository for QUIC code used in Google servers

[1] "Getting started with release branches". Retrieved from https://www.chromium.org/developers/how-tos/get-the-code/working-with-release-branches/ [accessed on 2024-03-05]

and the Chromium browser. Therefore, to build the 2017 version of the gQUICv37 server we use proto-quic and Google quiche to build the current QUICv1 server.

For the Chromium client, we utilized Chromium browser versions 60.0.3108.0, supporting gQUICv37, and version 111.0.3112.101, supporting QUICv1 (see row "Chromium browser" of Table 3.2), both obtained from the official Chromium build page for Linux[2].

Despite the standardization of QUIC, there are variations in QUIC implementations, including differences in design choices [26] and implementation of CCAs [22]. As a result, we have chosen to evaluate only a single QUIC implementation in our study. This decision allows us to focus solely on assessing the protocol evolution from Google QUIC to IETF QUIC within the Chromium project.

HAR Capturer

We utilize the Chromium browser's ability to capture network-level metrics by generating HTTP Response Archive (HAR) [53] files for each request. These HAR files enable us to measure the page load time (PLT) of web pages accurately. They help us validate, if our requests are being served using the desired protocol (HTTP/2 or HTTP/3) and that the requested resources are transmitted over the network without any compression or caching.

To capture these metrics, we employ the chrome-har-capturer tool [41], which automatically generates HAR files for each HTTP request made by the browser. This tool allows us to automate the process by running the browser in headless mode and capturing the HAR file of the request. We then use the HAR file to extract the resource timings and calculate the PLT of the web page, facilitating our performance evaluation. As shown in row "HAR Capturer tool" of Table 3.2, appropriate versions of the chrome-har-capturer tool are used for each Chromium browser version, ensuring compatibility with the respective browser.

[2] "Download Chromium". Retrieved from https://www.chromium.org/getting-involved/download-chromium/#downloading-old-builds-of-chrome-chromium [accessed on 2024-03-05]

To enable 0-RTT connections in QUIC, the client must retain server information from the previous connection. However, in our benchmarking experiments, each request is initiated from a new browser instance to prevent caching and the reuse of previous TCP/UDP connections. This approach, aimed at ensuring consistent testing conditions, inadvertently prevented our Chromium browser from attempting 0-RTT connections in QUIC experiments, even for subsequent connections to the same server. To address this issue, we modified the default behavior of the chrome-har-capture [41] tool by leveraging its user script feature.

To ensure 0-RTT connection our custom user script makes use of a single browser context for all measurements. Additionally, the script cleared the cache and closed connections between subsequent measurements, preventing their reuse. As a result, our first measurement involved launching a new Chromium browser instance to capture PLT values. However, since this initial fresh browser lacked sufficient information about the server (Chromium HTTPServerProperties), it was unable to attempt a 0-RTT connection. Subsequent measurements were conducted using the same browser context after cache clearance and connection closure. In this scenario, the Chromium browser possessed the necessary information about the QUIC server, enabling it to attempt 0-RTT connections. Consequently, to accurately represent QUIC features in our measurements, we discarded the initial measurement that used a 1-RTT connection. In total, we conducted 21 runs for each scenario. The initial measurement was discarded, and the remaining 20 runs were used in our experiments. However, this initial measurement is included for comparison purposes when analyzing 1-RTT versus 0-RTT PLTs of QUIC (see Section 4.3.1).

3.2.4 Setup Differences

In contrast to previous work by Kakhki *et al.* [5] (2017), our experimental setup differs in several key aspects as detailed in Table 3.1. While Kakhki *et al.* [5] used a desktop as the client and Amazon EC2 for the server, we utilize Emulab nodes

	Kakhki '17 [5]	This work
Client Machine	Desktop	Emulab Node
Server Machine	Amazon EC2	Emulab Node
Network Type	Live Network	Emulated Ethernet Bridge
Traffic Shaping	TC Netem	Dummynet and TC Netem
Workload	HTML web pages with Images	HTML web pages with Images

Table 3.1: Differences in experimental setup components. The use of Emulab and Dummynet enables reproducible experiments.

for both client and server components. Moreover, Kakhki *et al.* [5]'s experiments were conducted on a live network, with traffic shaping achieved using TC Netem, whereas we employ an emulated network environment provided by Emulab, with traffic shaping implemented using the Dummynet. We utilize Dummynet for all aspects of traffic shaping. However, Dummynet within the link bridge node does not support updating latency at a sufficiently high rate to introduce jitter. So, we use TC on the client node to achieve this functionality.

Since we evaluate two QUIC versions, we use two versions of the same setup as detailed in Table 3.2. In our first setup (Table 3.2, column "Replicated" for gQUICv37 experiments), aimed at replicating the conditions of the previous work by Kakhki *et al.* [5], we had to go back in time to evaluate older Google QUIC version 37 from 2017. Running the older gQUIC version posed several challenges. Firstly, locating the appropriate Chromium version that supported gQUICv37 and its corresponding Chromium browser client was a hurdle, as older gQUIC versions are phased out and are not readily available in the latest code base. Secondly, we had to set up the appropriate build environment to compile and build the old Chromium code base. For instance, Chromium from 2017 would only build on Ubuntu 14, 16, or 18, making it necessary to find compatible tools and libraries for the older software stack. Lastly, auxiliary tools like chrome-har-capturer had to be compatible with the older Chromium version to capture the HAR files for performance analysis. We detail the

specific software versions used in Table 3.2.

To gain access to the gQUICv37 protocol code, we used the old Chromium release branch version 60.0.3108.0 (60.0.3112.101, used by Kakhki *et al.* [5] was not available in proto-quic repo [51]). The build scripts of Chromium from 2017 were not supported in the latest Ubuntu 22.04. Hence, we use Ubuntu 18.04 with Linux kernel 4.15.0-204-generic (Ubuntu 14.04, used by Kakhki *et al.* [5] was deprecated in Emulab). Additionally, to encrypt HTTP/2 traffic, we utilized TLSv1.2 (TLSv1.3 was not available in 2017) to be comparable with gQUICv37 which uses QUIC Crypto for encryption.

In our second setup (Table 3.2, column "Extended" for QUICv1 experiments), the same experimental topology (from Figure 3.1) was used, but with updated versions to benchmark QUICv1. In contrast to the efforts required to set up the gQUICv37 environment, the QUICv1 setup was relatively straightforward. Since QUICv1 is the current version, the Chromium browser and corresponding QUIC server code were readily available in the latest Chromium codebase with support for the latest Ubuntu 22.04. Here, we utilized the latest Chromium browser version 111 and obtained the QUIC server code from the corresponding upstream branch of Google quiche [52]. We also employed the latest compatible versions of the chrome-har-capture tool, ensuring compatibility with Chromium and the supported Chrome Debugging Protocol. For HTTP/2, we upgraded to TLSv1.3, matching the encryption used by QUICv1 to make a fair comparison.

3.2.5 Emulab Setup Calibration

We chose Emulab as our testbed for its ability to provide a controlled and reproducible environment for our experiments. Our choice of an emulated test-bed presented its own challenges, which required some calibration listed below.

1. **Latency:** We add an implicit latency of 36 ms (validated via ping) to the link between server and client in Emulab, to match the typical WAN latency

	Kakhki '17 [5]	Replicated	Extended
QUIC version	gQUIC v37	gQUIC v37	IETF QUIC v1
TCP version (Kernel)	4.4.0-34-generic*	4.15.0-204-generic	5.15.0-56-generic
QUIC server (Chromium)	60.0.3112.101ˆ	60.0.3108.0	C43017f
TCP server (Apache)	Apache 2.4	Apache/2.4.29	Apache/2.4.52
QUIC crypto library	QUIC Crypto	QUIC Crypto	TLSv1.3
TCP crypto library	TLSv1.2	TLSv1.2	TLSv1.3
Operating System	Ubuntu 14.04*	Ubuntu 18.04	Ubuntu 22.04
Chromium browser	60.0.3112.101ˆ	60.0.3108.1	111.0.5563.0
HAR Capturer tool	Unknown	0.9.5	0.14.0

Table 3.2: Differences in software versions for gQUICv37 and QUICv1 experiments. (*) Ubuntu 14.04 was deprecated in Emulab, hence we used Ubuntu 18.04 for gQUICv37 experiments. (ˆ) Chromium 60.0.3112.101 was not available in proto-quic repo, so we used the closest available version 60.0.3108.0.

reported earlier [5].

2. **Bandwidth:** In the default settings of the Dummynet the throughput in higher delays was not close to the desired bandwidth. Hence, we adopt calibration from previous work [54] to reach the desired bandwidth (validated via iperf), by increasing socket buffers and Dummynet queue sizes to match the bandwidth-delay product (BDP) of our link.

Kakhki *et al.* [5] observed QUIC outperforming TCP in baseline scenarios (36 ms RTT, 0% loss - Section 4.3) due to an implicit latency of 36 ms present in their live network. However, our testbed had a significantly lower implicit latency of around 0.5 ms to 2 ms due to the direct connection between the server and client via a bridge node (Dummynet pipes). To align with Kakhki *et al.* [5], we added a default implicit latency of 36 ms in our experiment profile. Before each experiment, we validated the latency using the ping command to ensure that the desired latency was present.

To achieve the desired bandwidth in Emulab under high BDP settings, we adjusted the system configurations. Additionally, bandwidth measurements were conducted

using iperf [42] to ensure that the desired bandwidth was achieved. We ensured that the link could hold 2x BDP of data throughout its path by increasing the socket buffer sizes on all three nodes and enlarging the Dummynet queue size accordingly. For instance, in a setting with 100 Mbps and a 112 ms RTT, we set the buffer size and queue to 2800000 bytes ($2 * ((100 * 10^6 * 112 * 10^{-3})/8)$). We utilized previous work by Jones [54] to configure the Dummynet queue size.

Apart from the calibration to our Emulab's experimental setup, other software calibrations were needed for a fair comparison with previous work [5]. Kakhki *et al.* [5] had changed certain configurations of the QUIC sample server (Kakhki *et al.* [5] uses the term "toy server") to calibrate its performance close to Google servers. However, upon our investigation, we found that those configurations and bug fixes were already present in the current Chromium source code (Chromium version 111). Namely, the slow start bug has been fixed [55] and the maximum congestion window size has been increased [56] [57]. Our investigation in this regard was aided by previous replication work by Wong and Tieu [58].

3.3 Parameters, Workloads and Metrics

In this section, we present the parameters, workloads, and metrics used in our experiments to evaluate the performance of QUIC in a desktop environment.

Table 3.3 outlines the parameters considered for our tests, including variations in RTT (36 ms, 112 ms), packet loss rates (1%), and bandwidths (10 Mbps, 50 Mbps, 100 Mbps), as well as webpage sizes and the number of objects per page. While Kakhki *et al.* [5] considered additional setup and workloads such as proxy servers, mobile environment, and video streaming performance, we focus on the parameters listed in Table 3.3 to evaluate the PLT performance of QUIC in a desktop environment. We also do not consider 0.1% packet loss, as this was reported in Kakhki *et al.* [5] while testing cellular networks.

In addition to the parameters listed in Table 3.3, we also consider the impact of

Parameter	Values Tested	
	Kakhki '17 [5]	**This work**
Rate limits (Mbps)	5, 10, 50, 100	5, 10, 50, 100
Net Delay (RTT)	36ms, 112ms	36ms, 112ms
Extra Loss	0%, 0.1%, 1%	0%, 1%
Number of objects	1, 2, 5, 10, 100, 200	1, 2, 5, 10, 100, 200
Object sizes (KB)	5, 10, 100, 200, 500, 1000, 10,000, 210,000	5, 10, 100, 200, 500, 1000, 10,000, 210,000
Proxy	QUIC proxy, TCP proxy	None
Clients	Desktop	Emulab Node
Video qualities	tiny, medium, hd720, hd2160	None

Table 3.3: Differences in experimental parameters.

packet reordering on QUIC performance (Section 4.6). In our experiments, we cause packet reordering by introducing a jitter, which is created by applying continuous random latencies within a specific range (50 ms) to the link connecting server and client. This random variation in latency causes different packets to experience different latencies, resulting in packet reordering. To ensure a more rapid application of latency, we use the `tc` command on the local client interface to add random latency. This approach is preferred over Dummynet at the bridge node, as it provides a higher frequency of updates, which is essential for effective packet reordering.

We use the same type of workloads as Kakhki *et al.* [5], which consist of simple web pages comprising static HTML files referencing JPG images, of varying sizes and numbers. To prevent caching and compression, we include all necessary HTTP directives. For TCP, this involves modifying the Apache configuration (e.g., Header set Cache-Control "no-store,no-cache"), while for QUIC, we prepare web pages with embedded HTTP directives (e.g., cache-control: no-store,no-cache), due to the basic functionality of the QUIC sample server. To assess the impact of multiple objects

on webpage loading, we embed multiple image files within a single HTML page. In the TCP case, we use the Flask web framework to render HTML with multiple object references, while for QUIC, we manually reference the objects multiple times within the HTML file. All our workloads are synthetic and are shared as part of the replication package [48].

Since QUIC is often used with HTTP/3 to deliver user-facing web content, we focus on web PLT as a metric. PLT is the request-response latency obtained by measuring the total duration from the time the client request started to the time when the web page is completely loaded. PLT values are extracted from the "onLoad" field of HAR file [53] produced by chrome-har-capturer tool [41] (Section 3.2.3). We make sure to exclude DNS lookup time from the total timing, to ensure we only capture the resource loading time.

We run the Chromium browser in headless mode and make requests via the chrome-har-capturer [41], which connects to the browser using Chrome's remote debugging protocol. As described earlier, we utilize the chrome-har-capturer tool to capture resource timings for each request. All measurements are automated through a Python script, and the testbed is defined using the Emulab experimental profile [31]. To facilitate result reproducibility, we provide our synthesized workloads of web pages as part of our replication package on GitHub [48], which were absent in previous work [5] [27].

3.4 Concluding remarks

In this chapter, we detailed the components of our experimental setup (Figure 3.1), including the Emulab testbed, Dummynet traffic emulator, Chromium QUIC stack, and the chrome-har-capturer tool. We also discussed the differences in our experimental setup compared to the original work by Kakhki *et al.* [5], and the calibration required to align our setup with the previous work. We then presented the parameters, workloads, and metrics used in our experiments to evaluate the performance of QUIC against TCP in a desktop environment. In the next chapter, we present the results of our experiments, comparing the performance of QUIC and TCP under various network conditions and workloads.

Chapter 4

Empirical Results

By design, QUIC promises to provide performance improvements to TCP. It is important to evaluate these improvements in actual implementation, to measure the effectiveness of the new features of QUIC. In this chapter, we discuss the empirical results of evaluating QUIC and TCP using web workloads. Additionally, we benchmark two different versions of QUIC (i.e., gQUICv37 and QUICv1) and explain the evolution of design and its performance impact.

In our fairness experiments, we show that the QUIC CUBIC Congestion Control Algorithm (CCA), especially in Chromium, can be unfair to TCP CUBIC. However, by changing a single parameter in QUIC CUBIC (N = 1), we can restore fairness. In the performance experiment, QUIC outperforms TCP in most cases (e.g., with added loss and latency, QUIC is better across bandwidths and workloads, see Figures 4.6b and 4.6c) except when there are a large number of small objects (Figure 4.6d, column 10 KB x 100). We also show that the new dynamic reordering threshold in QUICv1 is better than the static reordering threshold of gQUICv37, for loss detection under packet reordering scenario (Figures 4.11c and 4.11d).

4.1 QUIC Fairness

When multiple protocols are operating over a network, these transport-layer protocols need to be fair to each other by consuming only their fair share of bottleneck

bandwidth. For example, if a network link is shared between two protocol flows, then each protocol should consume half of the total bandwidth. An unfair protocol consuming more than its share may negatively impact the performance of the competing protocol flow.

While Carlucci *et al.* [59] and Kakhki *et al.* [5] studied the fairness between older versions of QUIC and TCP in 2015 and 2017, respectively, we evaluate the fairness with the current state of QUIC (i.e., QUICv1). In addition to the CUBIC CCA evaluation from previous studies, we also evaluate the fairness of QUIC with BBR CCA, which was not readily available in 2017 [5]. We initiate HTTP/3 and HTTP/2 + TLS connection from the client to the server simultaneously and measure the throughput of each protocol. The HTTP/3 protocol is referred to as QUIC, and HTTP/2 + TLS is referred to as TCP in our experiments. Below is the list of CUBIC and BBR scenarios that we tested, and their results are summarised in Tables 4.1 and 4.2, respectively.

4.1.1 CUBIC Fairness

1. **QUIC CUBIC vs QUIC CUBIC** Similar to previous studies [5], we find that two QUIC flows are fair to each other (Table 4.1, QUIC vs QUIC), and the same is true for two TCP flows under CUBIC CCA (Table 4.1, TCP vs TCP).

2. **QUIC CUBIC vs TCP CUBIC** The Chromium QUIC implementation we studied uses a CUBIC congestion control parameter called *connection emulation N*, which emulates N TCP connections for the Congestion Window (CWND) update. This N had a default value of N = 2 in gQUICv34 [5] and still has N = 2 in QUICv1.

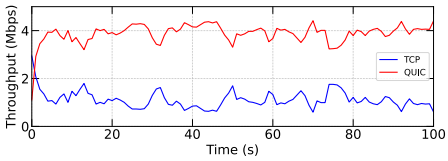
   As shown in the timeline Figure 4.1a, QUIC CUBIC (red line) is unfair to TCP CUBIC (blue line) by occupying twice the bandwidth as TCP CUBIC. While QUIC CUBIC on average occupies 3.74 Mbps, TCP CUBIC is only able

to occupy 1.24 Mbps (Table 4.1, QUIC vs TCP). This is expected, as N = 2 means QUIC emulates 2 TCP connections. We further changed the QUIC server (Table 3.2, row 'QUIC server', column 'Extended') to emulate 1 TCP connection (N = 1) to study the influence of CUBIC parameter N. From Figure 4.1b, we see that QUIC CUBIC and TCP CUBIC share almost equal amounts of bottleneck bandwidth and are fair to each other (Bandwidth numbers for N = 1 case is not shown in the Table). Hence, N directly influences the bandwidth-occupying capacity of QUIC CUBIC. So, QUIC is fair to TCP, when it is emulating a single TCP connection.

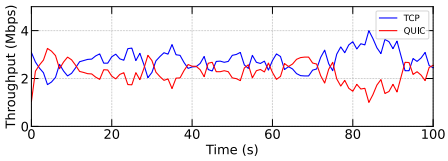
This result is different from the findings of Kakhki *et al.* [5], who reported that when N (connection emulation) was set to 1, it did not affect the fairness of QUIC and concluded that QUIC CUBIC was unfair to TCP CUBIC. But, our results show that when N is set to 1, QUIC CUBIC is fair to TCP CUBIC. It is important to note that we did not find the CUBIC parameter N, in other implementations of QUIC (e.g., ngtcp2), and it was only relevant to Google's implementation [52] (quiche).

3. **QUIC CUBIC vs multiple TCP CUBIC connections** To further validate the behavior of N, we repeat the experiment by having a single QUIC connection compete with multiple TCP connections. When there are M TCP connections and one QUIC connection, a QUIC connection should consume 2/(M+1) of the bottleneck bandwidth [5].

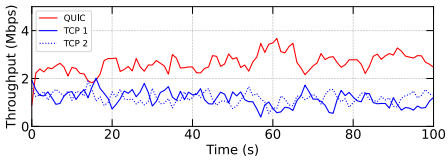
   From Figure 4.1c, we see that this holds true, where a single QUIC connection is occupying the bandwidth equivalent of two TCP connections (Table 4.1, QUIC vs TCPx2). The effect of N can be further validated from Figure 4.1d where a single QUIC connection with N = 1 roughly occupies equal bandwidth as the two other TCP connections. In general, with QUIC CUBIC parameter set appropriately (i.e., N = 1), QUIC CUBIC is fair to TCP CUBIC.



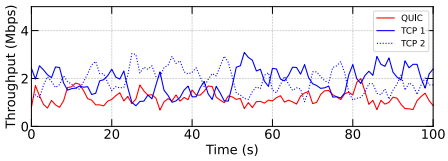
(a) QUIC CUBIC (N=2) vs. TCP CUBIC



(b) QUIC CUBIC (N=1) vs. TCP CUBIC

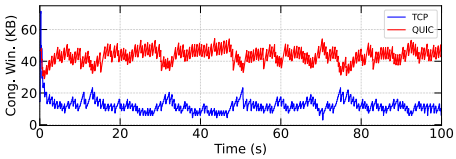


(c) QUIC CUBIC (N=2) vs. 2 TCP CUBIC flows

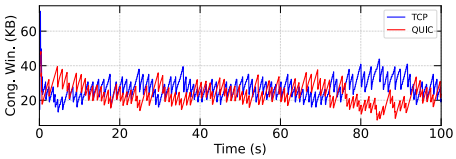


(d) QUIC CUBIC (N=1) vs. 2 TCP CUBIC flows

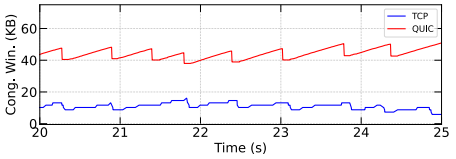
Figure 4.1: Throughput timeline, QUIC CUBIC is unfair to TCP CUBIC when N = 2. Throughput of QUIC and TCP when transferring data over the same 5 Mpbs bottleneck link (RTT = 36 ms, loss = 0 %, buffer = 30 KB).



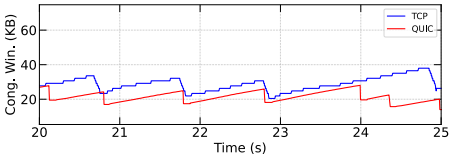
(a) QUIC CUBIC (N=2) vs. TCP CUBIC



(b) QUIC CUBIC (N=1) vs. TCP CUBIC



(c) 5-second zoom of above figure



(d) 5-second zoom of above figure

Figure 4.2: CWND timeline, the growth of QUIC's CWND is influenced by the CUBIC parameter N. Timeline showing CWND sizes of QUIC and TCP when transferring data over the same 5 Mbps bottleneck link (RTT = 36 ms, loss = 0 %, buffer = 30 KB).

To better understand the throughput behavior of these protocols and the effect of N, we conduct a similar analysis to that performed by Kakhki *et al.* [5] (c.f., Figure 5). This involves plotting the QUIC and TCP CWND during the throughput test. We instrument the QUIC server code (Table 3.2, row 'QUIC server', column 'Extended')

to extract CWND sizes for QUIC, whereas for TCP we use ftrace's tcpprobe event [60].

Figure 4.2 shows the CWND over time for TCP and QUIC. From Figure 4.2a, we observe that QUIC's CWND is larger than TCP's CWND because QUIC CUBIC is emulating two TCP connections (N = 2), whereas from Figure 4.2b, when N = 1, QUIC's CWND overlaps with TCP's CWND. The two CWND Figures 4.2a and 4.2b, make it evident that the QUIC's larger CWND which resulted in higher bandwidth utilization (Figure 4.1a) is a direct result of CUBIC parameter N. When we have a closer look at the CWND update as shown in Figure 4.2d, we can see that when N = 1, both QUIC and TCP grow their CWND uniformly.

Kakhki *et al.* [5] also tried modifying their QUIC implementation, but they were not able to achieve fairness. Hence, they reported that QUIC is unfair to TCP even with N = 1. However, our modifications to the Chromium code (version 60, gQUICv37) required changes to three, non-adjacent, easily overlooked locations of N in the source code. In QUICv1, which is used to produce the numbers in this discussion, there is only one instance of N in the source code that needs to be changed, but this version is from after 2017.

Our experiments demonstrate that QUIC CUBIC in Chromium is unfair to TCP in its default setting of N = 2. However, by changing N = 1, QUIC CUBIC becomes fair to TCP CUBIC. This observation aligns with the conclusions drawn by Mishra *et al.* [22], which underscores the significant improvement in Chromium CUBIC's adherence to the TCP kernel implementation when N is set to 1. The CUBIC (N = 2) fairness results for 5 runs are summarised in Table 4.1.

4.1.2 BBR Fairness

In 2016, Google introduced a new congestion-based CCA called BBR [61]. QUIC, due to user-space implementation and pluggable CCA, was quick to adopt BBR in its early stages. While, BBR (i.e., BBRv1) was not available in Linux 4.4.0-34-generic (2017), it is now available in Linux 5.15.0-56-generic (Table 3.2, TCP version). Hence,

Scenario	**Flow**	**Avg. Throughput (std. dev.)**	**Fairness**
QUIC vs QUIC	QUIC 1	2.45 (0.04)	fair
	QUIC 2	2.51 (0.04)	
TCP vs TCP	TCP 1	2.46 (0.03)	fair
	TCP 2	2.52 (0.03)	
QUIC vs TCP	QUIC	3.74 (0.14)	QUIC is unfair
	TCP	1.24 (0.14)	
QUIC vs TCPx2	QUIC	2.59 (0.02)	QUIC is unfair
	TCP 1	1.21 (0.03)	
	TCP 2	1.18 (0.01)	
QUIC vs TCPx4	QUIC	1.59 (0.03)	QUIC is unfair
	TCP 1	0.82 (0.03)	
	TCP 2	0.88 (0.01)	
	TCP 3	0.82 (0.01)	
	TCP 4	0.85 (0.02)	

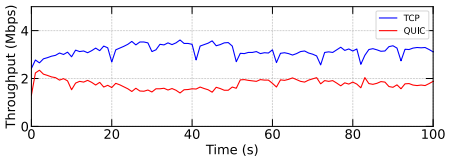
Table 4.1: Average throughput (5 Mbps link, RTT = 36 ms, loss = 0 %, buffer = 30 KB, averaged over 5 runs) allocated to QUIC and TCP flows when competing with each other. When both TCP and QUIC are using CUBIC congestion control, the unfairness caused by QUIC flow is simply due to N = 2 connection emulation.

we extend the fairness experiment from Kakhki *et al.* [5] to include BBR as CCA for both QUIC and TCP. All our experiments use BBR version 1 (BBRv1), which is simply referred to as BBR in this chapter.

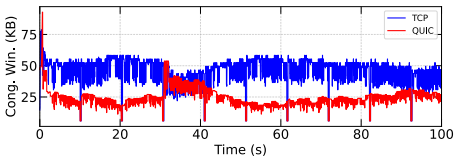
1. **QUIC BBR vs TCP BBR** Despite both QUIC and TCP using BBR, we find that TCP BBR can occupy slightly more bandwidth when compared to QUIC BBR as shown in Figure 4.3a. To understand this slight difference in bandwidth sharing, we look at the CWND graph in Figure 4.3b which shows that the blue TCP CWND for the most part sits above the red QUIC CWND. The reason for unfair bandwidth occupancy becomes clear when we have a closer look at this CWND update as shown in Figure 4.3c. Here we can see the TCP BBR CWND is getting updated at a faster rate when compared to QUIC BBR CWND (the frequency of CWND update is high). This allows TCP to gain marginally more bandwidth than QUIC. Although the fast CWND update of TCP BBR is observed, it is not clear why QUIC BBR is unable to match the CWND update frequency of TCP BBR.

2. **QUIC CUBIC vs TCP BBR** Previous studies have shown BBR being unfair to CUBIC [62] [63]. To observe its behavior from two different protocols, we conducted the same fairness experiment by running TCP with BBR and QUIC with CUBIC (N=2). From Figure 4.4 we found that irrespective of the protocol, BBR is unfair to CUBIC even when the CUBIC algorithm emulates 2 connections (N = 2).

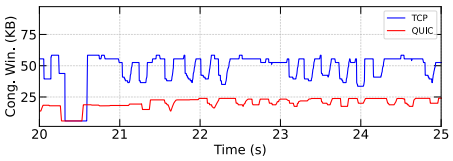
In our BBR results, we observe that TCP BBR against QUIC BRR is unfair, and so is TCP BRR against QUIC CUBIC. Specifically, we found that QUIC BBR (1.74 Mbps) occupies lesser bandwidth than TCP BBR (3.18 Mbps) (Table 4.2, QUIC BBR vs TCP BBR). On the other hand, Chromium QUIC CUBIC (1.34 Mbps), occupies less than half the bandwidth as TCP BBR (3.60 Mbps) (Table 4.2, QUIC CUBIC vs TCP BBR).



(a) QUIC BBR vs TCP BBR Throughput

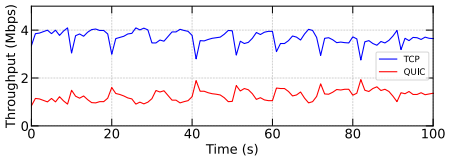


(b) QUIC BBR vs TCP BBR CWND

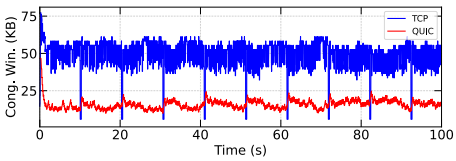


(c) 5-second zoom of CWND

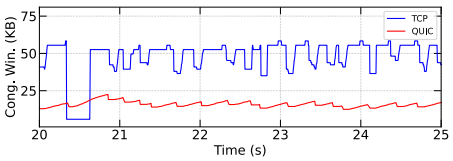
Figure 4.3: TCP BRR updates CWND more aggressively, capturing higher bandwidth than QUIC BBR. Timeline showing throughput and congestion window sizes of QUIC BBR and TCP BBR when transferring data over the same 5 Mbps bottleneck link (RTT = 36 ms, loss = 0 %, buffer = 30 KB).



(a) QUIC CUBIC vs TCP BBR Throughput



(b) QUIC CUBIC vs TCP BBR CWND



(c) 5-second zoom of CWND

Figure 4.4: BBR is unfair to CUBIC CCA, irrespective of its use in QUIC or TCP. Timeline showing throughput and congestion window sizes of QUIC CUBIC and TCP BBR when transferring data over the same 5 Mbps bottleneck link (RTT = 36 ms, loss = 0 %, buffer = 30 KB).

From these fairness experiments, we see that the fairness of a protocol is dependent on its CCA, namely CUBIC and BBR, whereas the behavior of these CCAs depends on their parameters. For example, as previously discussed in Section 4.1.1 connection emulation parameter (N) choice of CUBIC has an impact on how the CWND grows

Scenario	Flow	Avg. Throughput (std. dev.)	Fairness
QUIC vs TCP	QUIC BBR	1.74 (0.04)	TCP BBR is unfair
	TCP BBR	3.18 (0.04)	to QUIC BBR
QUIC vs TCP	QUIC CUBIC	1.34 (0.07)	TCP BBR is unfair
	TCP BBR	3.60 (0.06)	to QUIC CUBIC

Table 4.2: TCP BBR is unfair to both QUIC BBR and QUIC CUBIC. Average throughput (5 Mbps link, RTT = 36 ms, loss = 0 %, buffer = 30 KB, averaged over 5 runs) allocated to QUIC and TCP flows when competing with each other.

and shares the bandwidth with competing protocols. Similarly, BBR is unfair to CUBIC, irrespective of the protocol using it. Hence, in conclusion, we learn that, while fairness is the function of CCA, the behavior of these CCAs depends on the choices and accuracy of the parameters provided by the encapsulating protocols.

4.2 Page Load Time

Previously in Section 3.3, we discussed the use of Page Load Time (PLT) as a metric for evaluating the performance of QUIC. Now, we perform a comparative analysis of QUIC and TCP performance across various network conditions, with PLT as our primary metric. Alongside fairness, the ability of a transport protocol to perform optimally across diverse network conditions is crucial. Therefore, we assess the performance of QUIC and TCP under different emulated network settings (Table 3.3) by varying bandwidth (5 Mbps, 10 Mbps, 50 Mbps, and 100 Mbps), latency (36 ms, 112 ms) and loss (0%, 1%). Additionally, where applicable, we present the design difference between gQUICv37 and QUICv1, and their impact on performance.

As will be shown below, QUIC outperforms TCP in all conditions, except for the condition where there are a large number of small objects (Section 4.3.2) and when there is packet reordering in the network (Section 4.6). We note the evolution of designs between gQUICv37 and QUICv1 like the mechanism to adapt to packet re-

ordering using dynamic threshold, and show the performance impact of those changes.

In Figures 4.5 and 4.6, each square of a heatmap represents the percentage difference of TCP and QUIC PLTs averaged over 20 runs each (see Equation 4.1). The percentage difference is mapped to a heatmap, from +100% faster for QUIC (red) to -100% slower for QUIC (blue, i.e., faster for TCP).

$$PercentageDifference = \frac{\overline{TCP} - \overline{QUIC}}{\overline{QUIC}} * 100 \tag{4.1}$$

where:

- $\overline{TCP}$ = Average of 20 TCP PLTs
- $\overline{QUIC}$ = Average of 20 QUIC PLTs

If *PercentageDifference* is between 0 to +100 then QUIC is faster, indicated by a red shade in the heatmap. If *PercentageDifference* is between 0 to -100 then TCP is faster, indicated by a blue shade. For example, as seen in the left-top red square of heatmap Figure 4.6a, when the percentage difference is +72%, then QUIC is 72% faster than TCP. Similarly, in the right-bottom blue square of heatmap Figure 4.6d, when the percentage difference is -1.7%, then TCP is 1.7% faster than QUIC.

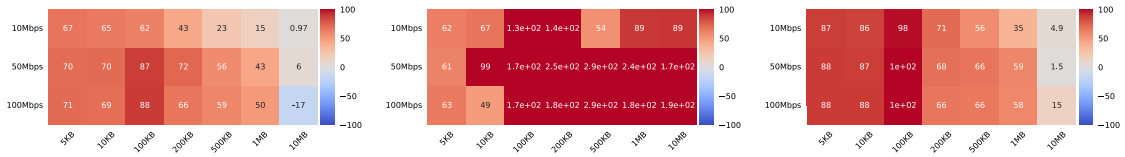
As done by Kakhki *et al.* [5], to make sure our results are not impacted by the noise in the experimental setup, we perform Welch's t-test [64]. Welch's t-test, or unequal variances t-test, is a two-sample location test that is used to test the (null) hypothesis that two populations have equal means [5]. We take PLTs of 20 runs of each TCP and QUIC and calculate the p-value according to Welch's t-test. If the p-value is smaller than our threshold (0.01), then we reject the null hypothesis that the mean performance for TCP and QUIC are identical, implying the difference we observe between the two protocols is statistically significant. If the p-value is greater than our threshold (0.01), the difference we observe is not significant and we indicate the corresponding square with white color.

Strictly speaking, Figure 4.5 replicates the results of Kakhki *et al.* [5], but Figure 4.6 extends the results by using QUICv1 (which was not available in 2017). Some important notes: First, the N = 2 CUBIC parameter in Figures 4.5 and 4.6 means that the QUIC implementation (by Google) would be unfair to TCP **if and only if** there are competing flows. However, other than for the fairness experiments (Section 4.1.1), there are no competing flows. The N parameter does have an impact on how packet loss is handled, which is discussed below (Section 4.4.1). Second, as discussed, we could force N = 1 by source code changes. Third, however, unless explicitly noted, N is set to 2 for all QUIC CUBIC experiments to maintain commonality with Kakhki *et al.* [5].

In the following sections from 4.3 to 4.6, we discuss QUIC's performance over a wide range of a parameter space. While Section 4.3 presents QUIC's performance in a baseline network setting of 36 ms RTT and 0% loss, Sections 4.4 and 4.5 talks about scenarios with added loss and latency, respectively. Further, in Section 4.6 we present QUIC under packet reordering before extending QUIC performance benchmarking for the newly available BBR CCA in Section 4.7.

4.3 QUIC in baseline setting: 36 ms RTT, 0% loss

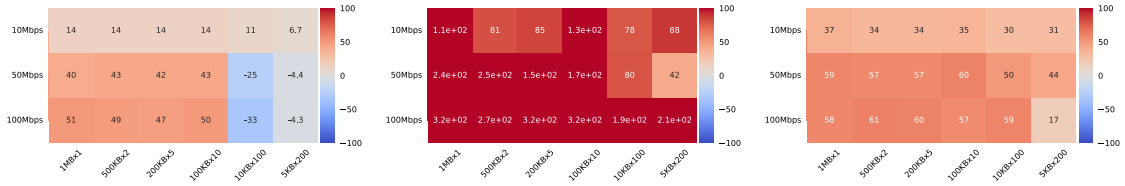
Due to the nature of our experimental setup in Emulab, where all the nodes are in the same data center, we measure (using ping) that the default latency between the server and client node connected via link bridge node (Figure 3.1) is in the range of 0.5 ms to 2 ms RTT. This kind of latency is lower than typical latency in a Wide Area Network (WAN). Hence, we add an implicit latency of 36 ms RTT to the link connecting server and client. The latency is introduced using the Dummynet pipes [44] in the link bridge node. We chose 36 ms following the latency reported by Kakhki *et al.* [5] for a shared WAN network, where they used a client machine connected to the server in an AWS EC2 instance over the Internet. Similarly, with the baseline parameter of 36 ms RTT and 0% loss, we choose bandwidths of 10 Mbps, 50 Mbps,



(a) Varying object size, 0% loss, 36 ms RTT

(b) Varying object size, 1% loss, 36 ms RTT

(c) Varying object size, 0% loss, 112 ms RTT

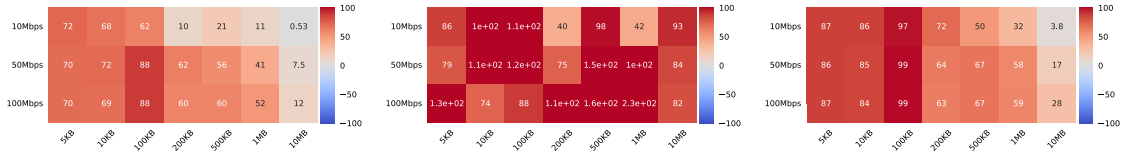


(d) Varying #object, 0% loss, 36 ms RTT

(e) Varying #object, 1% loss, 36 ms RTT

(f) Varying #object, 0% loss, 112 ms RTT

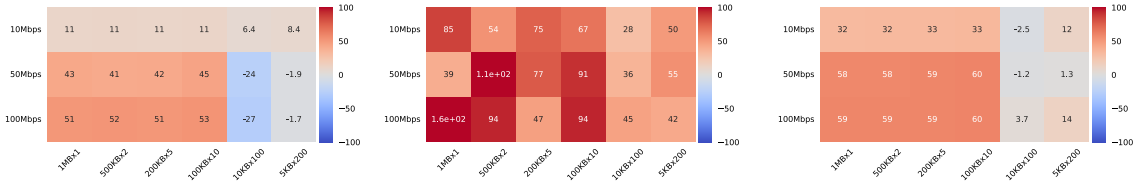
Figure 4.5: gQUICv37 0-RTT (CUBIC, N = 2) vs TCP (CUBIC). Chrome client. Red is better for gQUIC. Blue is better for TCP



(a) Varying object size, 0% loss, 36 ms RTT

(b) Varying object size, 1% loss, 36 ms RTT

(c) Varying object size, 0% loss, 112 ms RTT



(d) Varying #object, 0% loss, 36 ms RTT

(e) Varying #object, 1% loss, 36 ms RTT

(f) Varying #object, 0% loss, 112 ms RTT

Figure 4.6: QUICv1 0-RTT (CUBIC, N = 2) vs TCP (CUBIC), Chrome client. Red is better for QUIC RFC v1. Blue is better for TCP

and 100 Mbps for our experiments as used previously [5].

4.3.1 QUIC's performance for single object

The heatmap of Figure 4.6a shows the PLTs in the default setting of 36 ms latency and 0% loss for bandwidths of 10 Mbps, 50 Mbps, and 100 Mbps. Note that each square of the heatmap is annotated by the percentage difference of PLT between QUIC and TCP, as calculated from Equation 4.1. In this section we discuss the PLT heatmap, comparing the performance of QUIC and TCP for two versions of QUIC (QUICv1 and gQUICv37) using single objects of varying sizes (refer to Figure 4.6a and Figure 4.5a). In Section 4.3.2, we will extend our analysis to scenarios of the webpage with multiple objects of varying numbers. We first discuss the performance of QUICv1 in the baseline setting (36 ms RTT and 0% loss) as shown in Figure 4.6a, before discussing gQUICv37 results of Figure 4.5a.

As we can see from Figure 4.6a, QUIC outperforms TCP (i.e., red squares) throughout the range of object sizes from 5 KB to 10 MB (along the x-axis, each column represents different object size) over three different bandwidth of 10 Mbps, 50 Mbps, and 100 Mbps (along the y-axis, each row represents different bandwidths). For example, the performance difference ranges from 72% (top left square, 5 KB) to 0.53% (top right square, 10 MB) better for QUIC against TCP for bandwidth of 10 Mbps (Figure 4.6a top row).

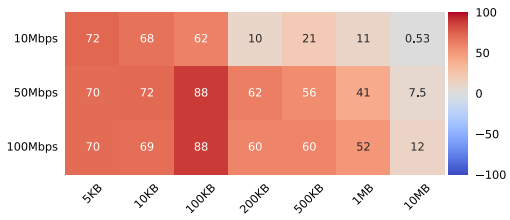
However, we can observe a pattern that the red shade indicating better performance for QUIC generally decreases as we move from object size of 5 KB to 10 MB of 10 Mbps bandwidth row in Figure 4.6a (with an exception of a 200 KB object). As noted earlier, the highest performance gain of 72% for the 5 KB object drops to 0.53% for the 10 MB object. A similar pattern can be observed for the bandwidth rows of 50 Mbps and 100 Mbps, except for the 100 KB object, where QUIC enjoys a 70% performance gain against TCP for an object sized 5 KB, before dropping to 7.5% for 50 Mbps bandwidth. The same is true for 100 Mbps bandwidth, with QUIC

performance dipping to 12% for the 10 MB object from 70% for the 5 KB object.

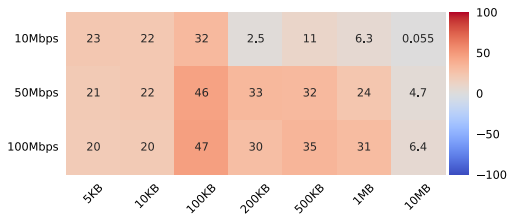
The primary reason for QUIC's superior performance over TCP in the baseline setting of 36 ms RTT and 0% loss, as depicted in Figure 4.6a, stems from its utilization of the 0-RTT connection time. As discussed in Section 2.1.1, QUIC connection establishment combines transport and cryptographic handshake reducing the connection time to 1-RTT for the initial connection and 0-RTT for subsequent connections (Figure 2.3). We will return to discussing the implications of 1-RTT connections later. In contrast, TCP necessitates 3-RTT for all connections (1.5 for TCP 3WHS and 1.5 for TLS handshake, Figure 2.2a). Consequently, with a latency of 36 ms RTT, QUIC's 0-RTT connection establishment gives it a significant advantage over TCP.

However, as the object size increases, the connection time contributes to a smaller portion of the total PLT, leading to the amortization of QUIC's 0-RTT advantage in the context of larger objects. This phenomenon occurs because, with larger objects like 10 MB, the time taken for the actual data transfer becomes a more substantial factor in the total PLT compared to the time required for connection establishment. Consequently, the initial advantage of QUIC's 0-RTT, utilized only at the beginning of the connection, starts to amortize. This trend explains the gradual decrease in QUIC's dominance as object size increases, particularly evident in the scenario with a 10 Mbps bandwidth, as illustrated in Figure 4.6a. Here, QUIC demonstrates only a 0.53% improvement over TCP for 10 MB objects, in comparison to the 72% improvement seen for the 5 KB object.

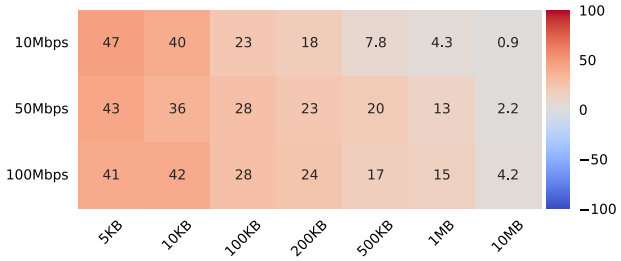
As with Kakhki *et al.* [5], to understand the impact of 0-RTT on the PLT performance, we directly compare the performance of QUIC with and without 0-RTT enabled (0-RTT vs 1-RTT, Figure 4.7c). We show that the 0-RTT advantage amortizes as the object size increases. The original work, as detailed in Fig. 7 of their paper [5], isolated the influence of 0-RTT by comparing the performance of QUIC with and without 0-RTT enabled. The results demonstrated a noticeable performance benefit for small objects, while the impact became statistically insignificant



(a) QUICv1 (0-RTT) vs TCP (3-RTT)



(b) QUICv1 (1-RTT) vs TCP (3-RTT)



(c) QUICv1 (0-RTT) vs QUICv1 (1-RTT).

Figure 4.7: Performance heatmaps showing the advantage given by QUIC's 0-RTT connection under 36 ms latency and 0% loss. The 0-RTT advantage is higher for smaller objects and diminishes as the object size increases. Subfigure (a) corresponds to Figure 4.6a showing QUIC with 0-RTT against TCP, whereas, in Subfigure (b) 0-RTT is disabled and shows QUIC with 1-RTT against TCP. The subfigure (c) shows the comparison between QUIC with 0-RTT and without 0-RTT (1-RTT).

for larger objects. In our study, we conducted a replication of a similar experiment. As explained in Section 3.2.3, we disable the ability of the HAR Capturer tool to reuse the existing browser context, therefore forcing the use of a 1-RTT connection in QUIC. By comparing the QUIC's 0-RTT PLT values with those obtained through 1-RTT connection, we aim to quantify the impact of 0-RTT connections on PLT performance.

In Figure 4.7c, we present a heatmap illustrating the PLT of QUIC with and

without 0-RTT (0-RTT vs 1-RTT) in a baseline scenario of 36 ms RTT and 0% loss. The red squares denote a positive performance difference, indicating the advantage for QUIC with 0-RTT. Our findings from Figure 4.7c, reveal that QUIC with 0-RTT exhibits a notable performance benefit, particularly for the smaller object of 5 KB. The performance difference ranges from 41% to 47%, favoring QUIC with 0-RTT. As we shift right towards the larger object of 10 MB on the heatmap (Figure 4.7c), the performance gain diminishes to between 0.9% to 4.2%, represented by grey squares. There is a noticeable drop in QUIC's 0-RTT performance against QUIC's 1-RTT for all bandwidths as we move from smaller objects (5 KB, left-most column) to larger objects (10 MB, right-most column). The amortization of QUIC's 0-RTT advantage (see Figure 4.7c) explains the diminishing performance of QUIC against TCP (see Figure 4.7a, same as Figure 4.6a) as the object size increases.

In comparing the PLT of the baseline scenario for a single object between gQUICv37 (Figure 4.5a) and QUICv1 (Figure 4.6a), we observe many similar patterns. Under the conditions of 36 ms RTT and 0% loss, both gQUICv37 and QUICv1 exhibit comparable trends. For instance, in Figure 4.5a, gQUICv37 consistently outperforms TCP across various scenarios, except for the 10 MB object at 100 Mbps bandwidth. Notably, gQUICv37 demonstrates a 67% improvement over TCP for a 5 KB object at 10 Mbps bandwidth (Figure 4.5a), mirroring QUICv1's 72% improvement for a similar object and bandwidth (Figure 4.6a). The pattern of similar results becomes even more apparent for 50 Mbps and 100 Mbps bandwidths, where the performance difference between gQUICv37 and QUICv1 remains within 10% for all object sizes (except for the 10 MB object at 100 Mbps bandwidth).

Furthermore, the observed pattern of decreasing QUIC dominance with larger object sizes is consistent in both gQUICv37 and QUICv1, with a notable exception for the 100 KB object in both versions. The performance gain of gQUICv37 over TCP decreases from 67% for a 5 KB object to 0.97% for a 10 MB object at 10 Mbps bandwidth (Figure 4.5a), resembling QUICv1's drop from 72% to 0.53% for the same

object sizes and bandwidth (Figure 4.6a). These findings in our baseline settings align with a prior study on gQUICv37 [5], suggesting that the performance implications of 0-RTT for different object sizes observed in gQUICv37 persist in QUICv1 as well.

The similarity in performance of gQUICv37 and QUICv1 is likely because, despite the use of different cryptographic handshake protocols in both versions of QUIC (Figure 2.3), the 0-RTT mechanism has remained unchanged. We know that gQUICv37 uses QUIC Crypto [17] and QUICv1 uses TLSv1.3 [9], and they both offer a similar 0-RTT functionality for QUIC. Since the 0-RTT has remained unchanged throughout QUIC's evolution, we see similar performance trends in both versions of QUIC.

4.3.2 QUIC's performance for multiple objects

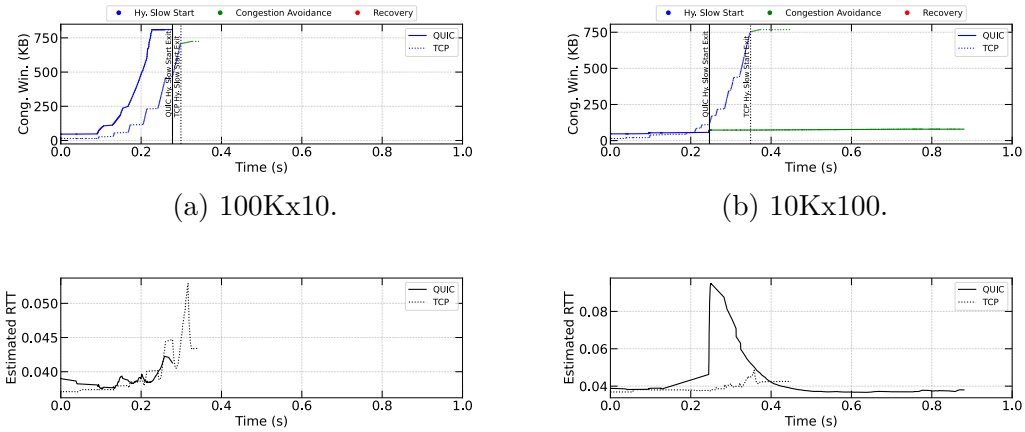
While measuring single-object performance is a good microbenchmark, a single page of a modern website often contains more than one object. Hence, in this section, we look at QUIC's performance for different numbers of objects ranging from 1 to 200 as shown along the x-axis of Figure 4.5d for gQUICv37 and Figure 4.6d for QUICv1. We find that QUIC performs poorly when there are a large number of small objects (e.g., 10 KB x 100, 5 KB x 200 in 50 and 100 Mbps bandwidth), but QUIC continues to perform better for a small number of large objects (e.g., 1 MB x 1, 500 KB x 2, 200 KB x 5 and 100 KB x 10).

Additionally, we validate Kakhki *et al.* [5]'s reasoning for QUIC's poor performance for a large number of small objects. We find that QUIC exits the Hybrid Slow Start [65] early due to an increase in estimated RTT while multiplexing a large number of objects causing it to perform poorly against TCP.

In Figure 4.5d and 4.6d, while QUIC has lower PLTs for the most squares of fewer number of objects like 1 MB x 1, 500 KB x 2, 200 KB x 5 and 100 KB x 10, as the number of objects increases QUIC starts to suffer. We can see this trend in Figure 4.5d for gQUICv37, where QUIC is consistently 40% to 51% faster than TCP for objects 1 MB x 1, 500 KB x 2, 200 KB x 5 and 100 KB x 10 of 50 Mbps and

100 Mbps bandwidth, but QUIC is 4.3% to 33% slower than TCP for 10 KB x 100 and 5 KB x 200 objects of the same bandwidth rows.

From Figure 4.5d and 4.6d, we also note that the performance issue of QUIC under a large number of small objects persist even in QUICv1. For example, consider the case of 100 objects of 10 KB (10 KB x 100) for bandwidths 50 Mbps and 100 Mbps in Figure 4.5d. We see gQUICv37 performs poorly, indicated by dark blue squares in the 10KBx100 column of Figure 4.5d, where gQUICv37 is 25% slower than TCP for 10 KB x 100 objects of 50 Mbps bandwidth, and 33% slower for 100 Mbps bandwidth. Similarly, looking at Figure 4.6d for QUICv1, we see QUICv1 is 24% slower than TCP for 10 KB x 100 objects of 50 Mbps bandwidth, and 27% slower for 100 Mbps bandwidth.



(a) 100Kx10.

(b) 10Kx100.

(c) 100Kx10.

(d) 10Kx100.

Figure 4.8: Hybrid Slow Start: QUICv1 (CUBIC, N = 1) vs TCP (CUBIC). CWND and estimated RTT while transferring different numbers of objects. When the number of objects increases (10Kx100) there is a sudden spike in estimated RTT for QUIC, which leads to early exit from Hybrid Slow Start phase.

We take the lead from Kakhki *et al.* [5]'s reasoning that QUIC's poor performance under a large number of small objects is due to the early exit of the Hybrid Slow Start phase. As shown in Figure 4.8, we look at CWND to verify if the same issue is present in QUICv1. We plot the CWND of TCP and QUIC with CUBIC as CCA over time for 100 KB x 10 and 10 KB x 100 objects by clearly indicating the phases

of CWND growth, namely, Hybrid Slow Start [65] ("Hy. Slow Start"), Congestion Avoidance, and Recovery. As discussed in Section 4.1.1, we know CUBIC CCA in Chromium QUIC emulates 2 TCP connections (N = 2). Hence, we make sure to use QUIC CUBIC with Emulated connection parameter N = 1, to make a fair comparison with TCP CUBIC.

The Hybrid Slow Start [65] algorithm is an enhancement of the TCP Slow Start algorithm [66], where the CWND grows exponentially until it either detects a loss or high latency. After loss or high latency is detected, the CCA falls into the congestion avoidance phase, in which CWND growth is slower. We identify the congestion stages for QUIC by instrumenting the Chromium server code, whereas for TCP we use data from tcpprobe [60]. The CWND line (Cong. Win. (KB)) along the y-axis in the time series Figure 4.8a is color coded with blue to show the "Hy. Slow Start" phase if `cwnd` (CWND in QUIC server code) is less than `ssthresh` (`cwnd` $<$ `ssthresh`), else it is marked as congestion avoidance phase indicated by green color (`cwnd` $\geq$ `ssthresh`). Chromium server code has a function to indicate the recovery phase (`largest_acked_packet_number` $\leq$ `largest_sent_at_last_cutback`) which is color coded as red, but we were not able to come up with heuristics to identify the recovery phase for TCP using tcpprobe data.

From Figure 4.8a, we see that while QUIC and TCP show the Hybrid Slow Start phase up to a close time duration, where QUIC exits the Hybrid Slow Start at 0.27 seconds (x-axis) and TCP exits Hybrid Slow Start at 0.29 seconds. As soon as the number of objects increases to 100 objects (10K x 100) in Figure 4.8b, QUIC exits Hybrid Slow Start far earlier than TCP as indicated by solid blue line of QUIC CWND ("Cong. Win. (KB)" along the y-axis), which ends around 0.24 seconds as compared to TCP Hybrid Slow Start exit which occurs at 0.34 seconds. More importantly, as an effect of QUIC's early exit from Hybrid Slow Start, QUIC CWND stays in the congestion avoidance phase for a longer duration of the total transfer time. As a result, we observe that the CWND growth of TCP represented by the dotted line is exponential

for a longer period than QUIC's CWND indicated by a solid line. The early exit from Slow Start results in QUIC utilizing a smaller portion of links bottleneck bandwidth (Figure 4.8b, solid line), resulting in higher PLT for QUIC.

To further validate the reason for early exit from the Hybrid Slow Start, we look at the estimated latency of QUIC and TCP. As we know high latency along with loss is one of the reasons for CCA to exit the Hybrid Slow Start phase, and the high latency was given as the reason for the early exit of QUIC by Kakhki *et al.* [5]. From Figure 4.8d we see that both QUIC and TCP's estimated RTT is initially (up to 0.24 seconds) close to the actual latency of the link which is 36 ms. However, we see from the same Figure 4.8c the immediate spike of estimated latency for QUIC right after 0.24-second mark in the transfer duration of 10 KB x 100 which causes QUIC to exit the Hybrid Slow Start prematurely and enters congestion avoidance phase indicated by green solid line in the corresponding CWND of Figure 4.8b. Further investigation with other implementations is required to understand the spike in estimated RTT when multiplexing multiple objects in QUIC.

4.4 QUIC with added loss: 36 ms RTT, 1% loss

In this section, we look at the performance of QUIC under lossy conditions of 1% packet loss. Emulab [36] provides link bridge nodes, which are equipped with Dummynet [44]. We use the Dummynet inside link bridge node to introduce packet loss, these link bridge node sits in between server and client nodes (Figure 3.1).

QUIC's superior performance under loss is evident from the Figures 4.5b and 4.5e for gQUICv37 and Figures 4.6b and 4.6e for QUICv1. Every square of Figure 4.5b is red, and performance improvement for gQUICv37 over TCP ranges from lowest of 48% for 10 KB of 100 Mbps to highest of close to 300% for 500 KB of the same bandwidth row. Similarly, QUICv1 of Figure 4.6b shows a similar trend of every square being red, and performance improvement for QUICv1 over TCP ranges from the lowest of 40% for 200 KB of 10 Mbps to highest of close to 230% (i.e., 2.3e+02)

for 1 MB object of the 100 Mbps bandwidth row. Furthermore, both QUIC versions show similar trends of QUIC outperforming TCP throughout the range of objects and bandwidth for multiple objects scenario, as seen from Figure 4.5e and 4.6e.

As discussed in Section 2.1.2, one of the advantages of HTTP/3 over HTTP/2 is the ability to avoid head-of-line (HOL) blocking [11], where HTTP/3 benefits from the underlying QUIC connection to support independent streams. Unlike HTTP/2 where every stream is dependent on a single TCP connection [18], HTTP/3 streams are independent of each other. Multiple stream support within a single QUIC connection ensures that a packet loss in a single stream does not cause the blocking of packets from another stream. This phenomenon helps QUIC to have better performance under lossy conditions.

4.4.1 Effect of N=2 on PLT under loss

In Section 4.1.1's discussion of fairness, we learned that QUIC CUBIC uses an Emulated connection (N) of 2. QUIC's use of N = 2 in CUBIC CCA, resulted in QUIC being unfair to competing TCP connections while using the same link. In this section, we see how the CUBIC factor N impacts QUIC performance under lossy conditions.

From the heatmaps of Figure 4.9, we see that while using QUIC CUBIC with N = 1 (Figure 4.9b and 4.9d), the performance of QUIC under loss condition diminishes for large objects and a large number of small objects. For example, in Figure 4.9b while moving from 5 KB to 10 MB objects columns, we see that QUIC performance decreases, going from 180% for 5 KB to -10% for 10 MB object of 10 Mbps bandwidth row. However, when QUIC CUBIC is using N = 2 in Figure 4.9a, the performance of QUIC under loss conditions is better than TCP for all object sizes and bandwidths.

Similarly, in case of multiple objects of Figure 4.9c and 4.9d, we observe a similar trend where QUIC CUBIC with N = 2 outperforming TCP for all squares in Figure 4.9c to QUIC CUBIC with N = 1 losing out to TCP when there are a large number of small objects (e.g., 100 KB x 10, 10 KB x 100, 5 KB x 200). This shows that CUBIC

gained an advantage against TCP during our previous PLT experiment (Figure 4.6), due to the use of a CUBIC factor called Emulated connections (N = 2).

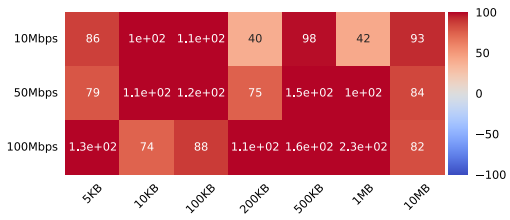
Kakhki *et al.* [5] in Section 5.2, attributed the performance improvement of QUIC under loss to QUIC CCA's ability to cope with loss. This was backed by plotting CWND of QUIC and TCP under loss (c.f., Figure 9 [5]). We perform the same experiment, as shown in Figure 4.10. We observe that the ability of QUIC to increase its CWND higher than TCP was simply due to the use of N = 2. By comparing the Figures 4.10a and 4.10c, we see that the QUIC CUBIC CWND for N = 1 of Figure 4.10c is same as TCP CWND of Figure 4.10a under loss condition. As we can see the CWND in both Figure 4.10a and 4.10c fluctuate between a low of around 5 KB and a high of just below 40 KB. Similarly, by looking at Figure 4.10b where QUIC CUBIC is using N = 2, the CWND range is higher, varying between a low of 20 KB to a high of around 50 KB. Hence, QUIC's higher performance for large objects (10 MB) and multiple objects under loss was an artifact of QUIC CUBIC using N = 2, and not due to its ability to cope with loss.

In summary, QUIC's ability to avoid HOL blocking [11] results in better performance than TCP under lossy conditions. However, for larger objects (10 MB) the use of CUBIC factor N = 2 in QUIC, results in QUIC being better than TCP due to its aggressive CWND growth (Figure 4.10b). For scenarios with no loss of Figure 4.6a, 4.6d, 4.6c, 4.6f, the value of N did not have a huge impact on QUIC performance.

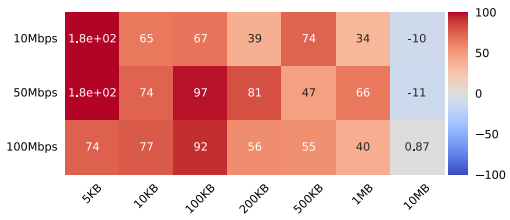
4.5 QUIC with added latency: 112 ms RTT, 0% loss

In this section, we look at QUIC performance under added latency. Similar to the added latency scenario of Kakhki *et al.* [5], we add 76 ms of latency to the baseline latency of 36 ms to induce a latency of 112 ms through Dummynet pipes.

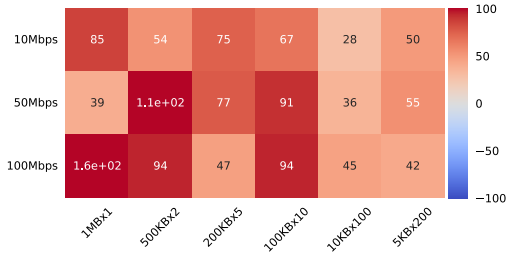
Figure 4.6c and 4.6f represents QUIC performance under the latency 112 ms RTT. We can observe that the trends from the baseline setting (36 ms RTT, 0% loss) of



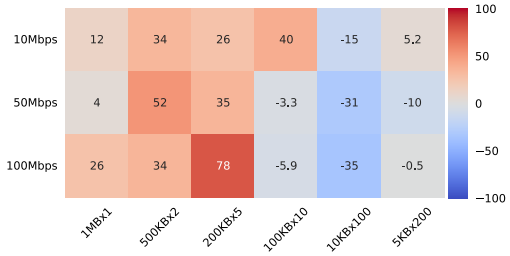
(a) Varying object size, 1% loss, 36 ms RTT. QUIC CUBIC N = 2 (Default)



(b) Varying object size, 1% loss, 36 ms RTT. QUIC CUBIC N = 1



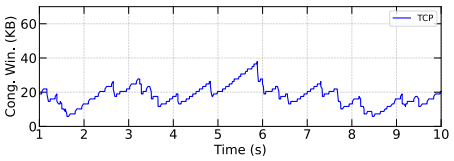
(c) Varying #object, 1% loss, 36 ms RTT. QUIC CUBIC N = 2 (Default)



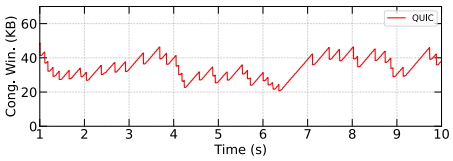
(d) Varying #object, 1% loss, 36 ms RTT. QUIC CUBIC N = 1

Figure 4.9: QUIC (CUBIC, N=2) vs TCP and QUIC (CUBIC, N=1) vs TCP at 36 ms RTT and 1% loss. Subfigure (a) and (b) corresponds to Figure 4.6b and 4.6e respectively, with CUBIC CCA emulating 2 connections (N = 2) in CWND update, which is the default in Chromium QUIC implementation.

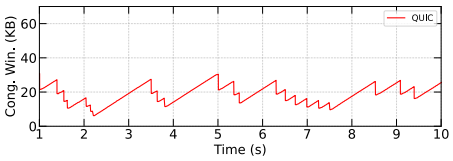
Figure 4.6c and 4.6f is similar to that of the added latency setting (112 ms RTT, 0% loss) of Figure 4.6a and 4.6d except for the fact that QUIC's dominance is more evident with darker red color throughout the range of objects. For example, in baseline Figure 4.6a we see QUIC outperforming TCP by 72% for 5 KB object of 10 Mbps bandwidth, which has increased by 15% to QUIC outperforming TCP by 87% for the same object size and bandwidth in added latency setting of Figure 4.6c. The same is true for all other object sizes and bandwidths, where the performance difference



(a) TCP CUBIC cwnd under loss



(b) QUIC CUBIC (N=2) cwnd under loss



(c) QUIC CUBIC (N=1) cwnd under loss

Figure 4.10: Congestion window over time for QUIC and TCP at 100 Mbps rate limit, 36 ms latency and 1% loss.

between QUIC and TCP increases in the added latency setting as compared to the baseline setting.

For the multiple object scenario of Figure 4.6f, we see that QUIC continues to improve as compared to the corresponding baseline Figure 4.6d. The added latency also resulted in QUIC gaining an advantage over TCP for a large number of small objects (e.g., 10 KB x 100, 5 KB x 200). For example, QUIC's negative performance difference for 10 KB x 100 in 50 Mbps, shown in Figure 4.6d, dropped from -24% to -1.2% with the added latency, as shown in Figure 4.6f.

The 0-RTT connection establishment time is the reason behind QUIC's improved performance under higher latency (112 ms) (see Figure 4.6f). QUIC's performance with respect to TCP increases as the latency increases because as the latency increases the connection time of TCP increases linearly. However, QUIC connection establishment time is largely insensitive to RTT due to the fixed latency cost of 0-RTT as noted in Langley *et al.* [7].

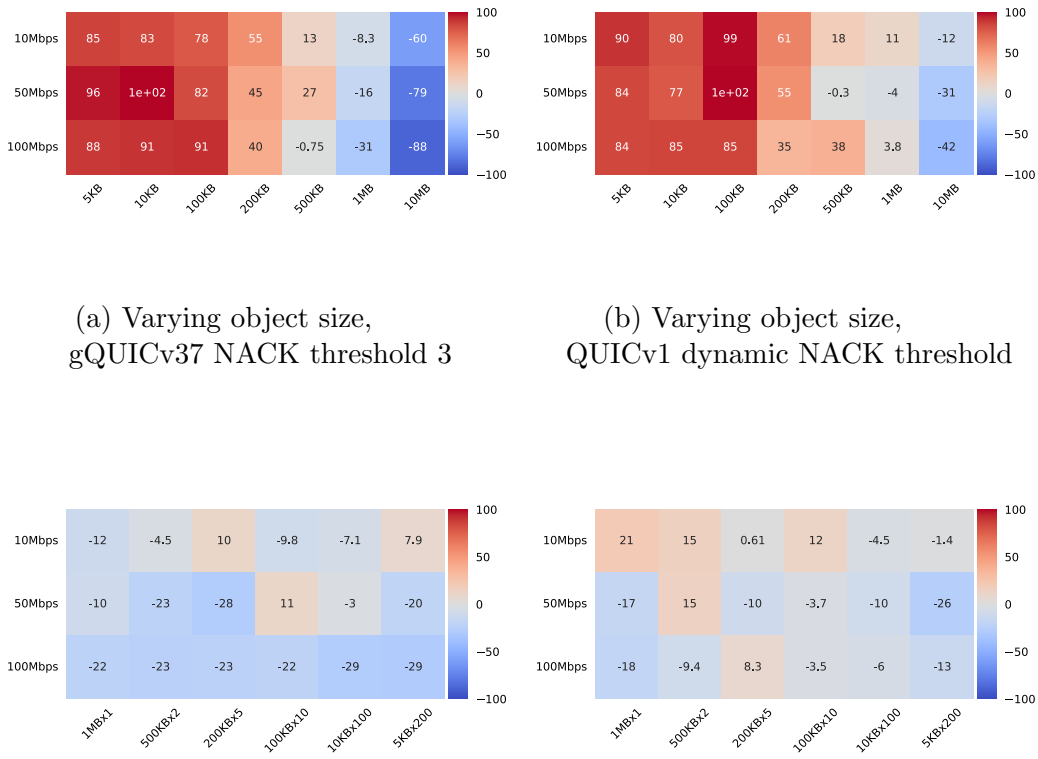
4.6 QUIC with packet reordering due to jitter

Networks exhibiting jitter (per packet variance in latency) might have multiple effects on packets, one of which is out-of-order packet arrival. Servers that rely on ACKs to detect loss will prematurely declare a packet is lost when it receives ACK for newer packets while the packet that was sent earlier is still in flight and not actually lost. Referring to the networks that possess packet reordering, RFC 9002 [10] (Section 6.1.1) notes that "packet reordering could be more common with QUIC than TCP because network elements that could observe and reorder TCP packets cannot do that for QUIC and also because QUIC packet numbers are encrypted." Hence, it is important to study the performance of QUIC under jitter conditions.

Packet reordering can be caused by varying latency (jitter) using TC (Section 3.3). In our setup, we induce a jitter of 50 ms to cause packet reordering, which is described in Section 3.3. While the same magnitude of packet reordering can be caused by a smaller jitter in a live network, due to the nature of our setup (directly connected server and client) we observed that to cause a measurable packet reordering we need to induce a jitter of 50 ms.

Figure 4.11a, shows the QUIC performance under jitter, which shows a trend of QUIC performing better for small objects of size from 5 KB to 500 KB. However, QUIC starts to suffer for large objects of size 1 MB and 10 MB of Figure 4.11a. Similarly, QUIC has poor performance compared to TCP for a whole range of multiple objects (from 1 MB x 1 to 5 KB x 200) of Figure 4.11c.

Now, we compare the jitter results of gQUICv37 from Figures 4.11a and 4.11c to QUICv1 results from Figures 4.11b and 4.11d. We notice that the blue square gets lighter in both single-object and multiple-object scenarios. For example, in Figure 4.11b for QUICv1, the percentage difference for 10 MB object of 10 Mbps bandwidth is higher than the corresponding square of gQUICv37 in Figure 4.11a, which increase from -60% to -12%, indicating an improvement to QUIC over TCP. Similarly, in Fig-



(a) Varying object size, gQUICv37 NACK threshold 3

(b) Varying object size, QUICv1 dynamic NACK threshold

(c) Varying #object, gQUICv37 NACK threshold 3

(d) Varying #object, QUICv1 dynamic NACK threshold

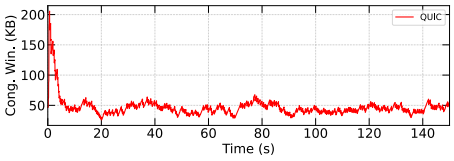
Figure 4.11: gQUICv37 vs TCP and QUICv1 vs TCP at 112 ms RTT with 50 ms jitter that causes packet reordering (Delay = 112 ms, Jitter = 50 ms, Loss = 0%). gQUICv37 uses a static NACK threshold of 3, while QUICv1 uses a dynamic NACK threshold. Performance improvement of Dynamic NACK is visible through blue squares getting lighter for QUIC v1 PLT results (Subfigure (b) and (d)).

ure 4.11d for QUICv1, the percentage difference for all objects of 100 Mbps bandwidth is higher than the corresponding row of gQUICv37 in Figure 4.11c. The above trend implies that QUICv1 is able to offer slightly better performance than gQUICv37 when there is packet reordering.

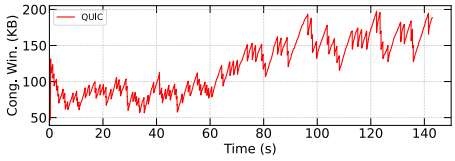
To explain this we look at the loss detection algorithm under jitter for both the gQUICv37 and QUICv1 protocol. The NACK threshold[1] (A.k.a. packet reordering

[1]We use the term NACK threshold instead of packet reordering threshold to be consistent with previous work [5]. RFCs use the term packet reordering threshold (kPacketThreshold).

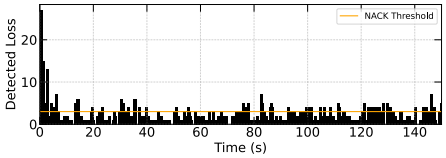
threshold) is the number of packet reorderings allowed before a packet is declared lost. While gQUICv37 uses a static NACK threshold of 3, QUICv1 uses a dynamic NACK threshold. Hence, in gQUICv37 if there is a packet reordering of more than 3 packets, the packet is declared lost, whereas in QUICv1 the NACK threshold increases with the increase in packet reordering.



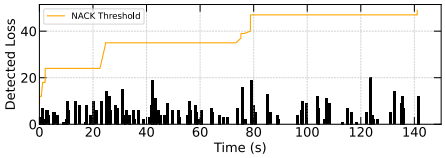
(a) gQUICv37 CUBIC CWND



(b) QUICv1 CUBIC CWND



(c) gQUICv37 detected loss with fixed NACK threshold



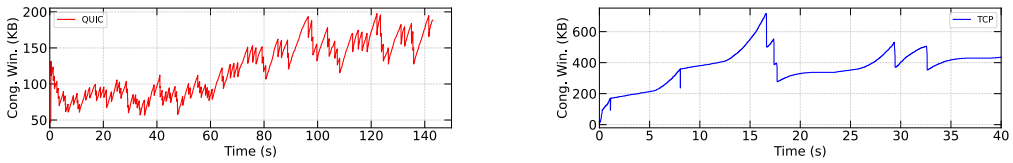
(d) QUICv1 detected loss with dynamic NACK threshold

Figure 4.12: QUIC's loss detection mechanisms often trigger false positives due to jitter-induced out-of-order packet delivery, hindering CWND growth. gQUICv37's use of a static threshold for packet reordering exacerbates this issue, leading to increased false positives. However, QUICv1 employs a dynamic threshold, effectively reducing false positives and facilitating CWND expansion. The timeline figures illustrate the relationship between CWND ("Cong. Win. (KB)") and detected losses during the transfer of a 210 MB object. (BW = 100 Mbps, Delay = 112 ms, Jitter = 50 ms, Loss = 0%).

As shown in the timeline Figure 4.12c, which plots detected loss (y-axis) against time (x-axis), the fixed NACK threshold (3) of gQUICv37 causes continuous premature loss detection due to packet reordering. This is indicated by the dense black bars occurring almost continuously throughout the test duration from 0 seconds to 150 seconds. The continuously detected loss constantly limits the CWND growth, as CUBIC updates CWND based on reported loss. As seen in Figure 4.12a, the CWND stays below 50 KB for the most part of 150 seconds due to the constant loss detection. As a result QUIC CUBIC under static NACK is unable to utilize the full capacity

of the link, resulting in higher PLT, represented by blue squares in Figure 4.11a and 4.11c.

However, from Figure 4.12d, we can observe that as the dynamic NACK threshold of QUICv1 increases (yellow line), the occurrence of premature loss decreases (black bars). This is indicated by the sparse appearance of black bars after the 80-second timestamp, coinciding with the increase in the NACK threshold above y = 40, as shown by the yellow line. As a result of this reduction in loss over time due to the higher NACK threshold, there is a gradual growth in the CWND, as depicted by the red line in Figure 4.12b. Initially, the CWND remains below 100 KB until the 60-second mark, after which it steadily increases, reaching 200 KB by the 120-second timestamp. This growth in CWND correlates with the decrease in loss observed around the same time duration, as illustrated in Figure 4.12d. As a result, QUICv1 is able to utilize the full capacity of the link, resulting in lower PLT, represented by the dimming of blue squares in Figure 4.11b and 4.11d.



(a) QUICv1 CUBIC (N = 2) CWND (b) TCP CUBIC CWND

Figure 4.13: Packet reordering affects QUIC's CWND growth more than TCP's CWND growth.

When there is packet reordering, even though QUICv1's dynamic NACK threshold improved performance over gQUICv37's static NACK threshold (from Figure 4.11c to Figure 4.11d), QUICv1 is still not on par with TCP's performance under jitter. We again look at CWND of both QUICv1 and TCP protocols, to see possible reasons. As shown in the CWND timeline Figure 4.13, the CWND growth of TCP is higher than QUICv1 for the same CCA under jitter. This suggests that the loss detection mechanism of QUICv1 is not able to cope with packet reordering as well as TCP's loss detection mechanism.

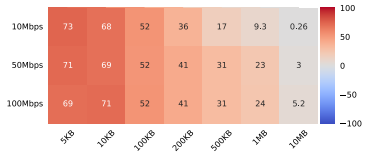
We look at the loss detection mechanism used in QUIC and TCP. QUIC's "Loss Detection and Congestion Control" RFC 9002 Section 6.1.1 [10], recommends using TCP's RACK-TLP [67] for updating the reordering threshold. However, we were not able to verify the same in Chromium QUIC through source code analysis. We also looked at other studies [22][29] which measure the conformation of Chromium QUIC implementation with TCP(kernel). While Mishra and Leong [29] shows that Chromium QUIC's CUBIC is not conforming to TCP CUBIC, it does not go into details of any deviation in implementation in terms of loss detection (except CUBIC Emulated connection, N). Therefore, we conclude that either a detailed analysis or a more sophisticated tool is required to identify if Chromium QUIC and other implementations are following the RFC standard for loss detection under packet reordering.

4.7 QUIC with BBR

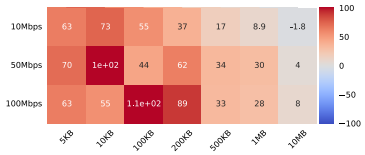
In this section, we demonstrate that QUIC with BBR outperforms TCP with BBR throughout the parameter sweep of our experiments (Figure 4.14). For example, many squares of multiple object in Figures 4.14d, 4.14e and 4.14f are red, with positive performance difference of QUIC over TCP ranging from 43% to 0.74%. We further show that QUIC is even more effective when used with BBR CCA as compared to CUBIC CCA under packet reordering scenario (Figure 4.15).

QUIC being in user space can evolve faster than other protocols. Additionally, QUIC is highly modular and can switch CCAs irrespective of its availability in the operating system kernel. While we cannot use BBR [61] with gQUICv37, we run QUICv1 with BBR (BBRv1) and configure the Linux kernel to use TCP with BBR (BBRv1).

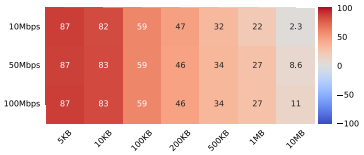
We perform the PLT experiment with both QUIC and TCP using BBR. When QUIC BBR results (Figure 4.14) are compared to QUIC CUBIC results (Figure 4.6), we find a similar trend under are three conditions (baseline Figure 4.6a, added loss Figure 4.6b and latency Figure 4.6c). In baseline Figure 4.14a, QUIC BBR outper-



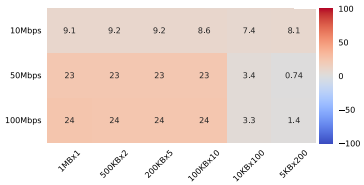
(a) Varying object size, 0% loss, 36 ms RTT.



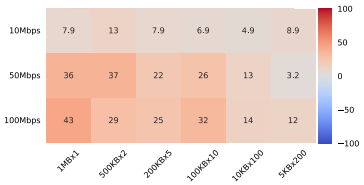
(b) Varying object size, 1% loss, 36 ms RTT.



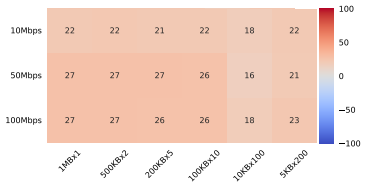
(c) Varying object size, 0% loss, 112 ms RTT.



(d) Varying #object, 0% loss, 36 ms RTT.



(e) Varying #object, 1% loss, 36 ms RTT.

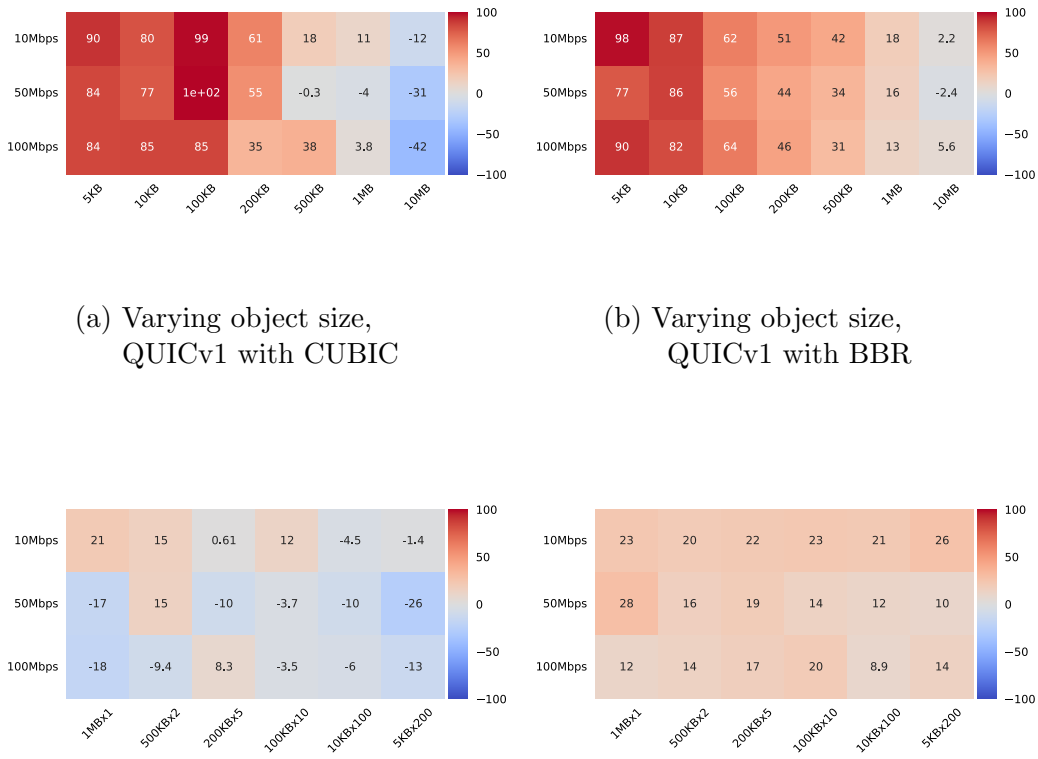


(f) Varying #object, 0% loss, 112 ms RTT.

Figure 4.14: QUICv1 (BBR) vs TCP (BBR). Red is better for QUICv1. Blue is better for TCP

forms TCP BBR for all object sizes and bandwidths, with the performance difference in 10 Mbps bandwidth decreasing from 73% to 0.26% as the object size increases from 5 KB to 10 MB. The decreasing dominance of QUIC BBR over TCP BBR is also observed in the case of added latency, as shown in Figure 4.14c. For example in the 100 Mbps bandwidth row, QUIC BBR is 87% better than TCP BBR for a 5 KB object, which drops to 11% for a 10 MB object.

However, we could observe that the QUIC BBR figures under 1% packet loss (Figure 4.14b) are more similar to QUIC CUBIC with N = 1 (Figure 4.9b) than N = 2 (Figure 4.9a). The performance of QUIC BBR is similar to QUIC CUBIC N=1 because Chromium QUIC BBR does not have a parameter similar to N (Emulated connection) in CUBIC. Further, in the case of multiple objects, QUIC BBR outperforms TCP BBR for all squares of Figure 4.14d and 4.14f, which is in contrast to QUIC CUBIC of Figure 4.6d and 4.6f.



(a) Varying object size, QUICv1 with CUBIC

(b) Varying object size, QUICv1 with BBR

(c) Varying #object QUICv1 with CUBIC

(d) Varying #object QUICv1 with BBR

Figure 4.15: QUICv1 (CUBIC) vs TCP (CUBIC) and QUICv1 (BBR) vs TCP (BBR) at 0% loss, 112 ms RTT with 50 ms jitter that causes packet reordering. As BBR does not rely on loss detection to regulate CWND, QUIC with BBR does not suffer performance degradation due to falsely detected loss due to packet reordering.

Furthermore, we look at the packet reordering scenario, where we find significant improvement in QUICv1 BBR performance as compared to QUICv1 with CUBIC under a jitter of 50 ms (compare Figure 4.15b with Figure 4.15a). For easier comparison, we present the jitter results of QUICv1 with CUBIC and BBR side by side in Figure 4.15. Where Figure 4.15b and 4.15d represents BBR results (QUIC vs TCP), Figure 4.15a and 4.15c represents CUBIC results (QUIC vs TCP). By comparing CUBIC (left column of Figure 4.15) results with BBR (right column of Figure 4.15), we notice the disappearance of blue squares in BBR results. The absence of blue

squares indicates that QUICv1 with BBR outperforms TCP with BBR for almost all object sizes and number of objects (except 10 MB of 50 Mbps).

The reason for QUICv1 BBR's better performance under jitter compared to QUICv1 CUBIC is that BBR does not rely on loss detection to regulate CWND. As seen previously in Section 4.6 and Figure 4.12, packet reordering led to falsely detected loss in both gQUICv37 and QUICv1 CUBIC, reducing CWND and increasing PLT (lower PLT). In contrast, BBR's loss-independent CWND regulation prevents performance degradation in QUIC with BBR under packet reordering.

4.8 Concluding remarks

In this chapter, we compared gQUICv37 and QUICv1 with TCP under different network conditions. We see that QUIC's performance is better than TCP under latency and lossy conditions due to its 0-RTT connection and ability to avoid HOL blocking. Our experiments show that the performance advantages of QUIC over TCP are consistent in both gQUICv37 and QUICv1, validating the results of the previous study [5]. However, we find different results than the previous study [5] on QUIC unfairness to TCP, and also observe scenarios where QUICv1 has improved over gQUICv37. The notable differences and new insights from our experiments are listed below:

1. **Unfairness**: We show the QUIC unfairness reported in previous study [5] was an artifact of the CUBIC parameter (Emulated connection, N=2) used in Chromium QUIC implementation and not because of QUIC protocol itself (Figure 4.1). When QUIC CUBIC Emulated connection is set to one (N = 1), it is fair to TCP CUBIC in both gQUICv37 and QUICv1. For example, from Figure 4.1a, QUIC CUBIC (red) with N = 2 occupies 4 Mbps bandwidth, which is twice as much as TCP CUBIC (blue). But, when QUIC CUBIC is emulated with N = 1 (Figure 4.1b), it occupies half of the total 5 Mbps bandwidth, which

is the same as TCP CUBIC (approximately 2.5 Mbps).

2. **Updated loss detection**: Our empirical result shows that the performance of QUICv1 has improved over gQUICv37 under jitter, due to an updated loss detection strategy (dynamic vs static threshold) under packet reordering (Figure 4.11). For instance, from 10 MB object square of 10 Mbps row in Figure 4.11b, QUICv1 with dynamic threshold is only 12% slower than TCP, which is an improvement from 60% slower for gQUICv37 with static (3) threshold (Figure 4.11a).

3. **BBR results**: We benchmark the newly available BBR CCA and show that QUIC BBR outperforms TCP BBR for PLT results, even in a jitter scenario (Figure 4.15). Most squares of QUIC BBR in Figure 4.14 are red indicating QUIC BBR is better than TCP BBR in the measured bandwidth, latency, and loss conditions. The same is true for BBR jitter scenario of Figure 4.15b and Figure 4.15d, where QUIC BBR outperforms TCP BBR for all object sizes and number of objects (except 10 MB of 50 Mbps), with the performance improvement of QUIC BBR over TCP BBR ranging from 98% to 8.9% (Figure 4.14).

While our experiments demonstrate that QUIC outperforms TCP for web workloads with a maximum object size of 10 MB under bandwidths up to 100 Mbps, other studies have shown that QUIC is not always superior, particularly for large object size under higher bandwidths [68]. This is because QUIC, being implemented in user space, does not benefit from NIC offloading and hardware acceleration like TCP [68]. Therefore, one must consider the specific workload and network conditions before choosing to use QUIC over TCP.

Chapter 5

Concluding Remarks

In this thesis, our primary aim was to understand the performance of a recent version of QUIC (i.e., QUICv1), using a contemporary experimental platform like Emulab, and provide an easily reproducible experimental framework for enabling the evaluation of future versions of QUIC. To achieve this goal, we adopt methodology from the previous work of Kakhki *et al.* [5] and benchmark gQUICv37 against TCP and then extended those experiments for QUICv1. Along the way, we encountered interesting findings that shed light on QUIC's performance and behavior.

During our first phase of benchmarking gQUICv37, we discovered a different result from previous a study [5] regarding QUIC fairness, which we found to be an artifact of the CUBIC parameter N (Emulated connection) being 2 in Chromium's QUIC implementation. The unfairness was an issue with the Chromium QUIC stack rather than an inherent issue with the QUIC protocol itself. Furthermore, in our second phase of experiments of benchmarking QUICv1, apart from establishing the consistent performance of two QUIC versions, we observed notable performance improvements in QUICv1 compared to gQUICv37, particularly in scenarios involving jitter, due to an updated loss detection strategy and the use of new BBR CCA.

Our use of Emulab addressed our second goal of reproducibility, providing a controlled and predictable environment for conducting experiments. Moving forward, our experimental framework in Emulab can serve as a valuable tool for evaluating

future versions of QUIC. With the recent release of QUIC extensions such as QUIC for proxy and QUIC for media, our framework can be extended to evaluate these features and contribute to a deeper understanding of the evolving QUIC protocol.

Bibliography

[1] *QUIC at 10,000 feet*, Nov. 2014. [Online]. Available: https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU/edit.

[2] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-transport-00, Nov. 2016, Work in Progress, 45 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-transport/00/.

[3] J. Iyengar and M. Thomson, *Rfc 9000: Quic: A udp-based multiplexed and secure transport*, USA, 2021.

[4] M. Duke, *QUIC Version 2*, RFC 9369, May 2023. DOI: 10.17487/RFC9369. [Online]. Available: https://www.rfc-editor.org/info/rfc9369.

[5] A. M. Kakhki, S. Jero, D. Choffnes, C. Nita-Rotaru, and A. Mislove, "Taking a long look at quic: An approach for rigorous evaluation of rapidly evolving transport protocols," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17, London, United Kingdom: Association for Computing Machinery, 2017, 290–303, ISBN: 9781450351188. DOI: 10.1145/3131365.3131368. [Online]. Available: https://doi.org/10.1145/3131365.3131368.

[6] J. Roskind, *QUIC: IETF-88 TSV Area Presentation*, 2013. [Online]. Available: https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf.

[7] A. Langley *et al.*, "The quic transport protocol: Design and internet-scale deployment," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, 183–196, ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. [Online]. Available: https://doi.org/10.1145/3098822.3098842.

[8] J. Dellaverson, T. Li, Y. Wang, J. Iyengar, A. Afanasyev, and L. Zhang, *A Quick Look at QUIC*, 2021. [Online]. Available: https://web.cs.ucla.edu/~lixia/papers/UnderstandQUIC.pdf.

[9] M. Thomson and S. Turner, *Using TLS to Secure QUIC*, RFC 9001, May 2021. DOI: 10.17487/RFC9001. [Online]. Available: https://www.rfc-editor.org/info/rfc9001.

[10] J. Iyengar and I. Swett, *QUIC Loss Detection and Congestion Control*, RFC 9002, May 2021. DOI: 10.17487/RFC9002. [Online]. Available: https://www.rfc-editor.org/info/rfc9002.

[11] M. Bishop, *HTTP/3*, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.

[12] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, "It's over 9000: Analyzing early quic deployments with the standardization on the horizon," in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21, Virtual Event: Association for Computing Machinery, 2021, 261–275, ISBN: 9781450391290. DOI: 10.1145/3487552.3487826. [Online]. Available: https://doi.org/10.1145/3487552.3487826.

[13] D. Belson and L. Pardue, *Examining HTTP/3 usage one year on*, 2023. [Online]. Available: https://blog.cloudflare.com/http3-usage-one-year-on.

[14] Wikipedia contributors, *Quic — Wikipedia, the free encyclopedia*, [Online; accessed 29-May-2024], 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=QUIC&oldid=1224553354.

[15] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, Aug. 2022. DOI: 10.17487/RFC9293. [Online]. Available: https://www.rfc-editor.org/info/rfc9293.

[16] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. DOI: 10.17487/RFC8446. [Online]. Available: https://www.rfc-editor.org/info/rfc8446.

[17] A. Langley and W.-T. Chang, *QUIC Crypto*, Dec. 2016. [Online]. Available: https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit.

[18] M. Thomson and C. Benfield, *HTTP/2*, RFC 9113, Jun. 2022. DOI: 10.17487/RFC9113. [Online]. Available: https://www.rfc-editor.org/info/rfc9113.

[19] R. T. Fielding, M. Nottingham, and J. Reschke, *HTTP/1.1*, RFC 9112, Jun. 2022. DOI: 10.17487/RFC9112. [Online]. Available: https://www.rfc-editor.org/info/rfc9112.

[20] J. Postel, *User Datagram Protocol*, RFC 768, Aug. 1980. DOI: 10.17487/RFC0768. [Online]. Available: https://www.rfc-editor.org/info/rfc768.

[21] P. Karn and C. Partridge, "Improving round-trip time estimates in reliable transport protocols," *ACM Trans. Comput. Syst.*, vol. 9, no. 4, 364–373, 1991, ISSN: 0734-2071. DOI: 10.1145/118544.118549. [Online]. Available: https://doi.org/10.1145/118544.118549.

[22] A. Mishra, S. Lim, and B. Leong, "Understanding speciation in quic congestion control," in *Proceedings of the 22nd ACM Internet Measurement Conference*, ser. IMC '22, Nice, France: Association for Computing Machinery, 2022, 560–566, ISBN: 9781450392594. DOI: 10.1145/3517745.3561459. [Online]. Available: https://doi.org/10.1145/3517745.3561459.

[23] C. Huitema, S. Dickinson, and A. Mankin, *DNS over Dedicated QUIC Connections*, RFC 9250, May 2022. DOI: 10.17487/RFC9250. [Online]. Available: https://www.rfc-editor.org/info/rfc9250.

[24] T. Pauly, D. Schinazi, A. Chernyakhovsky, M. Kühlewind, and M. Westerlund, *Proxying IP in HTTP*, RFC 9484, Oct. 2023. DOI: 10.17487/RFC9484. [Online]. Available: https://www.rfc-editor.org/info/rfc9484.

[25] D. Schinazi, *Proxying UDP in HTTP*, RFC 9298, Aug. 2022. DOI: 10.17487/RFC9298. [Online]. Available: https://www.rfc-editor.org/info/rfc9298.

[26] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same standards, different decisions: A study of quic and http/3 implementation diversity," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20, Virtual Event, USA: Association for Computing Machinery, 2020, 14–20, ISBN: 9781450380478. DOI: 10.1145/3405796.3405828. [Online]. Available: https://doi.org/10.1145/3405796.3405828.

[27] A. Yu and T. A. Benson, "Dissecting performance of production quic," in *Proceedings of the Web Conference 2021*, ser. WWW '21, Ljubljana, Slovenia: Association for Computing Machinery, 2021, 1157–1168, ISBN: 9781450383127. DOI: 10.1145/3442381.3450103. [Online]. Available: https://doi.org/10.1145/3442381.3450103.

[28] ngtcp2, *Ngtcp2 project is an effort to implement ietf quic protocol*, https://github.com/ngtcp2/ngtcp2, 2023.

[29] A. Mishra and B. Leong, "Containing the cambrian explosion in quic congestion control," in *Proceedings of the 2023 ACM on Internet Measurement Conference*, ser. IMC '23, Montreal QC, Canada: Association for Computing Machinery, 2023, 526–539, ISBN: 9798400703829. DOI: 10.1145/3618257.3624811. [Online]. Available: https://doi.org/10.1145/3618257.3624811.

[30] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating quic performance over web, cloud storage, and video workloads," *IEEE Transactions on Network and Service Management*, vol. 19, no. 2, pp. 1366–1381, 2022. DOI: 10.1109/TNSM.2021.3134562.

[31] Anonymous, *Quic setup - emulab profile*, 2023. [Online]. Available: https://figshare.com/s/7a8c5015e26b14a6020a.

[32] Anonymous, *Long look quic modified scripts*, 2023. [Online]. Available: https://figshare.com/s/5767a449a1a006e2d6b0.

[33] *Remove code-paths for pre-ietf quic*, Chromium code review, accessed on 2024-03-11. [Online]. Available: https://chromium-review.googlesource.com/c/chromium/src/+/4265375.

[34] *Quic version 44 and ietf quic*, Google Groups discussion, accessed on 2024-03-11. [Online]. Available: https://groups.google.com/a/chromium.org/g/proto-quic/c/b6gZ18W5qn0.

[35] M. Thomson, *Version-Independent Properties of QUIC*, RFC 8999, May 2021. DOI: 10.17487/RFC8999. [Online]. Available: https://www.rfc-editor.org/info/rfc8999.

[36] B. White *et al.*, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, USENIX Association, Boston, MA, Dec. 2002, pp. 255–270.

[37] *Emulab documentation: Creating profiles*, https://docs.emulab.net/creating-profiles.html, Accessed on November 5, 2023.

[38] Emulab. "Emulab Testbed." (), [Online]. Available: https://www.emulab.net/.

[39] *Emulab hardware*, The Emulab Manual, Accessed on 2024-03-02, 2023. [Online]. Available: https://docs.emulab.net/hardware.html.

[40] Chromium, *Playing with quic*, https://www.chromium.org/quic/playing-with-quic/, 2023.

[41] A. Cardaci, *Chrome-har-capturer*, 2023. [Online]. Available: https://www.npmjs.com/package/chrome-har-capturer.

[42] E. L. B. N. Laboratory, *Iperf3: A tcp, udp, and sctp network bandwidth measurement tool*, Accessed on 2024-03-11, 2024. [Online]. Available: https://software.es.net/iperf/.

[43] *Tcpdump*, Accessed on 2024-03-05, 2024. [Online]. Available: https://www.tcpdump.org/.

[44] M. Carbone and L. Rizzo, "Dummynet revisited," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 2, 12–20, 2010, ISSN: 0146-4833. DOI: 10.1145/1764873.1764876. [Online]. Available: https://doi.org/10.1145/1764873.1764876.

[45] B. S. Ltd). and the University of Utah., *Geni-lib*, 2014. [Online]. Available: https://docs.cloudlab.us/geni-lib/intro/intro.html.

[46] N. Muthuraj, *QUIC Setup*, https://github.com/naveenrajm7/quic-setup, 2024.

[47] M. Brinn and c.-c. Rob Ricci, *Geni federation software architecture document*, 2012. [Online]. Available: https://groups.geni.net/geni/raw-attachment/wiki/GeniArchitectTeam/GENI\%20Software\%20Architecture\%20v1.0.pdf.

[48] N. Muthuraj, *Long Look QUIC*, https://github.com/naveenrajm7/long-look-quic/tree/feb-2023, 2024.

[49] Wikipedia contributors, *Ipfirewall — Wikipedia, the free encyclopedia*, [Online; accessed 25-April-2024], 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Ipfirewall&oldid=1145680286.

[50] Emulab, *Emulab bridgenodes*, 2023. [Online]. Available: https://wiki.emulab.net/wiki/BridgeNodes.

[51] Google, *Proto-quic (archived)*, Archived repository, accessed on 2024-03-05, 2024. [Online]. Available: https://github.com/google/proto-quic/tree/merge-to-60.0.3108.0.

[52] Google, *Quiche (quic, http, etc.)* Accessed on 2024-03-05, 2024. [Online]. Available: https://github.com/google/quiche.

[53] J. Odvarko, A. Jain, and A. Davies, *Http archive (har) format*, 2012. [Online]. Available: https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/HAR/Overview.html.

[54] A. Jones, *Fun with dummynet: Maintaining high throughput in large bandwidth-delay product networks*, Dec. 5. [Online]. Available: https://www.cs.unc.edu/~jeffay/dirt/FAQ/hstcp-howto.pdf.

[55] Chromium, *Receiver buffer size*, https://groups.google.com/a/chromium.org/g/proto-quic/c/PyENXwCs1qc, 2023.

[56] Chromium, *Old congestion window*, https://source.chromium.org/chromium/chromium/src/+/refs/tags/52.0.2743.116:net/quic/congestion_control/send_algorithm_interface.cc;l=23, 2027.

[57] Chromium, *New congestion window*, https://source.chromium.org/chromium/chromium/src/+/main:net/third_party/quiche/src/quiche/quic/core/quic_protocol_flags_list.h;l=180, 2023.

[58] M. R. Wong and S. Tieu, *Reproducing "taking a long look at quic"*, https://reproducingnetworkresearch.files.wordpress.com/2020/06/wong_tieu.pdf, June 11, 2020.

[59] G. Carlucci, L. De Cicco, and S. Mascolo, "Http over udp: An experimental investigation of quic," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, Salamanca, Spain: Association for Computing Machinery, 2015, 609–614, ISBN: 9781450331968. DOI: 10.1145/2695664.2695706. [Online]. Available: https://doi.org/10.1145/2695664.2695706.

[60] S. Rostedt, *Ftrace - function tracer*, 2017. [Online]. Available: https://www.kernel.org/doc/html/v5.0/trace/ftrace.html.

[61] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *ACM Queue*, vol. 14, September-October, pp. 20 –53, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=3022184.

[62] M. Hock, R. Bless, and M. Zitterbart, "Experimental evaluation of bbr congestion control," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, 2017, pp. 1–10. DOI: 10.1109/ICNP.2017.8117540.

[63] H. Anvari and P. Lu, "Machine-learned recognition of network traffic for optimization through protocol selection," *Computers*, vol. 10, no. 6, p. 76, 2021. DOI: 10.3390/computers10060076.

[64] Wikipedia contributors, *Welch's t-test — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Welch%27s_t-test&oldid=1181349917, [Online; accessed 26-October-2023], 2023.

[65] S. Ha and I. Rhee, "Taming the elephants: New tcp slow start," *Comput. Netw.*, vol. 55, no. 9, 2092–2110, 2011, ISSN: 1389-1286. DOI: 10.1016/j.comnet.2011.01.014. [Online]. Available: https://doi.org/10.1016/j.comnet.2011.01.014.

[66] E. Blanton, D. V. Paxson, and M. Allman, *TCP Congestion Control*, RFC 5681, Sep. 2009. DOI: 10.17487/RFC5681. [Online]. Available: https://www.rfc-editor.org/info/rfc5681.

[67] Y. Cheng, N. Cardwell, N. Dukkipati, and P. Jha, *The RACK-TLP Loss Detection Algorithm for TCP*, RFC 8985, Feb. 2021. DOI: 10.17487/RFC8985. [Online]. Available: https://www.rfc-editor.org/info/rfc8985.

[68] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, "Quic on the highway: Evaluating performance on high-rate links," in *2023 IFIP Networking Conference (IFIP Networking)*, 2023, pp. 1–9. DOI: 10.23919/IFIPNetworking57963.2023.10186365.

Appendix A: Emulab profile

Listing A.1: Emulab profile for QUIC experiments

```
"""# The profile for experimenting with QUIC protocol
The profile has three nodes: **server**, **client** and **link**.
The Execute script install required packages for running experiments.

Instructions:
After the experiment is instatiated,
Start running example server from server node,
and example client from client node.
"""


# Import the Portal object.
import geni.portal as portal
# Import the ProtoGENI library.
import geni.rspec.pg as pg
# Import the emulab extensions library. (BridgedLink)
import geni.rspec.emulab as emulab

# Create a portal context, needed to defined parameters
pc = portal.Context()

# Create a Request object to start building the RSpec.
request = pc.makeRequestRSpec()

# Describe the parameter(s) this profile script can accept.
# quic_version is used to decide which OS version, chrome-har-capturer version is
    used.
pc.defineParameter( "quic_version", "Specify the quic version to setup (Q037, RFCv1)"
    , portal.ParameterType.STRING, "RFCv1" )
# project is used to specify the project path to give permissions to apache server.
pc.defineParameter( "project", "Specify the emulab project name", portal.
    ParameterType.STRING, "FEC-HTTP" )

# Retrieve the values the user specifies during instantiation.
params = pc.bindParameters()

# Check parameter validity.
# Add custom conditions here
valid_versions = ["Q037", "RFCv1"]
if params.quic_version not in valid_versions:
    error = portal.ParameterError("Invalid quic_version. It should be either 'Q037'
        or 'RFCv1'.", ['quic_version'])
    pc.reportError(error)

# this function will spit out some nice JSON-formatted exception info on stderr
pc.verifyParameters()

# Add a raw PC to the request.
server = request.RawPC("server")
# d430 -> 64GB ECC Memory, Two Intel E5-2630v3 8-Core CPUs at 2.4 GHz (Haswell)
server.hardware_type = 'd430'
```

```
# https://docs.emulab.net/advanced-topics.html , Public IP Access
# server.routable_control_ip = True
iface1 = server.addInterface()
# Specify the IPv4 address
iface1.addAddress(pg.IPv4Address("192.168.1.1", "255.255.255.0"))

client = request.RawPC("client")
# d710 -> 12 GB memory, 2.4 GHz quad-core
client.hardware_type = 'd710'
# client.routable_control_ip = True
iface2 = client.addInterface()
# Specify the IPv4 address
iface2.addAddress(pg.IPv4Address("192.168.1.2", "255.255.255.0"))

ubuntu_22 = "urn:publicid:IDN+emulab.net+image+emulab-ops//UBUNTU22-64-STD"
ubuntu_18 = "urn:publicid:IDN+emulab.net+image+emulab-ops//UBUNTU18-64-STD"
fbsd_image = "urn:publicid:IDN+emulab.net+image+emulab-ops:FBSD132-64-STD"

# Request that a specific image be installed on this node
ubuntu_image = ubuntu_22 if params.quic_version == 'RFCv1' else ubuntu_18
server.disk_image = ubuntu_image
client.disk_image = ubuntu_image

# Create the bridged link between the two nodes.
link = request.BridgedLink("link")
link.bridge.hardware_type = 'd710'
# Add the interfaces we created above.
link.addInterface(iface1)
link.addInterface(iface2)

link.bridge.disk_image = fbsd_image

# Give bridge some shaping parameters. (Implict parameter found in real link)
# link.bandwidth = 10000
# link.latency   = 36  # Implicit latency in live network link (IMC'17)

# pass variable to script
project = params.project
# Install and execute a script that is contained in the repository.
server.addService(pg.Execute(shell="sh", command="export PROJECT="+ project + "
    QUIC_VERSION="+ params.quic_version +"&& /local/repository/scripts/install-deps.
    sh"))
client.addService(pg.Execute(shell="sh", command="export PROJECT="+ project + "
    QUIC_VERSION="+ params.quic_version +"&& /local/repository/scripts/install-deps.
    sh"))

# Install specific packages
server.addService(pg.Execute(shell="sh", command="/local/repository/scripts/install-
    apache.sh"))
client.addService(pg.Execute(shell="sh", command="export QUIC_VERSION="+ params.
    quic_version +"&& /local/repository/scripts/install-client.sh"))
link.bridge.addService(pg.Execute(shell="sh", command="/local/repository/scripts/
    bridge-tunning.sh"))

# Print the RSpec to the enclosing page.
pc.printRequestRSpec(request)
```

Appendix B: PLT Values

Table B.1: Values for Figure 4.6a. Varying object size, 0% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	106.37	183.41	110.71	185.87	187.55	303.04	353.05	389.39	535.41	647.64	951.41	1057.93	8567.22	8612.22	
STD	2.17	9.44	1.82	1.77	2.25	2.72	25.22	2.03	2.59	2.04	7.11	2.25	2.18	1.97	10 Mbps
% diff	72		68		62		10		21		11		0.53		
Avg	102.67	174.23	102.48	175.84	151.68	284.69	186.09	301.94	239.05	373.34	324.13	456.61	1864.58	2004.53	
STD	2.32	1.32	2.15	1.53	2.31	2.18	2.75	2.24	1.74	2.28	2.72	1.76	2.52	4.53	50 Mbps
% diff	70		72		88		62		56		41		7.5		
Avg	102.77	174.61	102.94	174.11	151.32	284.29	183.93	293.74	227.87	364.47	269.95	409.03	1056.82	1180.75	
STD	1.92	2.65	2.58	2.24	2.68	2.65	2.2	2.64	2.5	2.75	2.52	3.93	2.83	9.1	100 Mbps
% diff	70		69		88		60		60		52		12		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.2: Values for Figure 4.6d. Varying #object, 0% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	953.45	1057.57	971.38	1077.74	971.82	1076.85	970.48	1078.01	1045.63	1112.41	1148.46	1245.02	
STD	9.46	1.62	4.53	2.01	7.41	1.71	7.13	1.86	12.01	7.32	17.33	31.35	10 Mbps
% diff	11		11		11		11		6.4		8.4		
Avg	322.75	460.92	326.79	460.71	324.74	460.04	325.11	470.5	707.12	535.35	822.8	807.31	
STD	2.05	17.94	2.15	2.55	2.8	2.41	2.44	27.61	62.34	5.92	48.55	19.09	50 Mbps
% diff	43		41		42		45		-24		-1.9		
Avg	270.42	408.77	270.95	412.12	272.28	412.4	268.94	411.54	727.09	533.73	821.93	807.81	
STD	2.89	3.68	3.54	3.5	2.68	6.47	2.63	3.08	38.96	12.32	49.7	12.67	100 Mbps
% diff	51		52		51		53		-27		-1.7		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.3: Values for Figure 4.6b. Varying object size, 1% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	110.86	206.44	110.11	225.71	195.26	400.33	325.85	455.28	567.78	1126.55	1038.29	1475.56	13583.54	26191.19	
STD	18.29	74.36	2.31	105.33	27.68	143.76	51.21	126.59	63.27	611.61	143.65	608.31	970.26	2456.35	10 Mbps
% diff	86		1e+02		1.1e+02		40		98		42		93		
Avg	109.73	196.3	105.53	218.81	169.53	374.98	220.84	385.66	349.27	862.78	665.46	1363.64	11969.13	22052.33	
STD	20	68.95	8.33	91.23	18.62	146.22	42.47	117.31	154.8	482.99	342.53	757.22	2736.38	2451.55	50 Mbps
% diff	79		1.1e+02		1.2e+02		75		1.5e+02		1e+02		84		
Avg	110.53	252.24	120.49	209.42	169.9	319.23	241.1	498.91	335.65	888.5	509.71	1663.57	12257.21	22250.19	
STD	23.4	238.67	36.36	78.74	19.35	79.78	53.07	217.03	116.66	459.24	191.48	812.58	1795.45	2649.96	100 Mbps
% diff	1.3e+02		74		88		1.1e+02		1.6e+02		2.3e+02		82		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.4: Values for Figure 4.6e. Varying #object, 1% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	1030.01	1909.06	1152.28	1771.06	1008.56	1763.49	1119.25	1866.74	1256.2	1610.27	1369.24	2052.4	
STD	143.13	910.9	289.87	714.56	92.68	601.84	208.81	1007.49	190.68	675.97	222.64	839.02	10 Mbps
% diff	85		54		75		67		28		50		
Avg	723.97	1009.85	695.91	1467.3	856.51	1515.85	660.8	1264.49	1213.1	1647.89	1253.59	1937.12	
STD	374.86	761.71	444.9	1026.37	451.67	967.24	403.61	845.93	327.45	868.94	304.97	874.14	50 Mbps
% diff	39		1.1e+02		77		91		36		55		
Avg	550.45	1426.71	896.68	1738.04	803.83	1179.02	802.51	1558.08	1106.53	1609.25	1250.19	1776.43	
STD	274.3	704.54	396.27	1024.98	425.83	891.92	394.7	826.04	310.42	843.27	313.73	860.96	100 Mbps
% diff	1.6e+02		94		47		94		45		42		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.5: Values for Figure 4.6c. Varying object size, 0% loss, 112 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	260.28	485.6	263.6	490.22	420.01	827.5	531.21	912.67	792.3	1188.65	1211.32	1599.5	8821.83	9154.85	
STD	2.4	1.54	2.4	1.79	1.53	2.3	2.32	1.91	2.69	1.59	1.82	2.34	1.66	2.06	10 Mbps
% diff	87		86		97		72		50		32		3.8		
Avg	257.76	479.36	260.33	480.54	411.02	818.3	508.55	835.94	632.46	1055.33	742.85	1174.86	2297.05	2691.91	
STD	1.94	1.79	1.85	2.22	1.56	2.1	2.18	2.7	2.99	2.63	3.23	2.38	12.95	2.57	50 Mbps
% diff	86		85		99		64		67		58		17		
Avg	257.03	479.63	260.38	480.18	410.72	818.01	508.43	827.02	631.48	1051.48	735.34	1166.99	1534.43	1962.65	
STD	1.49	2.03	1.67	1.31	1.97	2.3	2.22	2.48	2.04	2.29	3.04	2.68	2.88	2.34	100 Mbps
% diff	87		84		99		63		67		59		28		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.6: Values for Figure 4.6f. Varying #object, 0% loss, 112 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	1211.49	1599.54	1224	1620.58	1221.21	1619.75	1218.96	1619.81	1623.98	1583.61	1498.51	1681.48	
STD	2.13	2.38	2.24	1.81	1.65	2.1	2.06	1.73	390.42	13.28	295.47	97.67	10 Mbps
% diff	32		32		33		33		-2.5		12		
Avg	743.44	1175.53	746.16	1179.48	743.04	1178.64	736.8	1179.51	1149.98	1136.01	1266.47	1283.03	
STD	2.32	2.78	3.18	2.58	2.02	2.96	2.89	3.15	552.09	9.2	427.93	16.62	50 Mbps
% diff	58		58		59		60		-1.2		1.3		
Avg	735.36	1167.82	736.43	1170.51	733.81	1169.4	731.95	1169.44	1081.39	1121.04	1125.02	1280.77	
STD	1.93	2.75	2.94	2.94	2.54	2.16	2.76	2.68	510.79	15.09	324.1	13.27	100 Mbps
% diff	59		59		59		60		3.7		14		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.7: Values for Figure 4.11b. Varying object size, 0% loss, 112 ms RTT, 50 ms jitter.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	257.58	489.84	252.21	454.99	416.92	827.7	584.77	940.86	1123.26	1325.95	1740.02	1926.39	11227.84	9848.51	
STD	29.51	39.46	19.76	43.49	30.93	63.6	94.65	73.26	284.79	133.55	519.21	186.19	2134.96	702.72	10 Mbps
% diff	90		80		99		61		18		11		-12		
Avg	256.47	472.65	262.78	466.25	409.36	820.05	562.44	871.78	1121.06	1117.65	1532.03	1470.13	9324.74	6429.17	
STD	25.15	40.19	39.03	45.35	62.4	67.18	120.06	80.29	342.88	147.81	508.3	324.72	3656.8	2182.96	50 Mbps
% diff	84		77		1e+02		55		-0.3		-4		-31		
Avg	260.27	479.42	257.84	476.44	437.45	807.2	628.85	846.44	874.81	1207.84	1461.02	1516.11	10460.01	6043.42	
STD	29.41	49.39	27.88	64.99	54.85	57.08	124.89	74.13	237.74	163.13	753.49	360.96	3734.96	2468.65	100 Mbps
% diff	84		85		85		35		38		3.8		-42		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.8: Values for Figure 4.11d. Varying #object, 0% loss, 112 ms RTT, 50 ms jitter.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	1539.17	1855.34	1600.22	1839.07	1817.51	1828.52	1652.73	1858.42	2032.02	1941.51	2136.36	2107.1	
STD	425.65	232.33	513.11	260.39	479.64	221.85	482.32	217.73	550.89	302.76	532.96	411.26	10 Mbps
% diff	21		15		0.61		12		-4.5		-1.4		
Avg	1709.41	1423.43	1487.7	1708.18	1782.76	1598.1	1627.41	1567.44	1919.6	1727.42	2321.31	1709	
STD	620.92	291.7	638.66	402.74	657.22	326.24	541.67	297.72	673.35	510.81	688.61	313.71	50 Mbps
% diff	-17		15		-10		-3.7		-10		-26		
Avg	1770.1	1446.17	1787.54	1619.35	1515.38	1640.94	1615.06	1557.87	1821.09	1711.19	2153.56	1867.54	
STD	671.56	360.35	758.91	371.53	697.52	321.21	588.52	401.67	760.71	386.1	633.95	285.56	100 Mbps
% diff	-18		-9.4		8.3		-3.5		-6		-13		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.9: Values for Figure 4.5a. Varying object size, 0% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	104.73	175.04	108.77	179.02	186.11	300.71	272.28	389.16	531.72	652.26	942.02	1085.18	8541.5	8624.62	
STD	0.91	0.86	1.09	1.08	1.17	1.09	1.18	0.93	1.38	1.57	0.98	42.37	50.36	1.11	10 Mbps
% diff	67		65		62		43		23		15		0.97		
Avg	99.52	169.5	100.06	170.43	150.25	281.47	183.56	316.62	238.3	372.46	319.66	457.04	2320.5	2460.84	
STD	0.85	1.15	1.17	1.49	1	1.34	1.43	5.61	2.52	1.78	1.39	1.9	360.33	628.25	50 Mbps
% diff	70		70		87		72		56		43		6		
Avg	99.19	169.27	100.15	169.4	149.08	280.7	181.32	301.33	228.11	362.17	270.23	404.82	1416.39	1176.88	
STD	1.12	1.11	1.55	1.11	1.11	1.29	1.18	14.45	3.13	1.93	5.11	2.02	235.71	32	100 Mbps
% diff	71		69		88		66		59		50		-17		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.10: Values for Figure 4.5d. Varying #object, 0% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	941.89	1070.72	962.03	1092.38	963.13	1094.46	964.29	1094.5	992.34	1096.64	1059.44	1130.23	
STD	1.38	1.21	0.89	1.15	1.17	5.37	1.59	1.95	23.34	3.45	83.32	5.09	10 Mbps
% diff	14		14		14		14		11		6.7		
Avg	326.96	459.29	323.94	462.56	324.19	461.03	324.49	462.48	621.21	464.47	609.95	583.12	
STD	28.99	7.04	2.4	3.25	1.33	2.44	1.42	2.39	89.88	5.67	47.96	11.59	50 Mbps
% diff	40		43		42		43		-25		-4.4		
Avg	268.88	405.24	274.07	407.82	277.8	408.53	273.25	409.25	624.11	417.33	597.61	572.02	
STD	3.3	2.28	6.04	2.89	12.43	2.11	4.93	3.19	95.44	5.73	37.43	10.51	100 Mbps
% diff	51		49		47		50		-33		-4.3		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.11: Values for Figure 4.5b. Varying object size, 1% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	109.56	178	125.25	208.69	202.02	474.2	287.78	684.08	578.17	892.43	944.83	1783.29	8968.12	16961.06	
STD	12.34	7.99	48.88	73.48	51.13	94.82	27.06	71.19	100.16	207.29	8.63	489.15	329.78	1494.64	10 Mbps
% diff	62		67		1.3e+02		1.4e+02		54		89		89		
Avg	108.44	174.45	100.2	199.82	156.64	428.16	201.37	698.26	280.09	1088.82	534.33	1798.89	5923.66	16265.16	
STD	36.62	16.59	1.15	86.47	16.4	62.85	20.21	149.78	76.32	224.64	234.57	511.69	1560.84	1192.81	50 Mbps
% diff	61		99		1.7e+02		2.5e+02		2.9e+02		2.4e+02		1.7e+02		
Avg	113.45	185.37	115.72	172.52	169.67	457.98	205.75	568.58	290.17	1118	461.49	1292.74	5867.75	16798.87	
STD	45.1	25.76	46.66	13.36	40.75	156.27	30.21	103.23	64.41	261.28	176.07	374.87	1291.16	2090.2	100 Mbps
% diff	63		49		1.7e+02		1.8e+02		2.9e+02		1.8e+02		1.9e+02		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.12: Values for Figure 4.5e. Varying #object, 1% loss, 36 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	981.02	2048.42	979.69	1771.26	964.43	1781.55	994.5	2247.93	1015.3	1811.35	1046.72	1971.08	
STD	80.86	509.48	61.2	544.99	4.6	457.06	62.12	338.08	29.77	302.46	73.52	403.44	10 Mbps
% diff	1.1e+02		81		85		1.3e+02		78		88		
Avg	560.69	1892.52	488	1703.73	498.81	1251.33	432.32	1179.08	714.52	1283.82	810.71	1152.31	
STD	258.56	459.09	160.45	281.75	198.38	257	150.65	302.97	232.14	414.5	158.15	347.19	50 Mbps
% diff	2.4e+02		2.5e+02		1.5e+02		1.7e+02		80		42		
Avg	430.72	1787.56	442.61	1625.08	423.11	1760.23	442.24	1870.25	751.55	2204.79	770.65	2408.05	
STD	173.14	334.19	188.85	380.28	104.27	345.43	137.32	403.01	99.05	505.06	130.97	450.92	100 Mbps
% diff	3.2e+02		2.7e+02		3.2e+02		3.2e+02		1.9e+02		2.1e+02		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.13: Values for Figure 4.5c. Varying object size, 0% loss, 112 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	257.1	480.51	261.02	484.45	417.64	825.6	534.68	914.45	792.29	1233.13	1218.36	1648.09	8773.19	9203.46	
STD	1.27	1.07	1.2	1.17	0.63	1.4	1.65	1.25	1.51	68.7	27.84	17.14	1.75	19.24	10 Mbps
% diff	87		86		98		71		56		35		4.9		
Avg	252.45	475.18	253.26	474.76	405.8	816.68	508.97	857.54	633.3	1054.15	746.08	1185.39	2698.08	2738.84	
STD	1.13	1.1	1.09	1.13	1.08	0.94	1.75	6.54	2.86	3.05	2.26	46.78	1711.91	138.12	50 Mbps
% diff	88		87		1e+02		68		66		59		1.5		
Avg	252.51	474.85	252.49	474.8	406.15	816.37	508.11	844.74	633.09	1047.91	736.38	1166.33	1781.44	2044.88	
STD	1.1	1.21	1.08	0.7	1.27	0.73	1.56	16.21	1.8	2.71	2.38	8.32	439.9	117.03	100 Mbps
% diff	88		88		1e+02		66		66		58		15		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.14: Values for Figure 4.5f. Varying #object, 0% loss, 112 ms RTT.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	1217.54	1667.61	1240.51	1661.67	1241.72	1662.95	1244.02	1675.9	1267.13	1648.3	1257.65	1645.95	
STD	26.56	88.39	24.86	20.01	24.31	20.19	19.55	36.77	149.7	17.34	15.43	61.01	10 Mbps
% diff	37		34		34		35		30		31		
Avg	746.53	1184.27	749.53	1179.57	751.05	1180.42	750.67	1202.88	781.01	1172.8	793.94	1141.15	
STD	2.79	42.64	1.79	3.09	2.28	3.06	2.3	56.87	90.4	43.99	19.14	15.9	50 Mbps
% diff	59		57		57		60		50		44		
Avg	736.87	1165.88	738.8	1191.79	740.43	1186.8	747.07	1169.17	740.91	1181.18	984.2	1156.17	
STD	2.24	3.33	2.14	69.48	2.25	77.31	28.64	3.93	6.26	32.01	435.83	84.53	100 Mbps
% diff	58		61		60		57		59		17		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		

Table B.15: Values for Figure 4.11a. Varying object size, 0% loss, 112 ms RTT, 50 ms jitter.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	259.04	480.15	265.72	485.83	453.65	807.25	617.59	959.84	1192.04	1345.39	1904.64	1746.54	24825.78	9880.72	
STD	36.14	30.88	26.89	37.4	53.82	52.41	115.9	106.18	323.12	198.45	722.82	196.86	3570.03	506.86	10 Mbps
% diff	85		83		78		55		13		-8.3		-60		
Avg	244.82	480.92	242.95	485.19	430.87	784.63	603.27	871.83	920.46	1172.55	1854.38	1549.27	23875.52	4898.06	
STD	32.49	46.29	27.55	49.66	54.32	53.87	121.42	76.17	262.5	184.16	647.97	351.48	4598.88	1043.11	50 Mbps
% diff	96		1e+02		82		45		27		-16		-79		
Avg	241.9	455.87	248.37	473.38	430.54	822.05	587.48	824.92	1098.04	1089.77	1901.75	1310.84	24327.86	2833.4	
STD	19.83	52.16	30.54	29.8	71.92	56	101.8	69.26	336.82	86.03	617.94	190.12	5115.28	814.87	100 Mbps
% diff	88		91		91		40		-0.75		-31		-88		
	5 KB		10 KB		100 KB		200 KB		500 KB		1 MB		10 MB		

Table B.16: Values for Figure 4.11c. Varying #object, 0% loss, 112 ms RTT, 50 ms jitter.

	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	
Avg	2194.27	1928.75	2039.51	1947.71	1866.99	2057.53	1983.04	1787.75	2099.12	1950.31	1969.95	2125.73	
STD	748.93	335.13	626.45	256.27	575.56	299.96	639.45	190.61	603.62	265.97	623.06	273.12	10 Mbps
% diff	-12		-4.5		10		-9.8		-7.1		7.9		
Avg	1635.5	1464.03	1971.03	1526.45	2049.05	1467.63	1487.35	1644.2	1499.24	1453.73	1805.54	1439.42	
STD	802.89	337.97	632.15	358.63	517.37	275.13	680.2	328.15	598.46	212.84	679.1	253.89	50 Mbps
% diff	-10		-23		-28		11		-3		-20		
Avg	1626.07	1263.02	1696.82	1298.21	1813.36	1390.51	1651.75	1292.98	1727.91	1232.58	1811.22	1281	
STD	747.44	165.83	777.6	201.6	676.44	188.8	649.96	149.18	775.51	163.42	859.28	183.34	100 Mbps
% diff	-22		-23		-23		-22		-29		-29		
	1 MB x 1		500 KB x 2		200 KB x 5		100 KB x 10		10 KB x 100		5 KB x 200		