Replication: "Taking a long look at QUIC"

Naveenraj Muthuraj
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
nmuthura@ualberta.ca

Nooshin Eghbal
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
eghbal@ualberta.ca

Paul Lu
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
paullu@ualberta.ca

ABSTRACT

QUIC, a rapidly evolving protocol, has gained prominence with its standardization and increased adoption as HTTP/3 on the World Wide Web. Originating from Google in 2014, QUIC has seen significant changes in transitioning from Google QUIC (gQUIC) to an IETF standard in 2021. Understanding the performance of the current version of QUIC in comparison to its predecessors is crucial given its evolution and widespread adoption.

This study conducts a comprehensive performance evaluation of two versions of QUIC: Google QUIC version 37 (gQUICv37, 2017) and IETF QUIC version 1 (QUICv1, 2021). Following parameters and methodologies established by a notable QUIC paper from 2017, we replicate their experiments on gQUICv37 and extend it to QUICv1, leveraging the Emulab testbed to facilitate reproducible research.

We show that the performance advantages of QUIC over TCP, given by core features like 0-RTT and multiple streams, are consistent in gQUICv37 and QUICv1. However, notable performance differences arise between the versions due to the implementation of the new BBR congestion control algorithm and an updated loss-detection strategy in QUICv1, resulting in improved performance for QUICv1 under packet reordering scenarios. By utilizing Emulab and sharing our scripts, we enable replication and extension of our study for future QUIC versions.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network measurement**.

KEYWORDS

QUIC, TCP, transport-layer performance, Emulab, reproducibility

ACM Reference Format:
Naveenraj Muthuraj, Nooshin Eghbal, and Paul Lu. 2024. Replication: "Taking a long look at QUIC". In *Proceedings of the 2024 ACM Internet Measurement Conference (IMC '24), November 4–6, 2024, Madrid, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3646547.3688453

IMC '24, November 4–6, 2024, Madrid, Spain

ACM ISBN 979-8-4007-0592-2/24/11
https://doi.org/10.1145/3646547.3688453

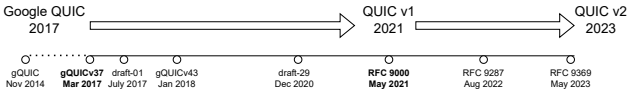


Figure 1: QUIC evolution timeline showing major QUIC versions and extensions. The versions in bold are the ones we studied in this work.

1 INTRODUCTION

Over the years, the QUIC transport protocol has undergone significant changes, evolving from its inception by Google to its standardization by the IETF (Figure 1). With the emergence of the Hypertext Transfer Protocol v3 (HTTP/3), which adopts QUIC as its underlying transport-layer protocol, understanding QUIC's performance is important to a large fraction of Internet traffic. Similarly, given the continuous evolution of protocols like QUIC, ensuring reliable and reproducible benchmarking of its performance is also paramount.

The three contributions of our experiments are: (1) Connecting and replicating QUIC's performance today (QUICv1), to a key paper from the past (gQUICv37) [23]; (2) Sharing an experimental methodology, based on Emulab [42], to enable other researchers to replicate our results both today and in the future; (3) Root-cause analysis to understand the performance differences between QUIC and TCP, as well as between gQUICv37 and QUICv1. Our benchmarking provides insights into the evolution of QUIC's performance characteristics.

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. The starting point for our replication (and extension) study is the 2017 IMC paper "Taking a long look at QUIC" by Kakhki et al. [23]. First, we replicate (most of) the experiments on gQUICv37. Second, we reproduce the same experiments using QUICv1 and draw comparisons with the results obtained from gQUICv37. All our experiments are conducted on Emulab [42], which enables reproducible research.

Our results and root-cause analysis (Section 4) show that QUIC's performance advantages over TCP, due to fundamental features like 0-RTT (Section 4.3) and the avoidance of 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 BBR (Section 4.7) and loss-detection strategies under packet reordering (Section 4.6) has notably improved in QUICv1 in comparison to its predecessor, gQUICv37. Additionally, contrary to the previous study [23], we show that a CUBIC parameter in the Chromium QUIC implementation has an impact on its fairness with TCP (Section 4.1).

2 BACKGROUND AND RELATED WORK

QUIC is a general-purpose transport-layer protocol introduced by Google circa 2014 [1], was released as IETF RFC 9000 QUIC version 1 (QUICv1) in 2021 [22], and has evolved to RFC 9369 QUIC version 2 (QUICv2) in 2023 [15]. The literature on QUIC's performance covers its adoption [26], implementations [27], and performance under various network parameters [23].

The motivations behind QUIC's development include the limitations of existing transport protocols, and the separation of protocols in layers for connection establishment and encryption [26]. For example, QUIC integrates the cryptographic layer and the transport layer, saving on round-trip time (RTT) overheads. Specifically, QUIC has 1-RTT and 0-RTT handshakes for secure connection (re-)establishment. Similarly, QUIC was designed to support the multiple streams of an application layer (e.g., like HTTP/2, but using UDP) avoiding HOL blocking.

Over the years, a variety of papers have studied the performance of QUIC. The differences between implementations and methodologies have made for difficult comparisons [27]. The paper by Kakhki et al. [23] focuses on analyzing the performance of Google QUIC (gQUIC) across a sweep of network parameters. Their experimental setup used an Amazon EC2 instance hosting a Chromium QUIC server and an Apache TCP server, with clients running Chromium in a desktop environment (similar to Figure 2), over a live network. They measure performance using page load time (PLT).

Notably, we use the same methodology and metrics as Kakhki et al. [23] to enable easier comparisons. However, we do not use a live network, which can complicate reproducibility, and use the Emulab testbed instead. Kakhki et al. [23] observe that QUIC generally outperforms TCP on desktop environments, except in a jitter scenario (c.f., Figure 8 [23]). 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, we examine the impact of an updated loss-detection mechanism under jitter in QUICv1 in Section 4.6. 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 [23]). We look at this issue of fairness in Section 4.1.

Yu and Benson [46] 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 the BBR CCA in H2, while H3 uses CUBIC.

Besides the performance studies of QUIC, there are also studies that have focused on variations in QUIC implementations and their impact. For example, Marx et al. [27] compare 15 IETF QUIC and HTTP/3 implementations, and find that the there is a large heterogeneity between QUIC stacks. Additionally, they introduce the qlog and qvis tools to generate QUIC logs and visualize QUIC connections, to facilitate root-cause analysis of different QUIC implementation behaviour. Mishra et al. [29] study the differences in QUIC CCA implementations with respect to the reference (kernel) implementation. They find significant deviations between the existing QUIC implementation of standard CCAs (CUBIC, BBR, Reno) from the reference implementations. However, in follow-up work [28] on 11 popular open-source QUIC stacks, they find that most QUIC CCA implementations are conformant to kernel implementations for shallow buffers, but less so for deep buffers.

Some studies only evaluated gQUIC [23] [26], or only the early draft versions of IETF QUIC [46][27], or both [37]. To the best of our knowledge, there is no prior work comparing the older gQUIC with the *standardized* IETF QUIC [22] to assess the changes in a single implementation over 5 or more years.

Our work takes a unique approach by benchmarking two different versions of QUIC (gQUICv37 and QUICv1) from the same implementation stack (Chromium), distinguished by two standards (Google QUIC and IETF QUIC), to assess five years of protocol changes (2017 to 2021). Using Emulab [42] as the testbed, we have configured the experiment profile [16] to facilitate the effortless replication of our experiment, while still maintaining pluggable aspects such as workloads, protocol versions, and network conditions. We provide all the necessary components including workloads, testbed configurations [31], and automation scripts [32] to facilitate the complete replication and extension of our experiments.

3 METHODOLOGY

Building upon the work of Kakhki et al. [23], 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 [22]) as QUICv1 (i.e., version 1 as QUICv1) and just QUIC to refer to the protocol.

QUIC Versions: While both gQUICv37 and QUICv1 share the same fundamental QUIC design, they possess distinct features (e.g., different cryptographic protocols for encryption) stemming from their separate development paths. Previous work conducted in 2017 [23] did not include QUICv1, which was not finalized until 2021 [22]. QUIC version 2 (QUICv2) was standardized in May 2023. Per RFC 9369 [15], QUICv2's purpose is to mitigate ossification by changing the wire format. Since this does not define or enhance any performance features, we do not include QUICv2 in our performance benchmarking.

Google QUIC (gQUIC): Introduced as a successor to HTTP/2 [35], Google's version of QUIC (gQUIC) converged into the IETF QUIC beginning with gQUICv44 [38], which incorporated changes outlined in the IETF invariants draft [39]. During our experimentation with Google QUIC, we found that Chromium (from 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) [9]. Hence, our work is useful in capturing gQUIC performance before it is fully phased out.

In our evaluation, we benchmark version 37 of Google QUIC (gQUICv37). Previous work by Kakhki et al. [23] demonstrated that gQUIC versions 34 and 37 exhibited similar performance, attributing the difference primarily to an increase in the Maximum Allowed Congestion Window (MACW) from 430 to 2000 in version 37.

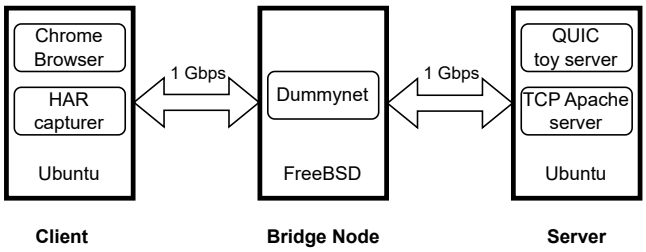


Figure 2: Experimental Topology in Emulab. The Server and Client are connected via a link Bridge Node, which shapes the traffic.

We access gQUICv37 from 2017 by leveraging an older Chromium codebase (proto-quic [18]) and a browser that supported this earlier version (Table 2, Column 'Replicated').

IETF QUIC (QUIC): IETF QUIC [22] represents the standardized and evolved version of Google QUIC [1]. Notably, IETF QUIC leverages TLSv1.3 for packet encryption, differing from gQUIC's use of its own cryptographic library [25]. While Google QUIC was started as a successor to HTTP/2+TLS, primarily to improve webpage load times, IETF QUIC defined a clear boundary between the transport layer and the application layer. RFC 9000 [22] established QUIC as a general-purpose transport-layer protocol. HTTP/3, as described in RFC 9114 [4] is the successor of HTTP/2 to use QUIC as its transport layer. Currently, QUIC version 1 (QUICv1) is supported by major web browsers like Chrome, Firefox, Safari, and Edge [44].

As our second QUIC version, we accessed QUICv1 by leveraging Google's quiche library (version C43017f) and the latest Chromium browser (version 111.0.5563.0), both of which are from 2023 (Table 2, Column 'Extended'). Google's quiche, officially QUICHE (QUIC, Http, Etc.) [19], is the production-ready implementation of QUIC, HTTP/2, HTTP/3, and related protocols, powering Google's servers, Chromium, Envoy, and other projects. This should not be confused with Cloudflare's QUIC library, which is also called quiche [14].

3.1 Emulab testbed

Emulab is a network emulation testbed facilitating reproducible research [42]. It provides researchers with complete control over the experimental testbed, allowing them to develop, debug and evaluate their system in a controlled environment. We use the primary installation [17] run by the Flux Group, part of the School of Computing at the University of Utah.

As shown in Figure 2, the topology 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 [2]. The d430 node has 64 GB of memory, and two Intel E5-2630v3 8-core CPUs running at 2.4 GHz. The d710 node has 12 GB of memory, and an 8-core Intel Xeon CPU E5530 running at 2.40 GHz. Both server and client nodes run Ubuntu 18.04 (Replicated) or 22.04 (Extended), while the link bridge node connecting them runs FreeBSD 13.2.

	Kakhki '17 [23]	This work
Client Machine	Desktop	Emulab Node
Server Machine	Amazon EC2	Emulab Node
Network Type	Live Network	Emulated Network
Traffic Shaping	TC Netem	Dummynet and TC
Workload	Simple web pages	Simple web pages

Table 1: Differences in experimental setup components. Use of Emulab and Dummynet enables reproducible experiments.

Similar to Kakhki et al. [23], 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 host, 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. Apache is used as the HTTP/2 server (with default Linux TCP stack configuration), and the QUIC sample server from Chromium [11] (called the "toy server") 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 work 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 the chrome-har-capturer tool [6] for request timing measurement. Apart from the browser, the client also has the necessary network tools to measure and validate the network parameters. For instance, we use iperf [24] to measure the bandwidth, and ping to measure the latency and packet loss. tcpdump [3] is used to capture the throughput.

To shape traffic between the server and client, we utilize Dummynet [5], 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.

In contrast to previous work by Kakhki et al. [23] (2017), our experimental setup differs in several key aspects as detailed in Table 1. While Kakhki et al. [23] used a desktop as the client and Amazon EC2 for the server, we utilize Emulab nodes for both client and server components. Moreover, Kakhki et al. [23]'s experiments were conducted on a live network, with traffic shaping achieved using TC/Netem. We employ an emulated network environment provided by Emulab, with traffic shaping implemented using Dummynet.

Since we evaluate two QUIC versions, we use two versions of the same setup as detailed in Table 2. In our first setup (Table 2, column "Replicated" for gQUICv37 experiments), aimed at replicating the conditions of the previous work by Kakhki et al. [23], we had to go back in time to evaluate older Google QUIC version 37 from 2017. Running the older gQUIC version posed several challenges and we had to use an old software stack (Chromium, har-capture-tool etc.,) to make gQUICv37 work. In our second setup (Table 2, column "Extended" for QUICv1 experiments), the same experimental topology' (Figure 2) was used but with updated versions to benchmark QUICv1.

3.1.1 Experimental Calibration. We selected Emulab as our testbed due to its controlled and reproducible environment. To match the default experimental conditions of Kakhki et al. [23], we added an

	Kakhki '17 [23]	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)	60.0.3112.101^	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	60.0.3112.101^	60.0.3108.1	111.0.5563.0
HAR Capturer tool	Unknown	0.9.5	0.14.0

Table 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. (^) Chromium 60.0.3112.101 was not available in proto-quic repo [18], so we use the closest available version 60.0.3108.0 .

Parameter	Values Tested	
	Kakhki '17 [23]	This work
Rate limits (Mbps)	5, 10, 50, 100	5, 10, 50, 100
Net Delay (RTT)	36ms, 112ms	36ms, 112ms
Extra Loss	0.1%, 1%	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 and TCP proxy	None
Clients	Desktop, Mobile	Emulab Node
Video qualities	tiny, medium, etc.,	None

Table 3: Differences in experimental parameters.

Scenario	Flow	Avg. Throughput in Mbps (std. dev.)
QUIC vs QUIC	QUIC 1	2.45 (0.04)
	QUIC 2	2.51 (0.04)
TCP vs TCP	TCP 1	2.46 (0.03)
	TCP 2	2.52 (0.03)
QUIC vs TCP	QUIC	3.74 (0.14)
	TCP	1.24 (0.14)
QUIC vs TCPx2	QUIC	2.59 (0.02)
	TCP 1	1.21 (0.03)
	TCP 2	1.18 (0.01)
QUIC vs TCPx4	QUIC	1.59 (0.03)
	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: Fairness: Average throughput (5 Mbps link, RTT = 36 ms, packet loss = 0 %, buffer = 30 KB, averaged over 5 runs) allocated to QUIC (QUICv1) and TCP flows when competing with each other. When both TCP and QUIC are using the CUBIC CCA, the unfairness caused by the QUIC flow is simply due to N = 2 connection emulation, a parameter in Chromium's CUBIC implementation.

implicit, baseline latency of 36 ms between the server and client in Emulab. Additionally, we made software calibrations to ensure fair comparison with their work. While Kakhki et al. [23] adjusted configurations of the QUIC sample server for performance, our investigation revealed that these adjustments were already incorporated in the current Chromium, including fixes for the slow start bug [12] and an increase in the maximum congestion window size [13][10]. Our work benefited from previous replication efforts [45].

3.2 Parameters, Workloads and Metrics

Table 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. [23] considered additional setup and workloads such as proxy servers, mobile environment, and video streaming performance, we focus on the parameters listed in Table 3 to evaluate the PLT performance of QUIC in a desktop-centric environment.

We use the same type of workloads as Kakhki et al. [23], which consist of simple web pages, comprising static HTML files referencing JPG images of various sizes and numbers. To prevent caching and compression, we include all necessary HTTP directives.

Similar to Kakhki et al. [23] we use PLT as our metric for comparison. PLT values are extracted from "onLoad" field of HAR files [33] produced by the chrome-har-capturer tool [6]. We make sure to exclude DNS lookup time from the total timing, to ensure we only capture the resource loading time.

Utilizing Kakhki et al. [23]'s scripts [30] and updating it to work with Emulab and the latest Chromium (QUICv1), we automate the experiments to ensure consistency and reproducibility [32]. The experimental testbed defined using the Emulab experimental profile is also shared [31]. To facilitate result reproducibility, we provide our synthesized workloads of web pages as part of our replication package [32].

4 EXPERIMENTAL RESULTS

We first discuss QUIC's fairness (Section 4.1), using CUBIC as the CCA, including a different finding than Kakhki et al. [23]. Further, from Sections 4.3 to 4.6 we present QUIC under loss, latency, and

jitter scenarios for both gQUICv37 and QUICv1, before extending QUICv1 performance benchmarking for the BBR CCA (Section 4.7).

Throughout this section, the performance numbers are not identical to Kakhki et al. [23], but the general trends are comparable, except where noted and discussed in more detail.

4.1 Fairness

We replicate the fairness experiments conducted by Kakhki et al. [23] to understand QUIC's fairness with TCP.

QUIC vs. QUIC Similar to the original study [23], we find that two QUIC flows are fair to each other (Table 4, QUIC vs QUIC), and two TCP flows (using the CUBIC CCA) (Table 4, TCP vs TCP) are (as expected) also fair.

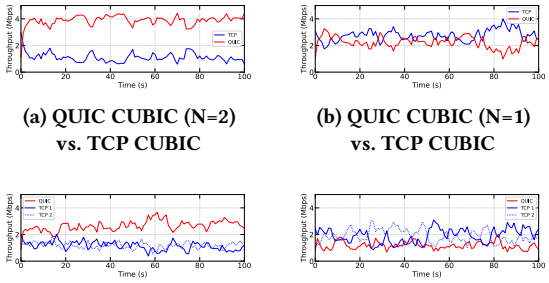
QUIC vs. TCP Also similarly, QUIC CUBIC (QUICv1 with default parameter N=2) is unfair to TCP (with CUBIC) (c.f., Table 4 in Kakhki et al. [23]). Figure 3a shows that QUIC CUBIC (red line) occupies twice the bandwidth of TCP CUBIC (blue line), which explains the 3.74 Mbps vs. 1.24 Mbps (Table 4, QUIC vs TCP). This unfairness is not surprising, and is consistent with Kakhki et al. [23], because the N=2 means that QUIC CUBIC is purposely emulating 2 TCP connections.

QUIC vs. multiple TCP connections We also reproduce the results to show that two TCP connections (Table 4, QUIC vs TCPx2 and Figure 3c) can occupy (in aggregate) half the throughput, versus one QUIC CUBIC connection, which confirms the basic interpretation of parameter N.

Difference in fairness result from Kakhki et al. [23]: To further explore the root cause and impact of parameter N (c.f. Figure 5 in Kakhki et al. [23]), we instrument the QUIC server code (Table 2, row 'QUIC server', column 'Extended') to extract Congestion Window (CWND) sizes for QUIC, and for TCP we use ftrace's tcpprobe event [36]. We observe that QUIC's CWND is larger than TCP's CWND when N = 2 (Figure 4a).

However, contrary to Kakhki et al. [23], we were able to force QUIC CUBIC to be fair to a single TCP flow by modifying the source code. Specifically, if we set N = 1 to force QUIC to emulate only one TCP connection (Figure 3b) then both QUIC and TCP grow their CWND similarly (Figure 4b). Kakhki et al. [23] also tried modifying their QUIC implementation, but they were not able to achieve fairness. Hence, they reported 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. Finally, we note that the CUBIC parameter N is only relevant to Google's implementation (i.e., quiche [19]), and not to others (e.g., ngtcp2).

The main takeaway is that the QUIC CUBIC protocol *can be fair* to TCP CUBIC-based flows. However, some implementations (e.g., Google) may have internal parameters (i.e., the N=2) that make it unfair by default. Our observation aligns with the conclusions drawn by Mishra et al. [29], which demonstrates 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 an average of 5 runs are summarised in the Table 4.



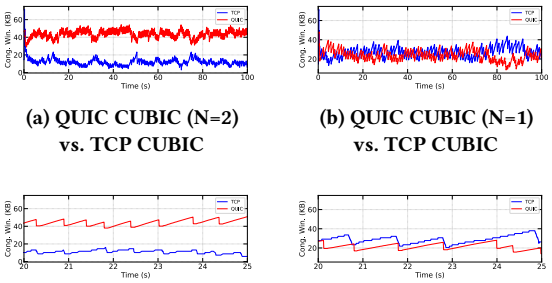
(a) QUIC CUBIC (N=2) vs. TCP CUBIC

(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 3: Fairness: Throughput timeline, QUIC CUBIC is unfair to TCP CUBIC when N = 2. Throughput of QUIC (QUICv1) 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

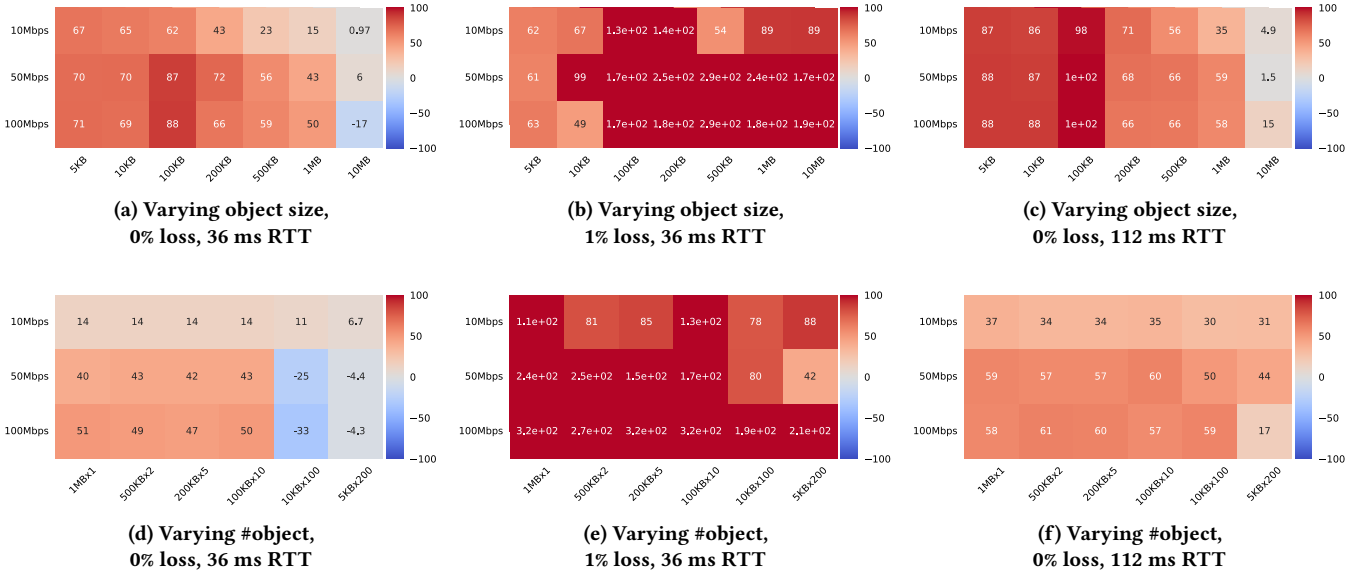
Figure 4: Fairness: CWND timeline, the growth of QUIC's CWND is influenced by the CUBIC parameter N. Timeline showing CWND sizes of QUIC (QUICv1) and TCP when transferring data over the same 5 Mbps bottleneck link (RTT = 36 ms, loss = 0 %, buffer = 30 KB).

4.2 Page Load Time

We assess the performance of QUIC and TCP under different emulated, rate-limit network settings (Table 3) by varying the bandwidth (5 Mbps, 10 Mbps, 50 Mbps and 100 Mbps), latency (36 ms, 112 ms) and loss (0%, 1%). Additionally, where applicable, we present design difference between gQUICv37 and QUICv1, and their impact on performance. As discussed in Section 3.2, we use page load time (PLT) as a metric for evaluating the performance of QUIC.

QUIC outperforms TCP under almost all conditions, except when 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, like the improved loss-detection mechanism in QUICv1 to adapt to packet reordering using dynamic reordering threshold, and show the performance impact of those changes.

In Figures 5 and 6, each square of a heatmap is annotated by and represents the percentage difference of TCP and QUIC PLTs averaged over 20 runs each (Equation 1). The percentage difference is mapped to a heatmap, from +100% (or more) faster for QUIC (red) to -100% slower for QUIC (blue, i.e., faster for TCP).



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

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

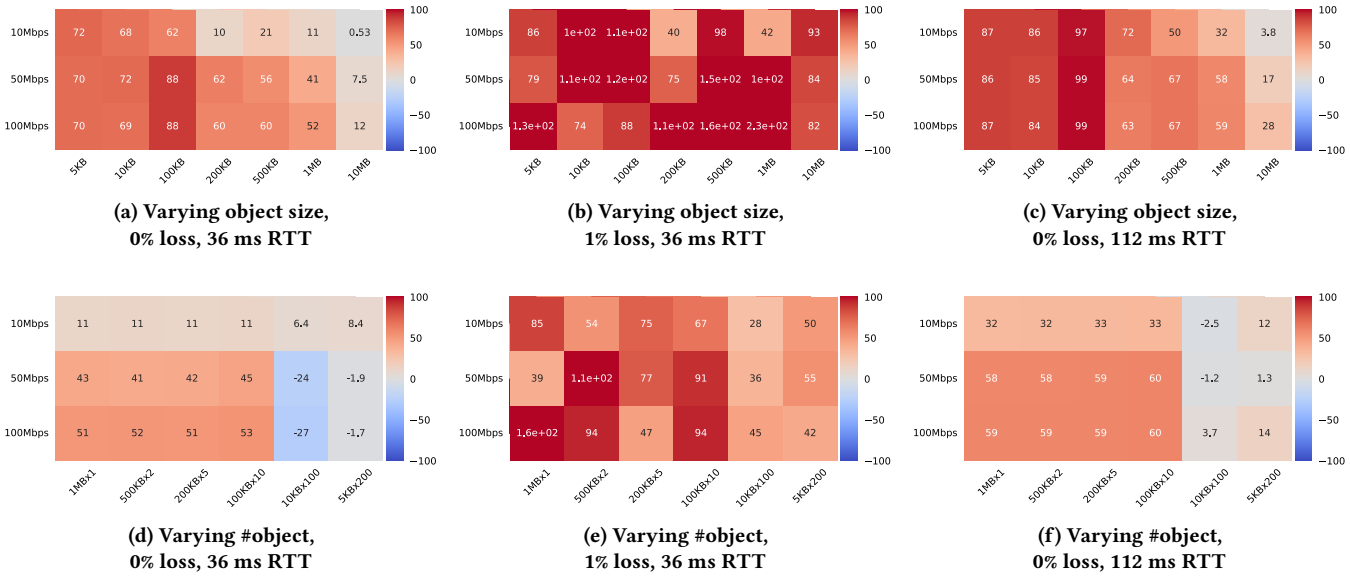
(c) Varying object size, 0% loss, 112 ms RTT

(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 5: Replication: gQUICv37 (CUBIC, N = 2) vs TCP (CUBIC). Red is better for gQUICv37. 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 6: Extension: QUICv1 (CUBIC, N = 2) vs TCP (CUBIC). Red is better for QUICv1. Blue is better for TCP.

$$\textit{Percentage Difference} = \frac{\overline{TCP} - \overline{QUIC}}{\overline{QUIC}} \times 100 \qquad (1)$$

where:

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

As done by Kakhki et al. [23], to reduce the chances that noise or variance in the experimental data invalidate the results, we perform Welch's t-test [43]. Welch's t-test assesses whether two populations have equal means. We conduct this test on 20 runs each (versus 10 runs by Kakhki et al. [23]) of TCP and QUIC PLTs, calculating the p-value. A p-value below our threshold (0.01) indicates a significant difference between TCP and QUIC performance, leading us to reject the null hypothesis of identical means. Conversely, a p-value above 0.01 suggests insignificant differences, represented by white

squares. Our heatmaps do not contain any white squares because all differences are significant, likely due to the increased number of runs (i.e., 20) conducted.

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

We start with the first scenario of no extra delay or packet loss, which we call the baseline setting. We set the baseline parameters of 36 ms RTT, 0% packet loss, and varying bandwidths of 10 Mbps, 50 Mbps and 100 Mbps as used previously [23]. As noted earlier (Section 3.1.1), the value of 36 ms comes from Kakhki et al. [23].

Strictly speaking, Figure 5 (see also Table 5) replicates the results of Kakhki et al. [23], but Figure 6 extends the results by using QUICv1 (which was not available in 2017). Some important notes: First, the N=2 parameter in Figures 5 and 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), 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, but that is not done here. Third, however, unless explicitly noted, N=2 for all QUIC CUBIC experiments to maintain commonality with Kakhki et al. [23].

4.3.1 QUIC's performance for single object. The heatmap of Figure 6a, shows the PLTs in the baseline setting for single objects of varying sizes. As we can see from Figure 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 a different object size) over three different bandwidths 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 a bandwidth of 10 Mbps (Figure 6a, top row).

We can observe a pattern that the red shading indicating better performance for QUIC generally decreases as we move from object sizes of 5 KB to 10 MB in the 10 Mbps bandwidth row in Figure 6a (with an exception of 100 KB object). As noted earlier, the highest performance gain of 72% for a 5 KB object, drops to 0.53% for a 10 MB object. A similar pattern can be observed for the bandwidth rows of 50 Mbps and 100 Mbps (with the exception of 100 KB object), where QUIC enjoys a 70% performance gain against TCP for an object size of 5 KB, before dropping to 7.5% for 50 Mbps bandwidth. The same is true for 100 Mbps of bandwidth, with QUIC performance dipping to 12% for 10 MB objects from 70% for 5 KB objects.

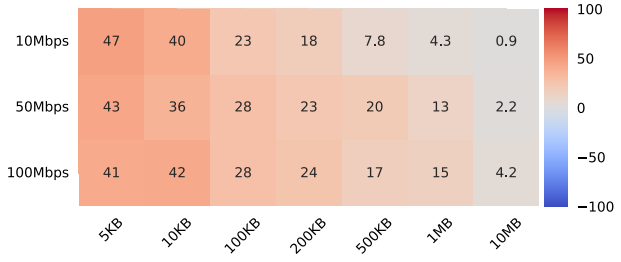


Figure 7: Performance comparison of QUICv1 with and without 0-RTT. The red color indicates the performance gain achieved by 0-RTT connections. The trend of decreasing 0-RTT advantage for larger objects in QUICv1 with TLSv1.3 is consistent with previous findings on gQUICv37 with QUIC Crypto [23].

The primary reason for QUIC's superior performance over TCP in the baseline setting stems from its utilization of 0-RTT connection time. QUIC connection establishment combines both transport and cryptographic handshakes reducing the connection time to 1-RTT for the initial connection and 0-RTT for subsequent connections. In contrast, TCP necessitates 3-RTT for all connections (1.5 for the TCP 3WHS and 1.5 for the TLS handshake). 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 trend explains the gradual decrease in QUIC's dominance as object size increases, particularly evident in the scenario with a 10 Mbps bandwidth (Figure 6a). Here, QUIC demonstrates only a 0.53% improvement over TCP for 10 MB objects, in comparison to the 72% improvement seen for 5 KB objects.

As with Kakhki et al. [23], 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 7). We show that the 0-RTT advantage amortizes as the object size increases. The original work, as detailed in Fig. 7 of their paper [23], 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 for larger objects. In our study, we conducted a replication of a similar experiment using QUICv1.

In Figure 7, we present a heatmap illustrating the PLT of QUICv1 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 7, reveal that 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 explains the diminishing performance of QUIC against TCP as the object size increases.

4.3.2 QUIC's performance for multiple objects. We look at QUIC's performance for different numbers of objects ranging from 1 to 200 as shown along the x-axis of Figure 5d for gQUICv37 and Figure 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 at 50 Mbps 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).

In Figures 5d and 6d, while QUIC has lower PLTs for the most squares of fewer numbers 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 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.

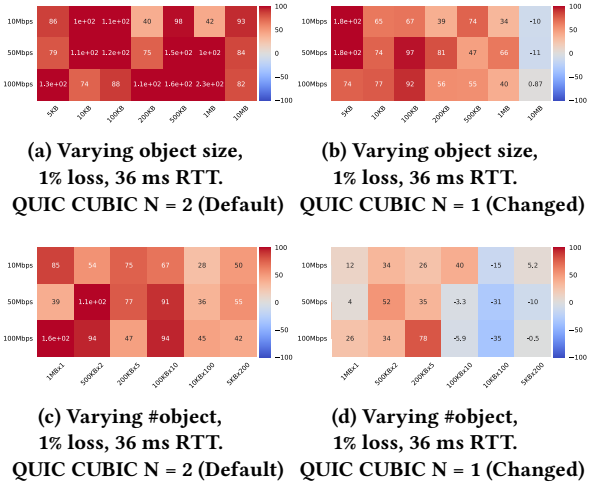
We validate the reasoning presented by Kakhki et al. [23] regarding QUIC's poor performance for large number of small objects. First, we experimented with different values for QUIC's Maximum Streams Per Connection (MSPC) parameter, which is 100 by default in Chromium. Our findings corroborate Kakhki's conclusion: decreasing the max streams per connection to 1 significantly worsens QUIC's performance, while increasing it to 500 does not result in any notable improvement. Second, we observed that QUIC exits the Hybrid Slow Start [20] early due to an increase in the estimated RTT when multiplexing a large numbers of objects, causing it to perform poorly compared to TCP. This behavior aligns with Kakhki's findings.

4.3.3 Comparing gQUICv37 and QUICv1 in baseline. In comparing the PLT of the baseline scenario for a single object between gQUICv37 (Figure 5a) and QUICv1 (Figure 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 5a, gQUICv37 consistently outperforms TCP across various scenarios, except for the 10 MB object for 100 Mbps bandwidth. Notably, gQUICv37 demonstrates a 67% improvement over TCP for a 5 KB object for 10 Mbps bandwidth (Figure 5a), mirroring QUICv1's 72% improvement for a similar object and bandwidth (Figure 6a). The pattern of 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 for 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 and 10 Mbps bandwidth (Figure 5a), resembling QUICv1's drop from 72% to 0.53% for the same object sizes and bandwidth (Figure 6a). These findings in our baseline settings align with the prior study on gQUICv37 [23], suggesting that the performance implications of 0-RTT for the 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, the 0-RTT mechanism has remained unchanged. We know that although gQUICv37 uses QUIC Crypto [25] and QUICv1 uses TLSv1.3 [40], 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.

From Figures 5d and 6d, we also note that the performance issue of gQUICv37 under a large number of small objects persists 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 5d. We see gQUICv37 performs poorly, indicated by dark blue squares in the 10KBx100 column of Figure 5d, where gQUICv37 is 25% slower than TCP for 10 KB x 100 objects and 50 Mbps bandwidth, and 33% slower for 100 Mbps bandwidth. Similarly, looking at Figure 6d for QUICv1, we see QUICv1 is 24% slower than TCP for 10 KB x 100 objects and 50 Mbps bandwidth, and 27% slower for 100 Mbps bandwidth.



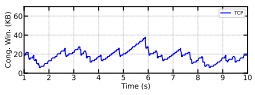
(a) Varying object size, 1% loss, 36 ms RTT. QUIC CUBIC N = 2 (Default)

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

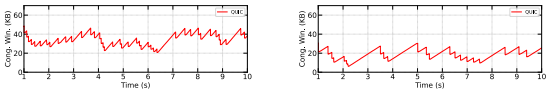
(c) Varying #object, 1% loss, 36 ms RTT. QUIC CUBIC N = 2 (Default)

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

Figure 8: Impact of N for packet loss, no competing flows: 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 Figures 6b and 6e, respectively. QUIC CUBIC with N = 2 outperforms TCP for large objects under packet loss, due to aggressive CWND update.



(a) TCP CUBIC CWND under packet loss



(b) QUIC CUBIC (N=2) CWND under packet loss

(c) QUIC CUBIC (N=1) CWND under packet loss

Figure 9: Congestion window over time for QUIC and TCP, 100 Mbps rate limit, 36 ms latency, and 1% packet loss. When QUIC CUBIC uses N = 2, its CWND growth is higher than that of TCP. However, when QUIC CUBIC uses N = 1, its CWND growth is similar to TCP. As a result, QUIC CUBIC with N = 2 outperforms TCP for larger objects under packet-loss conditions.

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

We look at the performance of QUIC under 1% packet loss. Emulab [42] provides link bridge nodes, which are equipped with Dummynet [5]. We use the Dummynet inside the link bridge node to introduce packet loss, as the link bridge node sits between server and client nodes (Figure 2).

QUIC's superior performance under packet loss is evident from Figures 5b and 5e for gQUICv37, and Figures 6b and 6e for QUICv1. Every square of Figure 5b is red, and the performance improvement for gQUICv37 over TCP ranges from a low of 48% for 10 KB at 100 Mbps, to a high of close to 300% for 500 KB at 100 Mbps. Similarly, QUICv1 in Figure 6b shows a similar trend of every square being red, and the performance improvement for QUICv1 over TCP ranges from a low of 40% for 200 KB at 10 Mbps to a high of close to 230% for 1 MB at 100 Mbps. Furthermore, both QUIC versions show similar trends with QUIC outperforming TCP throughout the range of objects and bandwidth for the multiple objects scenario, as seen from Figures 5e and 6e. QUIC's ability to avoid HOL blocking, due to independent streams, helps it to have better performance under lossy conditions.

4.4.1 Effect of N=2 on PLT under packet loss. In Section 4.1's discussion of fairness, we learned that QUIC CUBIC uses an emulated connection value N = 2, which results in QUIC being unfair to a TCP connection on a shared link. Now we discuss how CUBIC's N impacts QUIC performance under lossy conditions.

For QUIC CUBIC with a reduced N = 1 (Figures 8b and 8d), the performance of QUIC under packet loss diminishes for large objects and a large number of small objects. For example, in Figure 8b, while moving from the 5 KB to 10 MB object columns, we see that QUIC performance decreases, going from 180% for 5 KB to -10% for 10 MB objects at 10 Mbps. In contrast, when QUIC CUBIC is using N = 2 in Figure 8a, the performance of QUIC under packet loss conditions is better than TCP for all object sizes and bandwidths. When using N = 2, QUIC CUBIC deals with packet loss better than TCP.

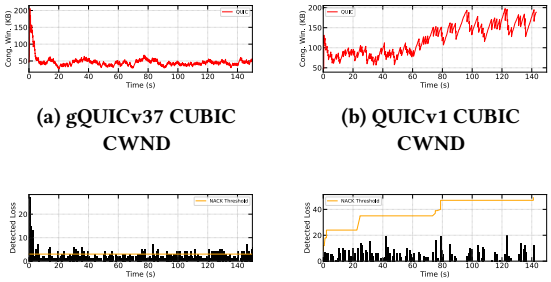
In Section 5.2 of Kakhki et al. [23], the performance improvement of QUIC under packet loss is attributed to the QUIC CCA's ability to cope with loss. This was backed by plotting CWND of QUIC and TCP under packet loss (c.f., Figure 9 [23]). We perform the same experiment, as shown in Figure 9. We observe 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 9a and 9c, we see that the QUIC CUBIC CWND for N = 1 of Figure 9c is same as TCP CWND of Figure 9a under loss condition. As we can see the CWND in both Figures 9a and 9c fluctuate between a low of around 5 KB and a high of just below 40 KB. Similarly, by looking at Figure 9a 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 a large object (10 MB) and multiple objects under packet loss was an artifact of QUIC CUBIC using N = 2.

In summary, QUIC's ability to avoid HOL blocking [4] results in better performance than TCP under lossy conditions. Furthermore, for larger objects (10 MB), QUIC CUBIC's default N = 2 results in QUIC being faster than TCP due to its aggressive CWND growth (Figures 9b). For scenarios with no packet loss (Figures 6a, 6c, 6d, 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

Figures 6c and 6f represents QUIC's performance with a RTT latency of 112 ms. We can observe that the trends from the baseline setting (36 ms RTT, 0% loss), in Figures 6c and 6f, are similar to that of the added latency setting (112 ms RTT, 0% loss) of Figures 6a and 6d, except for the fact that QUIC's dominance is more evident with a darker red color throughout the range of objects. For example, in baseline Figure 6a, we see QUIC outperforming TCP by 72% for a 5 KB object at 10 Mbps, increasing to QUIC outperforming TCP by 87% for the same object size and bandwidth (Figure 6c). The same is true for all other object sizes and bandwidths, where the performance difference between QUIC and TCP increases in the added latency setting, as compared to baseline setting.

The 0-RTT mechanism is the main reason behind QUIC's improved performance under higher latency (112 ms) (see Figure 6f). TCP's connection time is affected linearly by higher latency and QUIC's connection establishment time is less sensitive to RTT due to the fixed latency cost of 0-RTT as noted in Langley et al. [26].



(a) gQUICv37 CUBIC CWND

(b) QUICv1 CUBIC CWND

(c) gQUICv37 detected loss with static NACK threshold (3)

(d) QUICv1 detected loss with dynamic NACK threshold

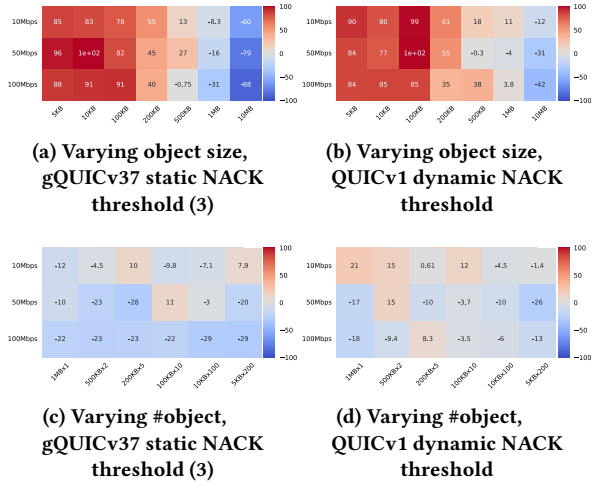
Figure 10: QUIC's loss-detection mechanisms often trigger false positives due to jitter-induced out-of-order packet delivery, hindering CWND growth. gQUICv37's use of a static threshold for packet reordering exacerbates this issue, leading to increased false positives. However, QUICv1 employs a dynamic threshold, effectively reducing false positives and facilitating CWND expansion. The timeline figures illustrate the relationship between CWND and detected losses during the transfer of a 210 MB object. (BW = 100 Mbps, Delay = 112 ms, Jitter = 50 ms, Loss = 0%).

4.6 QUIC with packet reordering due to jitter

Packet reordering can be caused by varying latency (jitter). In our setup, we induce a jitter of 50 ms to cause packet reordering. 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 11a, shows gQUIC's performance under jitter, which shows a trend of QUIC performing better for small objects of sizes from 5 KB to 500 KB. However, QUIC starts to suffer for large objects of sizes 1 MB and 10 MB (Figure 11a). Similarly, QUIC has poor performance as compared to TCP for a whole range of multiple objects (from 1 MB x 1 to 5 KB x 200) (Figure 11c).

4.6.1 Comparing gQUICv37 and QUICv1 under jitter. Now we compare the jitter results of gQUICv37 from Figures 11a and 11c to QUICv1 results from Figures 11b and 11d. We notice that the blue



(a) Varying object size, gQUICv37 static NACK threshold (3)

(b) Varying object size, QUICv1 dynamic NACK threshold

(c) Varying #object, gQUICv37 static NACK threshold (3)

(d) Varying #object, QUICv1 dynamic NACK threshold

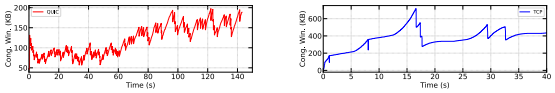
Figure 11: gQUICv37 vs TCP and QUICv1 vs TCP at 112 ms RTT with 50 ms jitter that causes packet reordering (Delay = 112 ms, Jitter = 50 ms, Loss = 0%). gQUICv37 uses a static NACK threshold of 3, while QUICv1 uses a dynamic NACK threshold. Performance improvement of Dynamic NACK is visible through blue squares getting dimmer for QUIC v1 PLT results (Subfigures (b) and (d)).

squares get lighter in both single object and multiple object scenarios. For example, in Figure 11b for QUICv1, the percentage difference for 10 MB objects at 10 Mbps is higher than the corresponding square of gQUICv37 in Figure 11a, which increase from -60% to -12%, indicating an improvement to QUIC over TCP. Similarly, in Figure 11d for QUICv1, the percentage difference for all objects at 100 Mbps is higher than the corresponding row of gQUICv37 in Figure 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 gQUICv37 and QUICv1 protocols. NACK threshold [1] (packet reordering threshold) is the number of packet reordering 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.

As shown in the timeline Figure 10c, with the fixed NACK threshold (of 3) of gQUICv37, the reordering of packets causes continuous premature loss detection, as indicated by the occurrence of black bars 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 10a, 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 PLTs, represented by blue squares in Figures 11a and 11c.

From Figure 10d, we observe that as the dynamic NACK threshold of QUICv1 increases (yellow line), the occurrence of premature loss (black bars) decreases. This is indicated by the sparser appearance of black bars after the x=80 s mark, coinciding with the NACK threshold rising above 40 on the y-axis, as shown by the yellow line. As a result of this reduction in premature loss, there is a gradual increase in the CWND, as depicted by the red line in Figure 10b. Initially, the CWND stays below 100 KB until the 60 s mark, after which it grows, reaching 200 KB by the 120 s mark. This growth in CWND correlates with the decrease in premature loss observed around the same time (Figure 10d). Consequently, QUICv1 is able to fully utilize the link's capacity, resulting in lower PLTs, as shown by the less-blue squares in Figures 11b (vs. 11a) and 11d (vs. 11c).



(a) QUICv1 CUBIC (N = 2) CWND

(b) TCP CUBIC CWND

Figure 12: Effects of packet reordering: QUIC's CWND growth is affected more by packet reordering than TCP's CWND growth during the transfer of a 210 MB object. (Bandwidth = 100 Mbps, Delay = 112 ms, Jitter = 50 ms, Loss = 0%).

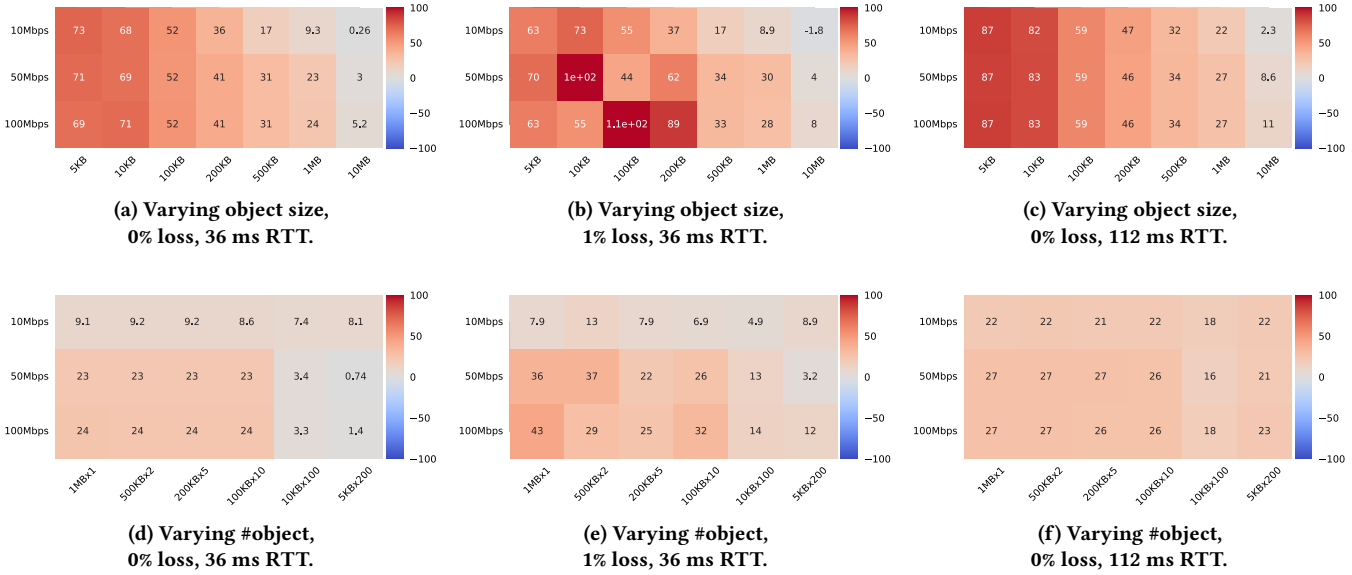
When there is packet reordering, even though QUICv1's dynamic NACK threshold improved performance over gQUICv37's static NACK threshold (from Figure 11c to Figure 11d), QUICv1 is still not on par with TCP's performance under jitter. We again look at the CWND of both QUICv1 and TCP protocols, to see possible reasons. As shown in the CWND timeline Figure 12, the CWND growth of TCP is higher than QUICv1 for the same CCA under jitter. This suggests that the loss-detection mechanism (dynamic NACK threshold) of QUICv1 is still 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 [21], recommends using TCP's RACK-TLP [8] for updating the reordering threshold, but we were not able to verify the implementation in Chromium QUIC through source code analysis. We also looked at other studies [29][28] which measure the conformation of Chromium QUIC implementation with TCP (kernel). While Mishra and Leong [28] show that Chromium QUIC's CUBIC is not conforming to TCP, it does not go into details about differences in implementation in terms of loss detection (except CUBIC emulated flows, 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 recommendation for loss detection under packet reordering.

4.7 QUIC with BBR

We demonstrate that QUIC with BBR outperforms TCP with BBR throughout the parameter sweep of our experiments (Figure 13). We further show that QUIC is even more effective when used with

[1] We use the term NACK threshold instead of packet reordering threshold to be consistent with previous work [23]. RFCs [21] use the term packet reordering threshold (kPacketThreshold).



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

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

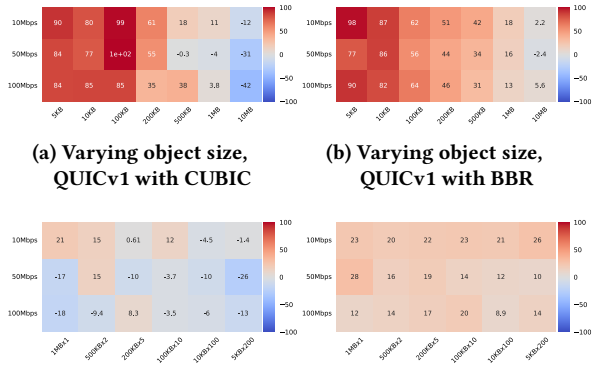
(c) Varying object size, 0% loss, 112 ms RTT.

(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 13: Extension: QUICv1 (BBR) vs TCP (BBR). Red is better for QUICv1. Blue is better for TCP



(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 14: 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.

the BBR CCA as compared to the CUBIC CCA, under a packet reordering scenario (Figure 14).

Since QUIC is implemented in the user-space, QUIC can evolve faster than other protocols. Additionally, QUIC is modular and has the ability to switch CCAs irrespective of its availability in the operating system kernel. While we cannot use BBR [7] with gQUICv37 (not readily available in 2017), we run QUICv1 with BBR and configure the Linux kernel to use TCP with BBR.

We perform the PLT experiments with both QUIC and TCP using BBR. When the QUIC BBR results (Figure 13) are compared to the QUIC CUBIC results (Figure 6), we find a similar trend under all three conditions (baseline Figure 6a, added loss Figure 6b and latency Figure 6c). In baseline Figure 13a, QUIC BBR outperforms TCP BBR for all object sizes and bandwidths, with the performance difference at 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 for larger objects is also observed in BBR (added latency) (Figure 13c). For example, in the 100 Mbps bandwidth row, QUIC BBR is 87% faster than TCP BBR for 5 KB objects, which drops to 11% for 10 MB objects.

However, we observe that the QUIC BBR figures under packet loss (Figure 13b) is closer to QUIC CUBIC with N = 1 (Figure 8b) than N = 2 (Figure 8a). The performance of QUIC BBR is similar to QUIC CUBIC N=1, because Chromium QUIC BBR does not have a parameter similar to the N (Emulated connection) of CUBIC. Further, in the case of multiple objects, QUIC BBR outperforms TCP BBR for all squares of Figures 13d and 13f, which is in contrast to QUIC CUBIC of Figures 6d and 6f.

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 14b with Figure 14a). For a easier comparison, we present the jitter results of QUICv1 with CUBIC and BBR side-by-side in Figure 14. Where Figures 14b and 14d represent BBR results (QUIC vs TCP), Figures 14a and 14c represent CUBIC results (QUIC vs TCP). By comparing the CUBIC (left of Figure 14) results with the BBR (right of Figure 14), we notice the disappearance of blue squares in the 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 at 50 Mbps).

The reason for QUICv1 BBR's better performance under jitter as compared to QUICv1 CUBIC is due to the fact that BBR does not rely on loss detection to regulate CWND. As seen previously

in Section 4.6 and Figure 10, packet reordering led to falsely detected loss in both gQUICv37 and QUICv1 CUBIC, reducing CWND and increasing PLT (lower performance). In contrast, BBR's loss-independent CWND regulation prevents performance degradation in QUIC with BBR under packet reordering.

4.8 Real Times for PLT

To help calibrate the performance results for future work, we present a selected number of datapoints in milliseconds (ms) (i.e., real time) (Table 5) for selected squares from Figures 5 and 6. Recall that the heatmaps are use ratios (Equation 1). For example, the 99.19 ms (gQUICv37, Baseline, 5 KB) versus 169.27 ms (TCP Linux 4.15) for a percentage difference of "71" is the bottom-left heatmap square of Figure 5(a). The 1534.43 ms (QUICv1, Latency 112 ms, 10 MB) versus 1962.65 ms (TCP Linux 5.15) for a percentage difference of "28" is the bottom-right heatmap square of Figure 6(c). The percentage difference of "-17" corresponds to the bottom-right heatmap square of Figure 5(a), and is a scenario where TCP 4.15 has lower PLT than gQUICv37. We discuss the relatively high standard deviation (235.71 ms) below.

Overall, these real times show that: First, the performance of TCP in 2017 and 2024 is comparable for both the "Baseline" (i.e., 36 ms RTT, no packet loss) and the "Latency 112 ms" (no packet loss) cases. Although several years of development separate Linux 4.15 and Linux 5.15 (Table 2), there may not have been large changes to the TCP stack that affect these experiments. Second, the performance of QUIC with the same Linux stacks in 2017 and 2024 are also either comparable to or better for the 2024 numbers. It is not surprising that QUICv1 is faster than gQUICv37. Further experiments would be required to attribute any performance improvements to specific changes in QUICv1, beyond the earlier discussion.

Interestingly, there are a couple of high standard deviations for gQUICv37 (e.g., 235.71 ms, 439.9 ms) that do not exist in the QUICv1 numbers. There is also a noticeably higher standard deviation (117.03 ms) for TCP Linux 4.15 and 10 MB resources, but that is not as high as for gQUICv37. Contention, resource limits, and implementation details are common causes for variability in performance. However, the root causes here are not clear. Overall, these selected numbers show some consistency between the 2017 and 2024 performance results.

High variability in PLTs in general is a broader issue in network benchmarking. We note that packet-loss datapoints from the heatmaps (not in our abbreviated Table 5) do exhibit more outliers and higher variance on the real times. Our analysis is incomplete, but recall that Kakhki et al. [23] use 10 runs for their averages, and we already use 20 runs for our averages, to reduce the effect of outliers. For future work, we will be repeating all experiments with 1,000 runs and doing more analysis. Also, we plan to do microbenchmarks to attribute any performance difference to a specific layer of software (Table 2).

			Replication 2017		Extension 2024	
			gQUIC v37	TCP Linux 4.15	QUIC v1	TCP Linux 5.15
Baseline	5 KB	Avg	99.19	169.27	102.77	174.61 ms
		STD	1.12	1.11	1.92	2.65
		% diff	71		70	
	10 MB	Avg	1416.39	1176.88	1056.82	1180.75
		STD	235.71	32	2.83	9.1
		% diff	-17		12	
Latency 112 ms	5 KB	Avg	252.51	474.85	257.03	479.63
		STD	1.1	1.21	1.49	2.03
		% diff	88		87	
	10 MB	Avg	1781.44	2044.88	1534.43	1962.65
		STD	439.9	117.03	2.88	2.34
		% diff	15		28	

Table 5: Selected PLT values in milliseconds (ms) for different heatmap squares in previous results. All values are for 100 Mbps bandwidth.

5 DISCUSSION AND FUTURE WORK

The original paper [23] includes important experiments that we have not yet replicated, including scenarios where QUIC is used in mobile computing and for video streaming. Now that we have, to some extent, established that Emulab is a reasonable platform for experimentation and open-sourced our software test environment, we hope to tackle these scenarios as a community.

Mobile computers, especially cell phones, are the primary computing devices for more and more users these days. Broadly speaking, mobile networks may suffer greater contention for bandwidth (compared to wired networks), may suffer bandwidth and error variance based on distance to cell towers, and may suffer more latency variance and packet reordering due to cell tower hand-offs. Even these properties may differ from LTE, to 5G, to future mobile technologies. Other than through the direct use of mobile phones [23], it can be complicated to create a satisfactory mobile testbed for the community.

On Emulab, we foresee approximating mobile environments by including more tests with shared network links (as a follow-up to and extension of Section 4.1) by extensive tests with jitter and packet reordering, by further analysis of fairness (including Jain fairness), and by empirically modeling the frequency and patterns of packet loss (e.g., bursty) as found on mobile networks. Our current approach to packet reordering (following the lead of Kakhki et al. [23]) can be made more sophisticated using a variety of queuing techniques. Mobile-specific emulators [41] might also be considered, as they have been parameterized using data-driven approaches.

Video streaming often dominates the total number of bytes carried across networks and is an important application for QUIC. If video streaming is accomplished by the sequential transfer of discrete objects (e.g., different portions of the video data), then the advantages shown by QUIC in our replicated experiments (so far) suggest that QUICv1 will also perform well for streaming. What is still required in future work is the direct use of QUIC to download YouTube videos (or similar) and measure the quality of experience (QoE) as done by Kakhki et al. [23]. In the context of Emulab, we would prefer to set up our own video server for ease of reproducibility and control.

6 CONCLUDING REMARKS

HTTP/3 over QUIC is the next generation of the World Wide Web. Revisiting, replicating, and extending a previous QUIC experiment [23] can be useful. Adopting common methodologies (e.g., similar parameter sweeps, metrics, and root-cause analysis via time-series CWND data) makes it easier to compare results and track progress over time.

Our replication has validated the performance benefits of key QUIC features, such as the 0-RTT/1-RTT connection. We have also extended the analysis by using a newer, IETF-based QUICv1 implementation from Google, with similar performance advantages. Finally, as a contribution to the research community, we provide the configurations [31] and scripts [32] for our Emulab [42] environment. In theory, this allows other researchers to explore new QUIC versions and extensions (e.g., MASQUE [34]).

As part of replication, our results are (in general) the same as Kakhki et al. [23]. Even though the specific patterns of the heatmaps (e.g., Figure 5) might be different, the general trends are comparable. Not surprisingly, reducing the RTT-based overheads have the greatest relative benefits for the smaller resource objects and fewer objects. The savings of 0-RTT/1-RTT handshakes can be amortized and see diminishing returns over longer data transfers. As in Kakhki et al. [23], there are still parameter combinations of larger objects and large number of small objects, for which TCP can still be comparable or faster. We also used the throughput and CWND analysis of Kakhki et al. [23] (e.g., Figures 3 and 4) to understand the root cause of QUIC's performance, including fairness and unfairness.

We extend the earlier work by including a performance analysis using Google's implementation of QUICv1, which was not available in 2017. As an IETF standard, QUICv1 is an important milestone. Our new analysis of QUICv1 includes results with the BBR CCA (Section 4.7), which has become more widely used in the past 7 years. We analyze how the improved packet reordering mechanism of QUICv1 represents a welcome improvement over gQUICv37, when there is packet reordering (Section 4.6). Finally, we demonstrated that the fairness properties of QUIC CUBIC can be enforced via the N parameter of the implementation, updating the earlier work.

With the adoption of the Emulab testbed, the community is able to further extend the parameter sweep of QUIC's performance and to more easily analyze other QUIC extensions. In fact, such analyses are part of our plan for future work.

ACKNOWLEDGMENTS

Thank you to Emulab for the generous access to their testbed. Many thanks to the anonymous reviewers and our shepherd. Thank you to Prajneet Singh Ruprai for help with some experiments.

A ETHICS

This work does not raise any ethical issues.

REFERENCES

[1] 2014. QUIC at 10,000 feet . https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU/edit
[2] 2023. Emulab Hardware. The Emulab Manual. https://docs.emulab.net/hardware.html Accessed on 2024-03-02.
[3] 2024. *tcpdump*. https://www.tcpdump.org/ Accessed on 2024-03-05.
[4] Mike Bishop. 2022. HTTP/3. RFC 9114. https://doi.org/10.17487/RFC9114
[5] Marta Carbone and Luigi Rizzo. 2010. Dummynet Revisited. *SIGCOMM Comput. Commun. Rev.* 40, 2 (apr 2010), 12–20. https://doi.org/10.1145/1764873.1764876
[6] Andrea Cardaci. 2023. chrome-har-capturer. https://www.npmjs.com/package/chrome-har-capturer
[7] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, September-October (2016), 20 – 53. http://queue.acm.org/detail.cfm?id=3022184
[8] Yuchung Cheng, Neal Cardwell, Nandita Dukkipati, and Priyaranjan Jha. 2021. The RACK-TLP Loss Detection Algorithm for TCP. RFC 8985. https://doi.org/10.17487/RFC8985
[9] Chromium. [n. d.]. Remove code-paths for pre-IETF QUIC. https://chromium-review.googlesource.com/c/chromium/src/+/4265375 Chromium code review, accessed on 2024-03-11.
[10] Chromium. 2023. 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.
[11] Chromium. 2023. Playing with QUIC. https://www.chromium.org/quic/playing-with-quic/.
[12] Chromium. 2023. Receiver buffer size. https://groups.google.com/a/chromium.org/g/proto-quic/c/PyENXwCs1qc.
[13] Chromium. 2027. 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.
[14] Cloudflare. 2024. Savoury implementation of the QUIC transport protocol and HTTP/3. https://github.com/cloudflare/quiche Accessed on 2024-09-02.
[15] Martin Duke. 2023. QUIC Version 2. RFC 9369. https://doi.org/10.17487/RFC9369
[16] Emulab. [n. d.]. Emulab Documentation: Creating Profiles. https://docs.emulab.net/creating-profiles.html. Accessed on November 5, 2023.
[17] Emulab. [n. d.]. *Emulab Testbed*. https://www.emulab.net/
[18] Google. 2024. proto-quic (archived). https://github.com/google/proto-quic/tree/merge-to-60.0.3108.0 Archived repository, accessed on 2024-03-05.
[19] Google. 2024. QUICHE (QUIC, Http, Etc.). https://github.com/google/quiche Accessed on 2024-03-05.
[20] Sangtae Ha and Injong Rhee. 2011. Taming the Elephants: New TCP Slow Start. *Comput. Netw.* 55, 9 (jun 2011), 2092–2110. https://doi.org/10.1016/j.comnet.2011.01.014
[21] Jana Iyengar and Ian Swett. 2021. QUIC Loss Detection and Congestion Control. RFC 9002. https://doi.org/10.17487/RFC9002
[22] J. Iyengar and M. Thomson. 2021. RFC 9000: QUIC: A UDP-Based Multiplexed and Secure Transport.
[23] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In *Proceedings of the 2017 Internet Measurement Conference* (London, United Kingdom) *(IMC '17)*. Association for Computing Machinery, New York, NY, USA, 290–303. https://doi.org/10.1145/3131365.3131368
[24] ESnet / Lawrence Berkeley National Laboratory. 2024. iperf3: A TCP, UDP, and SCTP network bandwidth measurement tool. https://software.es.net/iperf/ Accessed on 2024-03-11.
[25] Adam Langley and Wan-Teh Chang. 2016. QUIC Crypto. https://docs.google.com/document/d/1g5nIXAIkN_Y-7XJW5K45IblHd_L2f5LTaDUDwvZ5L6g/edit
[26] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 183–196. https://doi.org/10.1145/3098822.3098842
[27] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. 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* (Virtual Event, USA) *(EPIQ '20)*. Association for Computing Machinery, New York, NY, USA, 14–20. https://doi.org/10.1145/3405796.3405828
[28] Ayush Mishra and Ben Leong. 2023. Containing the Cambrian Explosion in QUIC Congestion Control. In *Proceedings of the 2023 ACM on Internet Measurement Conference* (Montreal QC, Canada) *(IMC '23)*. Association for Computing Machinery, New York, NY, USA, 526–539. https://doi.org/10.1145/3618257.3624811
[29] Ayush Mishra, Sherman Lim, and Ben Leong. 2022. Understanding Speciation in QUIC Congestion Control. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22)*. Association for Computing Machinery, New York, NY, USA, 560–566. https://doi.org/10.1145/3517745.3561459
[30] Arash Molavi. 2017. Taking a Long Look at QUIC scripts. https://github.com/arashmolavi/quic.
[31] Naveenraj Muthuraj. 2024. Replication: "Taking a long look at QUIC" - Emulab Profile. (8 2024). https://doi.org/10.6084/m9.figshare.26801056.v1
[32] Naveenraj Muthuraj. 2024. Replication: "Taking a long look at QUIC" - Scripts. (8 2024). https://doi.org/10.6084/m9.figshare.26801095.v1

[33] J. Odvarko, A. Jain, and A. Davies. 2012. HTTP Archive (HAR) format. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/HAR/Overview.html
[34] Tommy Pauly, David Schinazi, Alex Chernyakhovsky, Mirja Kühlewind, and Magnus Westerlund. 2023. Proxying IP in HTTP. RFC 9484. https://doi.org/10.17487/RFC9484
[35] Jim Roskind. 2013. QUIC: IETF-88 TSV Area Presentation. https://www.ietf.org/proceedings/88/slides/slides-88-tsvarea-10.pdf
[36] Steven Rostedt. 2017. ftrace - Function Tracer. https://www.kernel.org/doc/html/v5.0/trace/ftrace.html
[37] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2022. Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. *IEEE Transactions on Network and Service Management* 19, 2 (2022), 1366–1381. https://doi.org/10.1109/TNSM.2021.3134562
[38] Ian Swett. [n. d.]. QUIC Version 44 and IETF QUIC. https://groups.google.com/a/chromium.org/g/proto-quic/c/b6gZ18W5qn0 Google Groups discussion, accessed on 2024-03-11.
[39] Martin Thomson. 2021. Version-Independent Properties of QUIC. RFC 8999. https://doi.org/10.17487/RFC8999
[40] Martin Thomson and Sean Turner. 2021. Using TLS to Secure QUIC. RFC 9001. https://doi.org/10.17487/RFC9001
[41] Martino Trevisan, Ali Safari Khatouni, and Danilo Giordano. 2020. ERRANT: Realistic emulation of radio access networks. *Computer Networks* 176 (2020), 107289. https://doi.org/10.1016/j.comnet.2020.107289
[42] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. 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, 255–270.
[43] Wikipedia contributors. 2023. 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].
[44] Wikipedia contributors. 2024. QUIC — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=QUIC&oldid=1224553354 [Online; accessed 29-May-2024].
[45] Marc Robert Wong and Sarah Tieu. June 11, 2020. Reproducing "Taking a Long Look at QUIC". https://reproducingnetworkresearch.files.wordpress.com/2020/06/wong_tieu.pdf.
[46] Alexander Yu and Theophilus A. Benson. 2021. Dissecting Performance of Production QUIC. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/3442381.3450103