Taking a Long Look at QUIC

An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols

Arash Molavi Kakhki
Northeastern University
arash@ccs.neu.edu

Samuel Jero
Purdue University
sjero@purdue.edu

David Choffnes
Northeastern University
choffnes@ccs.neu.edu

Cristina Nita-Rotaru
Northeastern University
c.nitarotaru@neu.edu

Alan Mislove
Northeastern University
amislove@ccs.neu.edu

ABSTRACT

Google's QUIC protocol, which implements TCP-like properties at the application layer atop a UDP transport, is now used by the vast majority of Chrome clients accessing Google properties but has no formal state machine specification, limited analysis, and ad-hoc evaluations based on snapshots of the protocol implementation in a small number of environments. Further frustrating attempts to evaluate QUIC is the fact that the protocol is under rapid development, with extensive rewriting of the protocol occurring over the scale of months, making individual studies of the protocol obsolete before publication.

Given this unique scenario, there is a need for alternative techniques for understanding and evaluating QUIC when compared with previous transport-layer protocols. First, we develop an approach that allows us to conduct analysis across multiple versions of QUIC to understand how code changes impact protocol effectiveness. Next, we instrument the source code to infer QUIC's state machine from execution traces. With this model, we run QUIC in a large number of environments that include desktop and mobile, wired and wireless environments and use the state machine to understand differences in transport- and application-layer performance across multiple versions of QUIC and in different environments. QUIC generally outperforms TCP, but we also identified performance issues related to window sizes, re-ordered packets, and multiplexing large number of small objects; further, we identify that QUIC's performance diminishes on mobile devices and over cellular networks.

ACM Reference Format:
Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a Long Look at QUIC. In *Proceedings of IMC '17, London, United Kingdom, November 1–3, 2017,* 14 pages.
https://doi.org/10.1145/3131365.3131368

1 INTRODUCTION

Transport-layer congestion control is one of the most important elements for enabling both fair and high utilization of Internet links shared by multiple flows. As such, new transport-layer protocols typically undergo rigorous design, analysis, and evaluation—producing public and repeatable results demonstrating a candidate protocol's correctness and fairness to existing protocols—before deployment in the OS kernel at scale.

Because this process takes time, years can pass between development of a new transport-layer protocol and its wide deployment in operating systems. In contrast, developing an *application-layer* transport (i.e., one not requiring OS kernel support) can enable rapid evolution and innovation by requiring only changes to application code, with the potential cost due to performance issues arising from processing packets in userspace instead of in the kernel.

The QUIC protocol, initially released by Google in 2013 [10], takes the latter approach by implementing reliable, high-performance, in-order packet delivery with congestion control at the application layer (and using UDP as the transport layer).[1] Far from just an experiment in a lab, QUIC is supported by all Google services and the Google Chrome browser; as of 2016, more than 85% of Chrome requests to Google servers use QUIC [36].[2] In fact, given the popularity of Google services (including search and video), QUIC now represents a substantial fraction (estimated at 7% [26]) of all Internet traffic. While initial performance results from Google show significant gains compared to TCP for the slowest 1% of connections and for video streaming [18], there have been very few repeatable studies measuring and explaining the performance of QUIC compared with standard HTTP/2+TCP [17, 20, 30].

Our overarching goal is to understand the benefits and tradeoffs that QUIC provides. However, during our attempts to evaluate QUIC, we identified several key challenges for repeatable, rigorous analyses of application-layer transport protocols in general. *First*, even when the protocol's source code is publicly available, as QUIC's is, there may be a gap between what is publicly released and what is deployed on Google clients (i.e., Google Chrome) and servers. This requires gray-box testing and calibration to ensure fair comparisons with code running in the wild. *Second*, explaining protocol performance often requires knowing formal specifications and state machine diagrams, which may quickly become stale due to code evolution (if published at all). As a result, we need a way to automatically generate protocol details from execution traces and use them to explain observed performance differences. *Third*, given that application-layer protocols encounter a potentially endless

IMC '17, November 1–3, 2017, London, United Kingdom

ACM ISBN 978-1-4503-5118-8/17/11…$15.00
https://doi.org/10.1145/3131365.3131368

[1] It also implements TLS and SPDY, as described in the next section.
[2] Newer versions of QUIC running on servers are incompatibile with older clients, and ISPs sometimes block QUIC as an unknown protocol. In such cases, Chrome falls back to TCP.

array of execution environments in the wild, we need to carefully select and configure experimental environments to determine the impact of network conditions, middleboxes, server settings, and client device configurations on end-to-end performance.

In this work, we address these challenges to properly evaluate QUIC and make the following key contributions.

First, we identify a number of pitfalls for application-layer protocol evaluation in emulated environments and across multiple QUIC versions. Through extensive calibration and validation, we identify a set of configuration parameters that fairly compare QUIC, as deployed by Google, with TCP-based alternatives.

Second, we develop a methodology that automatically generates network traffic to QUIC- and TCP-supporting servers in a way that enables head-to-head comparisons. Further, we instrument QUIC to identify the root causes behind observed performance differences and to generate inferred state machine diagrams. We make this code (and our dataset) publicly available at http://quic.ccs.neu.edu.

Third, we conduct tests using a variety of emulated network conditions, against our own servers and those run by Google, from both desktop and mobile-phone clients, and using multiple historical versions of QUIC. This analysis allows us to understand how QUIC performance evolved over time, and to determine how code changes impact relevant metrics. In doing so, we produce the first state machine diagrams for QUIC based on execution traces.

Our key findings are as follows.

- In the desktop environment, QUIC outperforms TCP+HTTPS in nearly every scenario. This is due to factors that include 0-RTT connection establishment and recovering from loss quickly—properties known to provide performance benefits.
- However, we found QUIC to be sensitive to out-of-order packet delivery. In presence of packet re-ordering, QUIC performs significantly worse than TCP in many scenarios. This occurs because QUIC interprets such behavior as loss, which causes it to send packets more slowly.
- Due to its reliance on application-layer packet processing and encryption, we find that all of QUIC's performance gains are diminished on phones from 2013 and late 2014. It is likely that even older phones will see *worse* performance with QUIC.
- QUIC outperforms TCP in scenarios with fluctuating bandwidth. This is because QUIC's ACK implementation eliminates ACK ambiguity, resulting in more precise RTT and bandwidth estimations.
- We found that when competing with TCP flows, QUIC is unfair to TCP by consuming more than twice its fair share of the bottleneck bandwidth.
- QUIC achieves better quality of experience for video streaming, but only for high-resolution video.
- A TCP proxy can help TCP to shrink the performance gap with QUIC in low latency cases and also under loss. Furthermore, an unoptimized QUIC proxy improves performance under loss for large objects but can hurt performance for small object sizes due to lack of 0-RTT connection establishment.
- QUIC performance has improved since 2016 mainly due to a change from a conservative maximum congestion window to a much larger one.
- We identified a bug affecting the QUIC server included in Chromium version 52 (the stable version at the time of our experiments), where the initial congestion window and Slow Start threshold led to poor performance compared with TCP.

2 BACKGROUND AND RELATED WORK

In this section, we provide background information on QUIC and detail work related to our study.

2.1 Background

Google's Quick UDP Internet Connections (QUIC) protocol is an application-layer transport protocol that is designed to provide high performance, reliable in-order packet delivery, and encryption [10]. The protocol was introduced in 2013, and has undergone rapid development by Google developers. QUIC is included as a separate module in the Chromium source; at the time of our experiments, the latest stable version of Chrome is 60, which supports QUIC versions up to 37. 12 versions of QUIC have been released during our study, i.e., between September 2015 and January 2017.[3]

QUIC motivation. The design of QUIC is motivated largely by two factors. First, experimenting with and deploying new transport layers in the OS is difficult to do quickly and at scale. On the other hand, changing application-layer code can be done relatively easily, particularly when client and server code are controlled by the same entity (e.g., in the case of Google). As such, QUIC is implemented at the application layer to allow Google to more quickly modify and deploy new transport-layer optimizations at scale.

Second, to avoid privacy violations as well as transparent proxying and content modification by middleboxes, QUIC is encrypted end-to-end, protecting not only the application-layer content (e.g., HTTP) but also the transport-layer headers.

QUIC features. QUIC implements several optimizations and features borrowed from existing and proposed TCP, TLS, and HTTP/2 designs. These include:

- *"0-RTT" connection establishment*: Clients that have previously communicated with a server can start a new session without a three-way handshake, using limited state stored at clients and servers. This shaves multiple RTTs from connection establishment, which we demonstrate to be a significant savings for data flows that fit within a small number of packets.
- *Reduced "head of line blocking"*: HTTP/2 allows multiple objects to be fetched over the same connection, using multiple streams within a single flow. If a loss occurs in one stream when using TCP, all streams stall while waiting for packet recovery. In contrast, QUIC allows other streams to continue to exchange packets even if one stream is blocked due to a missing packet.
- *Improved congestion control*: QUIC implements better estimation of connection RTTs and detects and recovers from loss more efficiently.

Other features include forward error correction[4] and improved privacy and flow integrity compared to TCP.

[3] Throughout this paper, unless stated otherwise, we use QUIC version 34, which we found to exhibit identical performance to versions 35 and 36. Changelogs and source code analysis confirm that none of the changes should impact protocol performance.

[4] This feature allows QUIC to recover lost packets without needing retransmissions. Due to poor performance it is currently disabled [37].

				PLT Experiments		Test Environments[5]					
	QUIC Version	Calib-ration[2]	Root Cause Analysis[3]	# of Tested Pages[4]	# of Emulated Scenarios	Net. type	Devices	Fair-ness	Video QoE	Packet Reorder.	Proxy-ing
Megyesi [30]	20*	✗	✗	6	12	F	D	✓	✗	✗	✗
Carlucci[1] [17]	21	✗	✗	3	9	F	D	✗	✗	✗	✗
Biswal [16]	23	✗	✗	20	10	F	D	✗	✗	✗	✗
Das [20]	23*	✗	✗	500†	100 (9)‡	F/C	D	✗	✗	✗	✗
This work	25 to 37	✓	✓	13	18	F/C	D/M	✓	✓	✓	✓

Table 1: Overview of new and extended contributions compared to prior work, i.e., [16, 17, 20, 30]. [1]This work studied impact of FEC, which was removed from QUIC in early 2016. [2]Lack of calibration in prior work led to misleading reports of poor QUIC performance for high-bandwidth links and large pages. [3]Prior work typically speculates on the reasons for observed behavior. [4]Our choice of pages isolate impact of number and size of objects. [5]Mobile (M), desktop (D), fixed-line (F), cellular (C). † Replay of 500 real web pages with no control over size/number of objects to isolate their impact. ‡ Das tested a total of 100 network scenarios, but details of only 9 are mentioned. *Based on specified Chromium version/commit#.

Most relevant to this paper are the congestion and flow control enhancements over TCP, which have received substantial attention from the QUIC development team. QUIC currently[5] uses the Linux TCP Cubic congestion control implementation [35], and adds with several new features. Specifically, QUIC's ACK implementation eliminates ACK ambiguity, which occurs when TCP cannot distinguish losses from out-of-order delivery. It also provides more precise timing information that improves bandwidth and RTT estimates used in the congestion control algorithm. QUIC includes packet pacing to space packet transmissions in a way that reduces bursty packet losses, tail loss probes [22] to reduce the impact of losses at the end of flows, and proportional rate reduction [28] to mitigate the impact of random loss on performance.

Source code. The QUIC source code is open and published as part of the Chromium project [3]. In parallel with deployment, QUIC is moving toward protocol standardization with the publication of multiple Internet drafts [5, 9, 11].

Current deployment. Unlike many other experimental transport-layer protocols, QUIC is widely deployed to clients (Chrome) and already comprises 7% of all Internet traffic [26]. While QUIC can in theory be used to support any higher-layer protocol and be encapsulated in any lower-layer protocol, the only known deployments[6] of QUIC use it for web traffic. Specifically, QUIC is intended as a replacement for the combination of TCP, TLS, and HTTP/2 and runs atop UDP.

Summary. QUIC occupies an interesting place in the space of deployed transport layers. It is used widely at scale with limited repeatable analyses evaluating its performance. It incorporates many experimental and innovative features and rapidly changes the enabled features from one version to the next. While the source code is public, there is limited support for independent configurations and evaluations of QUIC. In this paper, we develop an approach that enables sound evaluations of QUIC, explains the reasons for performance differences with TCP, and supports experimentation with a variety of deployment environments.

[5]Google is developing a new congestion control called BBR [19], which is not yet in general deployment.
[6]These include the Chromium source code and Google services that build on top of it.

2.2 Related Work

Transport-layer performance. There is a large body of previous work on improving transport-layer and web performance, most of it focusing on TCP [21, 22, 28] and HTTP/2 (or SPDY [39]). QUIC builds upon this rich history of transport-layer innovation, but does so entirely at the application-layer. Vernersson [38] uses network emulation to evaluate UDP-based reliable transport, but does not focus specifically on QUIC.

QUIC security analysis. Several recent papers explore the security implications of 0-RTT connection establishment and the QUIC TLS implementation [23, 25, 27], and whether explicit congestion notification can be used with UDP-based protocols such as QUIC [29]. Unlike such studies, we focus entirely on QUIC's end-to-end network performance and do not consider security implications or potential extensions.

Google-reported QUIC performance. The only large-scale performance results for QUIC in production come from Google. This is mainly due to the fact that at the time of writing, Google is the only organization known to have deployed the protocol in production. Google claims that QUIC yields a 3% improvement in mean page load time (PLT) on Google Search when compared to TCP, and that the slowest 1% of connections load one second faster when using QUIC [18]. In addition, in a recent paper [26] Google reported that on average, QUIC reduces Google search latency by 8% and 3.5% for desktop and mobile users respectively, and reduces video rebuffer time by 18% for desktop and 15.3% for mobile users. Google attributes these performance gains to QUIC's lower-latency connection establishment (described below), reduced head-of-line blocking, improved congestion control, and better loss recovery.

In contrast to our work, Google-reported results are aggregated statistics that do not lend themselves to repeatable tests or root cause analysis. *This work takes a complementary approach, using extensive controlled experiments in emulated and operational networks to evaluate Google's performance claims (Sec. 5) and root cause analysis to explain observed performance.*

QUIC emulation results. Closely related to this work, several papers explore QUIC performance. Megyesi et al. [30] use emulated

network tests with desktop clients running QUIC version 20 and Google Sites servers. They find that QUIC runs well in a variety of environments, but HTTP outperforms QUIC in environments with high bandwidth links, high packet loss, and many large objects. Biswal et al. [16] find similar results, except that they report QUIC outperforms HTTP in presence of loss.

Carlucci et al. [17] investigate QUIC performance (in terms of goodput, utilization, and PLT) using QUIC version 21 running on desktop machines with dummy QUIC clients and servers connected through emulated network environments. They find that FEC makes QUIC performance worse, QUIC unfairly consumes more of the bottleneck link capacity than TCP, QUIC underperforms when web pages have multiple objects due to limited numbers of parallel streams, and QUIC performs worse than TCP+HTTP when there are multiple objects with loss. We do not study FEC because it was removed from QUIC in early 2016.

In an M.S. thesis from 2014, Das [20] evaluates QUIC performance using mahimahi [32] to replay 500 real webpages over emulated network conditions. The author found that QUIC performs well only over low-bandwidth, high-RTT links. Das reported that when compared to TCP, QUIC improved performance for small webpages with few objects, but the impact on pages with large numbers of objects was inconclusive. Unlike this work, we focus exclusively on QUIC performance at the transport layer and isolate root causes for observed differences across network environments and workloads. Along with models for complex web page dependencies, our results can inform metrics like page-interactive time for webpage loads (as done in the Mobilyzer study [33]).

Our contributions. This work makes the following new and extended contributions compared to prior work (summarized in Table 1).

- *Recent, properly configured QUIC implementations.* Prior work used old QUIC versions (20–23, compared with 34 in this work), with the default conservative maximum allowed congestion window (MACW). As we discuss in Sec. 4.1, using a small MACW causes poor performance for QUIC, specifically in high-bandwidth environments. This led to misleading reports of poor QUIC performance for high bandwidth links and large pages in prior work. In contrast, we use servers that are tuned to provide nearly identical performance to Google's QUIC servers[7] and demonstrate that QUIC indeed performs well for large web pages and high bandwidth.
- *Isolation of workload impact on performance.* Previous work conflates the impact of different workloads on QUIC performance. For example, when [20] studies the effect of multiplexing, both the number of objects and the page size changes. This conflates QUIC's multiplexing efficiency with object-size efficiency. In contrast, we design our experiments so that they test one workload factor at a time. As a result, we can isolate the impact of parameters such as number and size of objects, or the benefit of 0-RTT connection establishment and proxies.
- *Rigorous statistical analysis.* When comparing QUIC with TCP, most prior work do not determine if observed performance differences are statistically significant. In contrast, we use statistical tests to ensure reported differences are statistically significant; if not, we indicate that performance differences are inconclusive.
- *Root cause analysis.* Prior work typically speculates on the reasons for observed behavior. In contrast, we systematically identify the root causes that explain our findings via experiment isolation, code instrumentation, and state-machine analysis.
- *More extensive test environments.* We consider not only more emulated network environments than most prior work, but we also evaluate QUIC over operational fixed-line and cellular networks. We consider both desktop and mobile clients, and multiple QUIC versions. To the best of our knowledge, we are the first to investigate QUIC with respect to out-of-order packet delivery, variable bandwidth, and video QoE.

[7] This required significant calibration and communication with Google, as described in the following section.

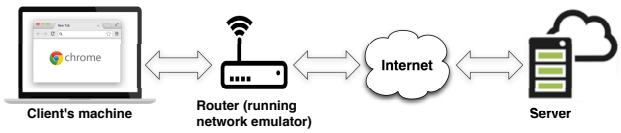


Figure 1: Testbed setup. The server is an EC2 virtual machine running both a QUIC and an Apache server. The empirical RTT from client to server is 12ms and loss is negligible.

3 METHODOLOGY

We now describe our methodology for evaluating QUIC, and comparing it to the combination of HTTP/2, TLS, and TCP. The tools we developed for this work and the data we collected are publicly available.

3.1 Testbed

We conduct our evaluation on a testbed that consists of a device machine running Google's Chrome browser[8] connected to the Internet through a router under our control (Fig. 1). The router runs OpenWRT (Barrier Breaker 14.07, Linux OpenWrt 3.10.49) and includes Linux's Traffic Control [7] and Network Emulation [6] tools, which we use to emulate network conditions including available bandwidth, loss, delay, jitter, and packet reordering.

Our clients consist of a desktop (Ubuntu 14.04, 8 GB memory, Intel Core i5 3.3GHz) and two mobile devices: a Nexus 6 (Android 6.0.1, 3 GB memory, 2.7 GHz quad-core) and a MotoG (Android 4.4.4, 1 GB memory, 1.2 GHz quad-core).

Our servers run on Amazon EC2 (Kernel 4.4.0-34-generic, Ubuntu 14.04, 16 GB memory, 2.4 GHz quad-core) and support HTTP/2 over TCP (using Cubic and the default linux TCP stack configuration) via Apache 2.4 and over QUIC using the standalone QUIC server provided as part of the Chromium source code. To ensure comparable results between protocols, we run our Apache and QUIC servers on the same virtual machine and use the same machine/device as the client. We increase the UDP buffer sizes if necessary to ensure there are no networking bottlenecks caused by the OS. As we discuss in Sec. 4, we configure QUIC so it performs identically to Google's production QUIC servers.

[8] The only browser supporting QUIC at the time of this writing.

Parameter	Values tested
Rate limits (Mbps)	5, 10, 50, 100
Extra Delay (RTT)	0ms, 50ms, 100ms
Extra Loss	0.1%, 1%
Number of objects	1, 2, 5, 10, 100, 200
Object sizes (KB)	5, 10, 100, 200, 500, 1000, 10,000, 210,000
Proxy	QUIC proxy, TCP proxy
Clients	Desktop, Nexus6, MotoG
Video qualities	tiny, medium, hd720, hd2160

Table 2: Parameters used in our tests.

QUIC uses HTTP/2 and encryption on top of its reliable transport implementation. To ensure a fair comparison, we compare QUIC with HTTP/2 over TLS, atop TCP. Throughout this paper we refer to such measurements that include HTTP/2+TLS+TCP as "TCP".

Our servers add all necessary HTTP directives to avoid any caching of data. We also clear the browser cache and close all sockets between experiments to prevent "warmed up" connections from impacting results. However, we do *not* clear the state used for QUIC's 0-RTT connection establishment.

3.2 Network Environments

We compare TCP and QUIC performance across a wide range of network conditions (i.e., various bandwidth limitations, delays, packet losses) and application scenarios (i.e., web page object sizes and number of objects; video streaming). Table 2 shows the scenarios we consider for our tests.

We emulate network conditions on a separate router to avoid erroneous results when doing so on an endpoint. Specifically, we found that when `tc` and `netem` are used at an endpoint directly, they result in undesired behavior such as bursty traffic. Further, if loss is added locally with `tc`, the loss is immediately reported to the transport layer, which can lead to immediate retransmission as if there was no loss—behavior that would not occur outside the emulated environment.

We impose bandwidth caps using token bucket filters (TBF) in `tc`. We conducted a variety of tests to ensure that we did not use settings leading to unreasonably long or short queues or bucket sizes that benefit or harm QUIC or TCP. Specifically, for each test scenario we run experiments to determine whether the network configuration negatively impacts the protocols independent of additional delay or loss, and pick settings that allow the flows to achieve transfer rates that are close to the bandwidth caps.

Our tests on cellular networks use client devices that are directly connected to the Internet.

3.3 Experiments and Performance Metrics

Experiments. Unless otherwise stated, for each evaluation scenario (network conditions, client, and server) we conduct *at least* 10 measurements of each transport protocol (TCP and QUIC). To mitigate any bias from transient noise, we run experiments in 10 rounds or more, each consisting of a download using TCP and one using QUIC, back-to-back. We present the percent differences in performance between TCP and QUIC and indicate whether they are statistically significant ($p < 0.01$). All tests are automated using

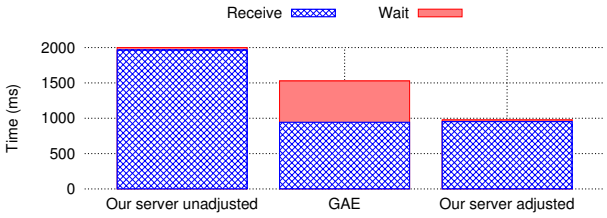


Figure 2: Google App Engine (GAE) vs. our QUIC servers on EC2 before and after configuring them. Loading a 10MB image over a 100Mbps link. The bars show the wait time (red) and download time (blue) after the connection is established and the request has reached the server (averaged over 10 runs). GAE (middle) has a high wait time.

Python scripts and Chrome's debugging tools. We use Android Debug Bridge [1] for automating tests running on mobile phones.

Applications. We test QUIC performance using two applications that currently integrate the protocol: the Chrome browser and YouTube video streaming.

For Chrome, we evaluate QUIC performance using web pages consisting of static HTML that references JPG images (various number and sizes of images) *without* any other object dependencies or scripts. While previous work demonstrates that many factors impact load times and user-perceived performance for typical, popular web pages [4, 33, 39], the focus of this work is only on transport protocol performance. Our choice of simple pages ensures that page load time measurements reflect *only the efficiency of the transport protocol* and not browser-induced factors such as script loading and execution time. Furthermore, our simple web pages are essential for isolating the impact of parameters such as size and number of objects on QUIC multiplexing. We leave investigating the effect of dynamic pages on performance for future work.

In addition, we evaluate video streaming performance for content retrieved from YouTube. Specifically, we use the YouTube iFrame API to collect QoE metrics such as time to start, buffering time, and number of rebuffers.

Performance Metrics. We measure throughput, "page load time" (i.e., the time to download all objects on a page), and video quality metrics that include time to start, rebuffering events, and rebuffering time. For web content, we use Chrome's remote debugging protocol [2] to load a page and then extract HARs [34] that include all resource timings and the protocol used (which allows us to ensure that the correct protocol was used for downloading an object[9]). For video streaming, we use a one-hour YouTube video that is encoded in all quality levels (i.e., from "tiny" to 4K HD).

4 EVALUATION FRAMEWORK

Our testbed provides a platform for running experiments, but does not alone ensure that our comparisons between QUIC and TCP are sound, nor does it explain any performance differences we see. We face two key challenges in addressing this.

First, Google states that the public version of QUIC is not "*performant*" and is for integration testing purposes only [8]. To ensure

[9] Chrome "races" TCP and QUIC connections for the same server and uses the one that establishes a connection first. As such, the protocol used may vary from the intended behavior.

our findings are applicable to Google's deployment environment, we must configure our QUIC servers to match the performance of Google's QUIC servers.

Second, QUIC is instrumented with a copious amount of debugging information but no framework that maps these logs into actionable information that explains performance. While there is a design document, there is no state machine to compare with.

We now describe how we address these challenges in our evaluation framework.

4.1 Calibration

At first glance, a simple approach to experimenting with QUIC as configured by Google would simply be to use Google's servers. While this intuition is appealing, it exhibits two major issues. First, running on Google servers, or any other servers that we do not control, prevents us from instrumenting and altering the protocol to explain why performance varies under different network environments. Second, our experience shows that doing so leads to highly variable results and incorrect conclusions.

For example, consider the case of Google App Engine (GAE), which supports QUIC and allows us to host our own content for testing. While the latency to GAE frontends was low and constant over time, we found a variable wait time between connection establishment and content being served (Fig. 2, middle bar). We do not know the origins for these variable delays, but we suspect that the constant RTT is due to proxying at the frontend, and the variable delay component is due to GAE's shared environment without resource guarantees. The variability was present regardless of time of day, and did not improve when requesting the same content sequentially (thus making it unlikely that the GAE instance was spun down for inactivity). Such variable delay can dominate PLT measurements for small web pages, and cannot reliably be isolated when multiplexing requests.

To avoid these issues, we opted instead to run our own QUIC servers. This raises the question of how to configure QUIC parameters to match those used in deployment. We use a two-phase approach. First, we extract all parameters that are exchanged between client and server (e.g., window sizes) and ensure that our QUIC server uses the same ones observed from Google.

For parameters not exposed by the QUIC server to the client, we use grey-box testing to infer the likely parameters being used. Specifically, we vary server-side parameters until we obtain performance that matches QUIC from Google servers.

The end result is shown in Fig. 2. The left bar shows that QUIC as configured in the public code release takes twice as long to download a large file when compared to the configuration that most closely matches Google's QUIC performance (right bar)[10].

We made two changes to achieve parity with Google's QUIC servers. First, we increased the maximum allowed congestion window size. At the time of our experiments, this value was 107 by default in Chrome. We increased this value to 430, which matched the maximum allowed congestion window in Chromium's development channel. Second, we found and fixed a bug in QUIC that prevented the slow start threshold from being updated using the receiver-advertised buffer size. Failure to do so caused poor performance due to early exit from slow start.[11]

State	Description
Init	Initial connection establishment
Slow Start	Slow start phase
Congestion Avoidance (CA)	Normal congestion avoidance
CA-Maxed	Max allowed win. size is reached
Application Limited	Current cong. win. is not being utilized, hence window will not be increased
Retransmission Timeout	Loss detected due to timeout for ACK
Recovery	Proportional rate reduction fast recovery
Tail Loss Probe [22]	Recover tail losses

Table 3: QUIC states (Cubic CC) and their meanings.

Prior work did no such calibration. This explains why they observed poor QUIC performance in high bandwidth environments or when downloading large web pages [16, 20, 30].

4.2 Instrumentation

While our tests can tell us how QUIC and TCP compare to each other under different circumstances, it is not clear what exactly causes these differences in performance. To shed light on this, we compile QUIC clients and servers from source (using QUIC versions 25 and 34) and instrument them to gain access to the inner workings of the protocol.

QUIC implements TCP-like congestion control. To reason about QUIC's behavior, we instrumented our QUIC server to collect logs that allow us to infer QUIC's state machine from execution traces, and to track congestion window size and packet loss detection. Table 3 lists QUIC's congestion control states.

We use statistics about state transitions and the frequency of visiting each state to understand the root causes behind good or bad performance for QUIC. For example, we found that the reason QUIC's performance suffers in the face of packet re-ordering is that re-ordered packets cause QUIC's loss-detection mechanism to report high numbers of false losses.

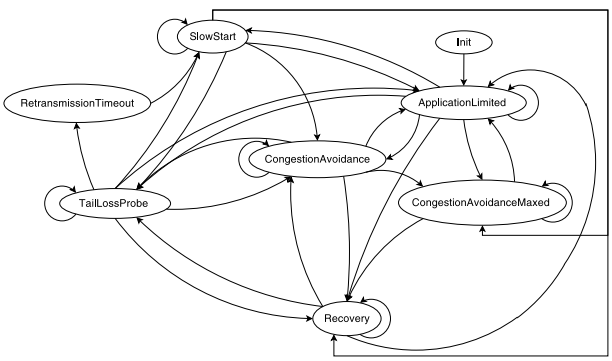
Note that we evaluate QUIC as a whole, in lieu of isolating the impact of protocol components (e.g., congestion avoidance, TLP, *etc.*). We found that disentangling and changing other (non-modular) parts of QUIC (e.g., to change loss recovery techniques, add HOL blocking, change how packets are ACKed) requires rewriting substantial amount of code, and it is not always clear how to replace them. This is an interesting topic to explore in future work.

5 ANALYSIS

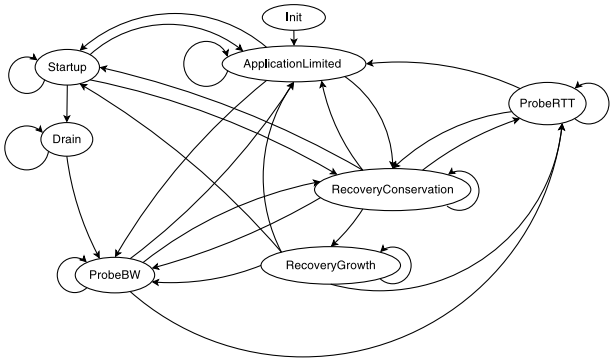
In this section, we conduct extensive measurements and analysis to understand and explain QUIC performance. We begin by focusing on the protocol-layer behavior, QUIC's state machine, and its fairness to TCP. We then evaluate QUIC's application-layer performance, using both page load times (PLT) and video streaming as example application metrics. Finally, we examine the evolution

[10] We focus on PLT because it is the metric we use for end-to-end performance comparisons throughout the paper.

[11] We confirmed our changes with a member of the QUIC team at Google. He also confirmed our bug report.

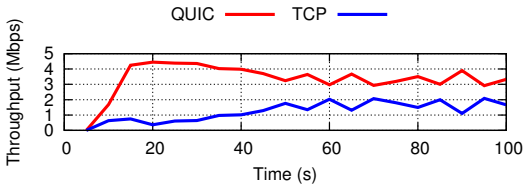


(a) QUIC's Cubic CC

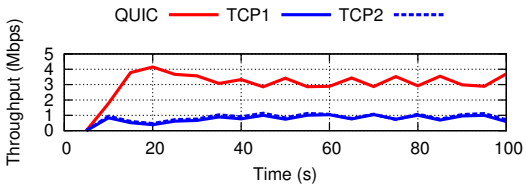


(b) QUIC's BBR CC

Figure 3: State transition diagram for QUIC's CC.



(a) QUIC vs. TCP



(b) QUIC vs. two TCP flows

Figure 4: Timeline showing unfairness between QUIC and TCP when transferring data over the same 5 Mbps bottleneck link (RTT=36ms, buffer=30 KB).

Scenario	Flow	Avg. throughput (std. dev.)
QUIC vs. TCP	QUIC	2.71 (0.46)
	TCP	1.62 (1.27)
QUIC vs. TCPx2	QUIC	2.8 (1.16)
	TCP 1	0.7 (0.21)
	TCP 2	0.96 (0.3)
QUIC vs. TCPx4	QUIC	2.75 (1.2)
	TCP 1	0.45 (0.14)
	TCP 2	0.36 (0.09)
	TCP 3	0.41 (0.11)
	TCP 4	0.45 (0.13)

Table 4: Average throughput (5 Mbps link, buffer=30 KB, averaged over 10 runs) allocated to QUIC and TCP flows when competing with each other. Despite the fact that both protocols use Cubic congestion control, QUIC consumes nearly twice the bottleneck bandwidth than TCP flows combined, resulting in substantial unfairness.

of QUIC's performance and evaluate the performance that QUIC "leaves on the table" by encrypting transport-layer headers that prevent transparent proxying commonly used in cellular (and other high-delay) networks.

5.1 State Machine and Fairness

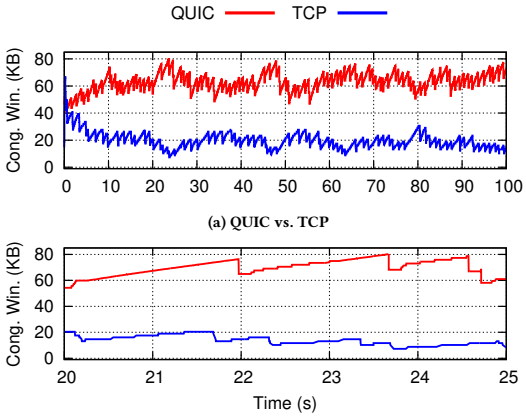
In this section, we analyze high-level properties of the QUIC protocol using our framework.

State machine. QUIC has only a draft formal specification and no state machine diagram or formal model; however, the source code is made publicly available. Absent such a model, we took an empirical approach and used traces of QUIC execution to infer the state machine to better understand the dynamics of QUIC and their impact on performance.

Specifically, we use Synoptic [15] for *automatic* generation of QUIC state machine. While static analysis might generate a more complete state machine, a complete model is not necessary for understanding performance changes. Rather, as we show in Section 5.2, we only need to investigate the states visited and transitions between them at runtime.

Fig. 3a shows the QUIC state machine automatically generated using traces from executing QUIC across all of our experiment configurations. The diagram reveals behaviors that are common to standard TCP implementations, such as connection start (`Init`, `SlowStart`), congestion avoidance (`CongestionAvoidance`), and receiver-limited connections (`ApplicationLimited`). QUIC also includes states that are non-standard, such as a maximum sending rate (`CongestionAvoidanceMaxed`), tail loss probes, and proportional rate reduction during recovery.

Note that capturing the empirical state machine requires instrumenting QUIC's source code with log messages that capture transitions between states. In total, this required adding 23 lines of code in 5 files. While the initial instrumentation required approximately 10 hours, applying the instrumentation to subsequent QUIC versions required only about 30 minutes. To further demonstrate how our approach applies to other congestion control implementations, we instrumented QUIC's experimental BBR implementation and present its state transition diagram in Fig. 3b. This instrumentation took approximately 5 hours. Thus, our experience shows that our approach is able to adapt to evolving protocol versions and implementations with low additional effort.



(a) QUIC vs. TCP

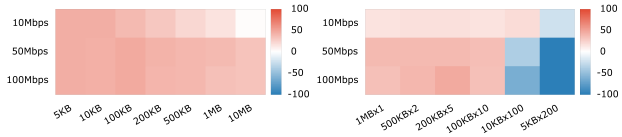
(b) 5-second zoom of above figure

Figure 5: Timeline showing congestion window sizes for QUIC and TCP when transferring data over the same 5 Mbps bottleneck link (RTT=36ms, buffer=30 KB).

We used inferred state machines for root cause analysis of performance issues. In later sections, we demonstrate how they helped us understand QUIC's poor performance on mobile devices and in the presence of deep packet reordering.

Fairness. An essential property of transport-layer protocols is that they do not consume more than their fair share of bottleneck bandwidth resources. Absent this property, an unfair protocol may cause performance degradation for competing flows. We evaluated whether this is the case for the following scenarios, and present aggregate results over 10 runs in Table 4. We expect that QUIC and TCP should be relatively fair to each other because *they both use the Cubic congestion control protocol.* However, we find this is not the case at all.

- **QUIC vs. QUIC.** We find that two QUIC flows are fair to each other. We also found similar behavior for two TCP flows.
- **QUIC vs. TCP.** QUIC multiplexes requests over a single connection, so its designers attempted to set Cubic congestion control parameters so that one QUIC connection emulates N TCP connections (with a default of $N = 2$ in QUIC 34, and $N = 1$ in QUIC 37). We found that N had little impact on fairness. As Fig. 4a shows, QUIC is unfair to TCP as predicted, and consumes approximately twice the bottleneck bandwidth of TCP even with $N = 1$. We repeated these tests using different buffer sizes, including those used by Carlucci et al. [17], but did not observe any significant effect on fairness. This directly contradicts their finding that larger buffer sizes allow TCP and QUIC to fairly share available bandwidth.
- **QUIC vs. multiple TCP connections.** When competing with M TCP connections, one QUIC flow should consume $2/(M + 1)$ of the bottleneck bandwidth. However, as shown in Table 4 and Fig. 4b, QUIC still consumes more than 50% of the bottleneck bandwidth even with 2 and 4 competing TCP flows. Thus, QUIC is not fair to TCP even assuming 2-connection emulation.



(a) Varying object size

(b) Varying object count

Figure 6: QUIC (version 34) vs. TCP with different rate limits for (a) different object sizes and (b) with different numbers of objects. Each heatmap shows the percent difference between QUIC over TCP. Positive numbers—colored red—mean QUIC outperforms TCP and has smaller page-load time. Negative numbers—colored blue—means the opposite. White cells indicate no statistically significant difference.

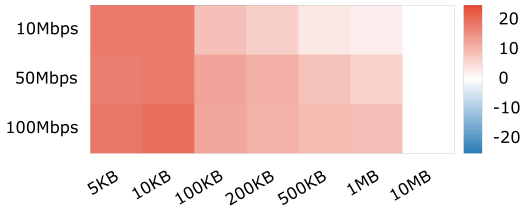


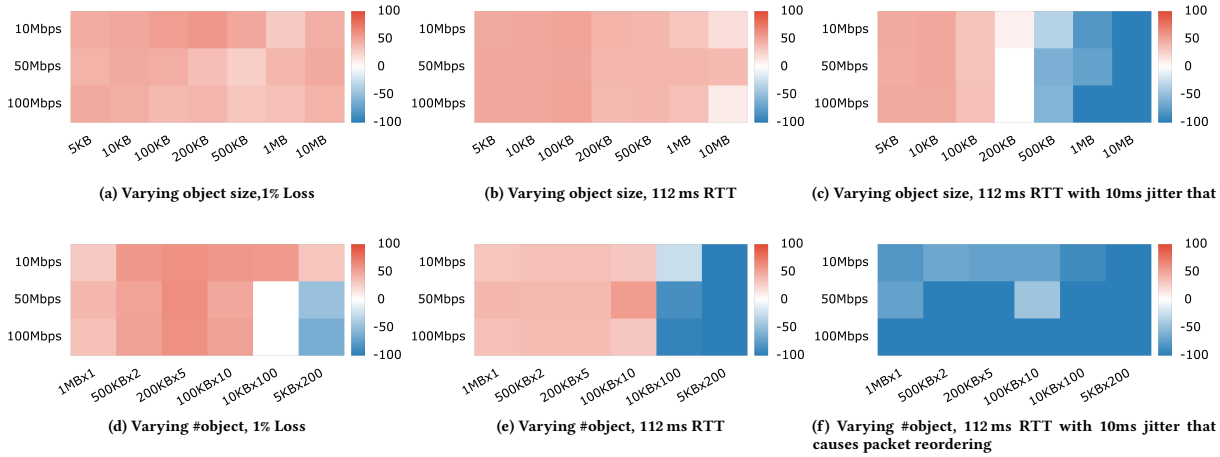
Figure 7: QUIC with and without 0-RTT. Positive numbers—colored red—show the performance gain achieved by 0-RTT. The gain is more significant for small objects, but becomes insignificant as the bandwidth decreases and/or objects become larger, where connection establishment is a tiny fraction of total PLT.

To ensure fairness results were not an artifact of our testbed, we repeated these tests against Google servers. The unfairness results were similar.

We further investigate why QUIC is unfair to TCP by instrumenting the QUIC source code, and using `tcpprobe` [13] for TCP, to extract the congestion window sizes. Fig. 5a shows the congestion window over time for the two protocols. When competing with TCP, QUIC is able to achieve a larger congestion window. Taking a closer look at the congestion window changes (Fig. 5b), we find that while both protocols use Cubic congestion control scheme, QUIC increases its window more aggressively (both in terms of slope, and in terms of more frequent window size increases). As a result, QUIC is able to grab available bandwidth faster than TCP does, leaving TCP unable to acquire its fair share of the bandwidth.

5.2 Page Load Time

This section evaluates QUIC performance compared to TCP for loading web pages (i.e., page load time, or PLT) with different sizes and numbers of objects. Recall from Sec. 3 that we measure PLT using information gathered from Chrome, that we run TCP and QUIC experiments back-to-back, and that we conduct experiments in a variety of emulated network settings. Note that our servers add all necessary HTTP directives to avoid caching content. We also clear the browser cache and close all sockets between experiments to prevent "warmed up" connections from impacting results.



(a) Varying object size,1% Loss (b) Varying object size, 112 ms RTT (c) Varying object size, 112 ms RTT with 10ms jitter that

(d) Varying #object, 1% Loss (e) Varying #object, 112 ms RTT (f) Varying #object, 112 ms RTT with 10ms jitter that causes packet reordering

Figure 8: QUIC v34 vs. TCP at different rate limits, loss, and delay for different object sizes (a, b, and c) and different numbers of objects (d, e, and f).

However, we do *not* clear the state used for QUIC's 0-RTT connection establishment. Furthermore, our PLTs do not include any DNS lookups. This is achieved by extracting resource loading time details from Chrome and excluding the DNS lookups times.

In the results that follow, we evaluate whether the observed performance differences are statistically significant or simply due to noise in the environment. We use the *Welch's t-test* [14], a two-sample location test which is used to test the hypothesis that two populations have equal means. For each scenario, we calculate the p-value according to the 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. Otherwise the difference we observe is not significant and is likely due to noise.

Desktop environment. We begin with the desktop environment and compare QUIC with TCP performance for different rates, object sizes, and object counts—without adding extra delay or loss (RTT = 36ms and loss = 0%). Fig. 6 shows the results as a heatmap, where the color of each cell corresponds to the percent PLT difference between QUIC and TCP for a given bandwidth (vertical dimension) and object size/number (horizontal direction). Red indicates that QUIC is faster (smaller PLT), blue indicates that TCP is faster, and white indicates statistically insignificant differences.

Our key findings are that QUIC outperforms TCP in every scenario except in the case of large numbers of small objects. QUIC's performance gain for smaller object sizes is mainly due to QUIC's 0-RTT connection establishment—substantially reducing delays related to secure connection establishment that corresponds to a substantial portion of total transfer time in these cases. To isolate the impact of 0-RTT, we plotted performance differences between QUIC with and without 0-RTT enabled in Fig. 7. As expected, the benefit is relatively large for small objects and statistically insignificant for 10MB objects.

To investigate the reason why QUIC performs poorly for large numbers of small objects, we explored different values for QUIC's *Maximum Streams Per Connection (MSPC)* parameter to control the level of multiplexing (the default is 100 streams). We found there was no statistically significant impact for doing so, except when setting the MSPC value to a very low number (e.g., 1), which worsens performance substantially.

Instead, we focused on QUIC's congestion control algorithm and identified that in such cases, QUIC's *Hybrid Slow Start* [24] causes early exit from Slow Start due to an increase in the minimum observed RTT by the sender, which Hybrid Slow Start uses as an indication that the path is getting congested. This can hurt the PLT significantly when objects are small and the total transfer time is not long enough for the congestion window to increase to its maximum value. Note that the same issue (early exit from Hybrid Slow Start) affects the scenario with a large number of *large* objects, but QUIC nonetheless outperforms TCP because it has enough time to increase its congestion window and remain at high utilization, thus compensating for exiting Slow Start early.[12]

Desktop with added delay and loss. We repeat the experiments in the previous section, this time adding loss, delay, and jitter. Fig. 8 shows the results, again using heatmaps.

Our key observations are that QUIC outperforms TCP under loss (due to better loss recovery and lack of HOL blocking), and in high-delay environments (due to 0-RTT connection setup). However, in the case of high latency, this is not enough to compensate for QUIC's poor performance for large numbers of small objects. Fig. 9 shows the congestion window over time for the two protocols at 100Mbps and 1% loss. Similar to Fig. 5, under the same network conditions QUIC better recovers from loss events and adjusts its congestion window faster than TCP, resulting in a larger congestion window on average and thus better performance.

Under variable delays, QUIC performs *significantly worse* than TCP. Using our state machine approach, we observed that under variable delay QUIC spends significantly more time in the recovery state compared to relatively stable delay scenarios. To investigate

[12] We leave investigating the reason behind sudden increase in the minimum observed RTT when multiplexing many objects to future work.

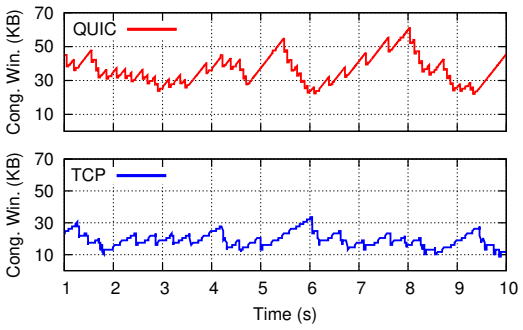


Figure 9: Congestion window over time for QUIC and TCP at 100Mbps rate limit and 1% loss.

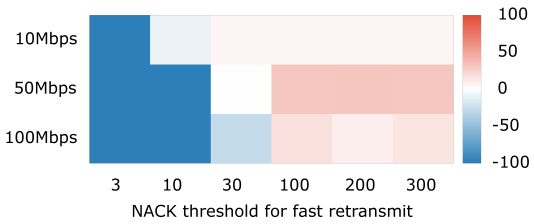


Figure 10: QUIC vs. TCP when downloading a 10MB page (112 ms RTT with 10ms jitter that causes packet reordering). Increasing the NACK threshold for fast retransmit allows QUIC to cope with packet reordering.

this, we instrumented QUIC's loss detection mechanism, and our analysis reveals that variable delays cause QUIC to incorrectly infer packet loss when jitter leads to out-of-order packet delivery. This occurs in our testbed because `netem` adds jitter by assigning a delay to each packet, then queues each packet based on *the adjusted send time*, not the packet arrival time—thus causing packet re-ordering.

The reason that QUIC cannot cope with packet re-ordering is that it uses a fixed threshold for number of NACKs (default 3) before it determines that a packet is lost and responds with a fast retransmit. Packets reordered deeper than this threshold cause false positive loss detection.[13] In contrast, TCP uses the DSACK algorithm [41] to detect packet re-ordering and adapt its NACK threshold accordingly. As we will show later in this section, packet reordering occurs in the cellular networks we tested, so in such cases QUIC will benefit from integrating DSACK. We quantify the impact of using larger DSACK values in Fig. 10, demonstrating that in the presence of packet reordering larger NACK thresholds substantially improve end to end performance compared to smaller NACK thresholds. We shared this result with a QUIC engineer, who subsequently informed us that the QUIC team is experimenting with dynamic threshold and time-based solutions to avoid falsely inferring loss in the presence of reordering.

Desktop with variable bandwidth. The previous tests set a static threshold for the available bandwidth. However, in practice

[13]Note that reordering impact when testing small objects is insignificant because QUIC does not falsely detect losses until a sufficient number of packets are exchanged.

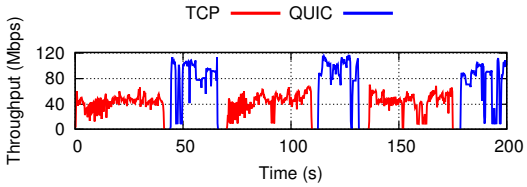


Figure 11: QUIC vs. TCP when downloading a 210MB object. Bandwidth fluctuates between 50 and 150Mbps (randomly picks a rate in that range every one second). Averaging over 10 runs, QUIC is able to achieve an average throughput of 79Mbps (STD=31) while TCP achieves an average throughput of 46Mbps (STD=12).

such values will fluctuate over time, particularly in wireless networks. To investigate how QUIC and TCP compare in environments with variable bandwidth, we configured our testbed to change the bandwidth randomly within specified ranges and with different frequencies.

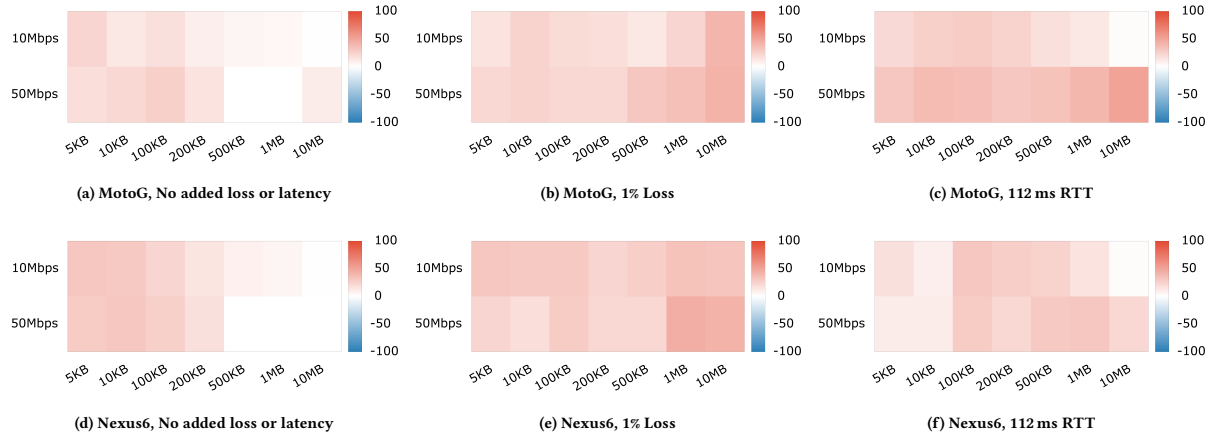
Fig. 11 shows the throughput over time for three back-to-back TCP and QUIC downloads of a 210MB object when the bandwidth randomly fluctuates between 50 and 150Mbps. As shown in this figure, QUIC is more responsive to bandwidth changes and is able to achieve a higher average throughput compared to TCP. We repeated this experiment with different bandwidth ranges and change frequencies and observed the same behavior in all cases.

Mobile environment. Due to QUIC's implementation in userspace (as opposed to TCP's implementation in the OS kernel), resource contention might negatively impact performance independent of the protocol's optimizations for transport efficiency. To test whether this is a concern in practice, we evaluated an increasingly common resource-constrained deployment environment: smartphones. We use the same approach as in the desktop environment, controlling Chrome (with QUIC enabled) over two popular Android phones: the Nexus 6 and the MotoG. These phones are neither top-of-the-line, nor low-end consumer phones, and we expect that they approximate the scenario of a moderately powerful mobile device.

Fig. 12 shows heatmaps for the two devices when varying bandwidth and object size.[14] We find that, similar to the desktop environment, in mobile QUIC outperforms TCP in most cases; however, *its advantages diminish across the board.*

To understand why this is the case, we investigate the QUIC congestion control states visited most in mobile and non-mobile scenarios under the same network conditions. We find that in mobile QUIC spends most of its time (58%) in the "Application Limited" state, which contrasts substantially with the desktop scenario (only 7% of the time). The reason for this behavior is that QUIC runs in a userspace process, whereas TCP runs in the kernel. As a result, QUIC is unable to consume received packets as quickly as on a desktop, leading to suboptimal performance, particularly when there is ample bandwidth available.[15] Fig. 13 shows the full state diagram (based on server logs) in both environments for 50Mbps with no added latency or loss. By revealing the changes in time spent in

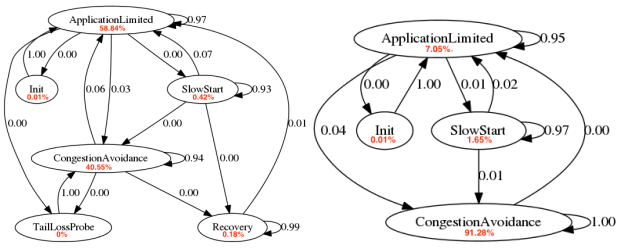
[14]We omit 100 Mbps because our phones cannot achieve rates beyond 50 Mbps over WiFi, and we omit results from varying the number of objects because they are similar to the single-object cases.
[15]A parallel study from Google [26] using aggregate data identifies the same performance issue but does not provide root cause analysis.



(a) MotoG, No added loss or latency (b) MotoG, 1% Loss (c) MotoG, 112 ms RTT

(d) Nexus6, No added loss or latency (e) Nexus6, 1% Loss (f) Nexus6, 112 ms RTT

Figure 12: QUICv34 vs. TCP for varying object sizes on MotoG and Nexus6 smartphones (using WiFi). We find that QUIC's improvements diminish or disappear entirely when running on mobile devices.



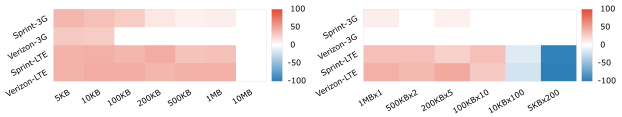
(a) MotoG (b) Desktop

Figure 13: QUIC state transitions on MotoG vs. Desktop. QUICv34, 50 Mbps, no added loss or delay. Red numbers indicate the fraction of time spent in each state, and black numbers indicate the state-transition probability. The figure shows that poor performance for QUIC on mobile devices can be attributed to applications not processing packets quickly enough. Note that the zero transition probabilities are due to rounding down.

each state, such inferred state machines help diagnose problems and develop a better understanding of QUIC dynamics.

Tests on commercial cellular networks. We repeated our PLT tests—without any network emulation—over Sprint's and Verizon's cellular networks, using both 3G and LTE. Table 5 shows the characteristics of these networks at the time of the experiment. To isolate the impact of the network from that of the device they run on, we used our desktop client tethered to a mobile network instead of using a mobile device (because the latter leads to suboptimal performance for QUIC, shown in Fig. 12 and 13). We otherwise keep the same server and client settings as described in Sec. 3.1.

Fig. 14 shows the heatmaps for these tests. For LTE, QUIC performs similarly to a desktop environment with low bandwidth (Fig. 7). In these cell networks, the benefit of 0-RTT is larger for the 1MB page due to higher latencies in the cellular environment.



(a) Varying object size (b) Varying object count

Figure 14: QUICv34 vs. TCP over Verizon and Sprint cellular networks.

	Thrghpt. (Mbps)		RTT (STD) (ms)		Reordering (%)		Loss (%)	
	3G	LTE	3G	LTE	3G	LTE	3G	LTE
Verizon	0.17	4.0	109 (20)	62 (14)	9	0.25	0.05	0
Sprint	0.31	2.4	70 (39)	55 (11)	1.38	0.13	0.02	0.02

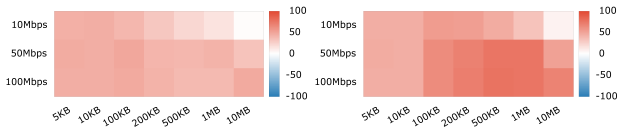
Table 5: Characteristics of tested cell networks. Throughput and RTT represent averages.

In the case of 3G, we see the benefits of QUIC diminish. Compared to LTE, the bandwidth in 3G is much lower and the loss is higher—which works to QUIC's benefit (see Fig. 8a). However, the packet reordering rates are higher compared to LTE, and this works to QUIC's disadvantage. Note that in 3G scenarios, in many cases QUIC had better performance on average (i.e., lower average PLT); however, the high variance resulted in high p-values, which means we cannot reject the null hypothesis that the two sample sets were drawn form the same (noisy) distribution.

5.3 Video-streaming Performance

This section investigates QUIC's impact on video streaming in the desktop environment. Unlike page load times, which tend to be limited by RTT and multiplexing, video streaming relies on the transport layer to load large objects sufficiently quickly to maintain smooth playback. This exercises a transport-layer's ability to quickly ramp up and maintain high utilization.

We test video-streaming performance using YouTube, which supports both QUIC and TCP. We evaluate the protocols using well known QoE metrics for video such as the time spent waiting for



(a) MACW=430 (b) MACW=2000

Figure 15: QUIC (version 37) vs. TCP with different maximum allowable congestion window (MACW) size. In (a), MACW=430 and QUIC versions 34 and 37 have identical performance (see Fig. 6a), In (b), we use the default MACW=2000 for QUIC 37, which results in higher throughput and larger performance gains for large transfers in high bandwidth networks.

initial playback, and the number of rebuffering events. For the latter metric, Google reports that, on average users experience 18% fewer re-buffers when watching YouTube videos over QUIC [26].

We developed a tool for automatically streaming a YouTube video and logging quality of experience (QoE) metrics via the API mentioned in Sec. 3.3. The tool opens a one-hour-long YouTube video, selects a specific quality level, lets the video run for 60 seconds, and logs the following QoE metrics: time to start the video, video quality, quality changes, re-buffering events, and fraction of video loaded. As we demonstrated in previous work [31], 60 seconds is sufficient to capture QoE differences. We use this tool to stream videos using QUIC and TCP and compare the QoE for the two protocols.

Table 6 shows the results for 100 Mbps bandwidth and 1% loss,[16] a condition under which QUIC outperforms TCP (Sec. 8). In this environment, at low and medium resolutions we see no significant difference in QoE metrics, but for the highest quality, `hd2160`, QUIC is able to load a larger fraction of the video in 60 seconds and experience fewer rebuffers per time played, which is consistent with our PLT test results (Sec. 5.2) and with what Google reported. Thus, to refine their observations, we find that QUIC can outperform TCP for video streaming, but this matters only for high resolutions.

5.4 Historical Comparison

To understand how QUIC performance has changed over time, we evaluated 10 QUIC versions (25 to 34)[17] in our testbed. In order to only capture differences due to QUIC version changes, and not due to different configuration values, we used the same version of Chrome and the same QUIC parameter configuration when testing different QUIC versions.

We found that when using the same configuration most QUIC versions in this range yielded nearly identical results, despite substantial changes to the QUIC codebase (including reorganization of code that frustrated our attempts to instrument it). This is corroborated by changelogs [12] that indicate most modifications were to the cryptography logic, QUIC flags, and connection IDs.

Based on the relatively stable QUIC performance across recent versions, we expect that our observations about its performance using its current congestion control algorithm are likely to hold in the future (except in places where we identified opportunities to improve the protocol).

Note that at the time of writing, the recently proposed BBR congestion control algorithm has not been deployed in the "stable" branch and thus we could not evaluate its performance fairly against Cubic in QUIC or TCP. Private communication with a QUIC team member indicated that BBR is "not yet performing as well as Cubic in our deployment tests."

[16] We observed similar results for 10 and 50 Mbps under similar loss.

[17] The stable version of Chrome at the time of analysis (52) did not support QUIC versions earlier than 25. To avoid false inferences from using different Chrome versions *and* different QUIC versions, we only tested QUIC versions 25 and higher.

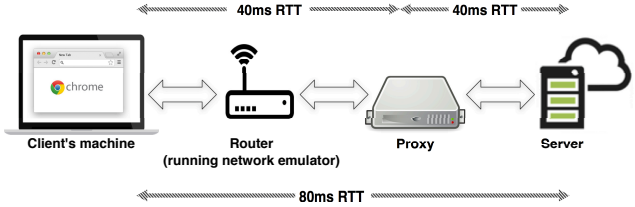


Figure 16: QUIC proxy test setup. The proxy is located midway between client and server.

Comparison with QUIC 37. At the time of publication, the latest stable version of Chromium was 60.0.3112.101, which includes QUIC 37 as the latest stable version. To enhance our longitudinal analysis and demonstrate how our approach easily adapts to new versions of QUIC, we instrumented, tested, and compared QUIC 37 with 34 (the one used for experiments through out this paper).

We found that the main change in QUIC 37 is that the maximum allowed congestion window (MACW) increased to 2000 (from 430 used in our experiments) in the new versions of Chromium. This allows QUIC to achieve much higher throughput compared to version 34, particularly improving performance when compared with TCP for large transfers in high bandwidth networks. Fig. 15 shows the comparison between TCP and QUIC version 37 for various object sizes with MACW of 430 (Fig. 15a) and 2000 (Fig. 15b). When comparing Fig. 15a and 6a, we find that QUIC versions 34 and 37 have almost identical performance when using the same MACW; this is corroborated by QUIC version changelogs [12]. All our previous findings, e.g., QUIC performance degradation in presence of deep packet reordering, still hold for this new version of QUIC.

5.5 Impact of Proxying

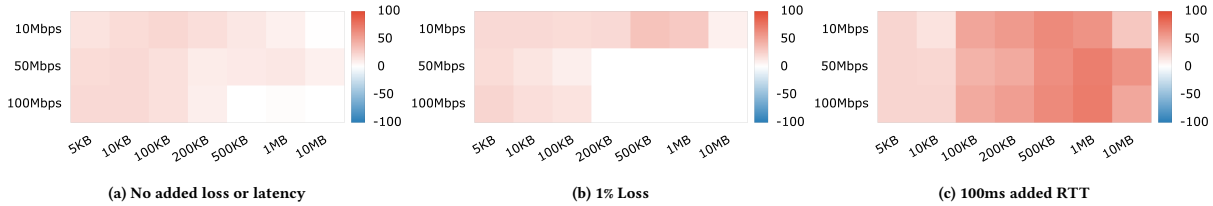
We now test the impact of QUIC's design decisions on in-network performance optimization. Specifically, many high-latency networks use transparent TCP proxies to reduce end-to-end delays and improve loss recovery [40]. However, due to the fact that QUIC encrypts not only payloads but also transport headers, such proxying is impossible for in-network devices.

We evaluate the extent to which this decision impacts QUIC's potential performance gains. Specifically, we wrote a QUIC proxy, and co-located it with a TCP proxy so that we could compare the impact of proxying on end-to-end performance (Fig. 16). For these experiments, we consider PLTs as done in previous sections.

We present two types of comparison results: QUIC vs. proxied TCP (as this is what one would expect to find in many cellular networks), and QUIC vs. proxied QUIC (to determine how QUIC would benefit if proxies could transparently terminate QUIC connections). For the former case, we find that QUIC continues to outperform

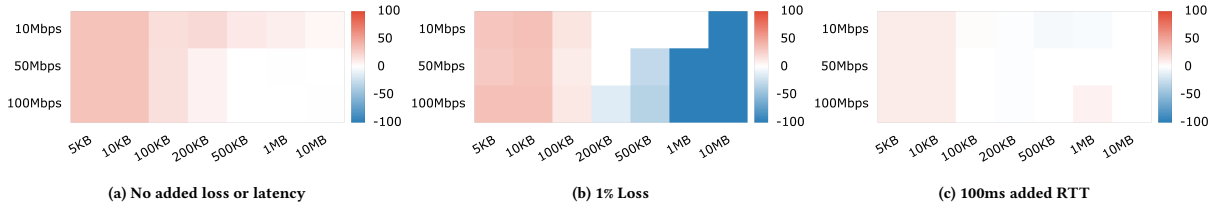
Quality	Time to Start (secs)		video loaded in 1 min (%)		Buffer/Play Time† (%)		#rebuffers		#rebuffers per playing secs	
	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP	QUIC	TCP
tiny	0.5 (0.11)	0.5 (0.21)	33.8 (0.01)	33.8 (0.01)	0.9 (0.17)	0.9 (0.34)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
medium	0.9 (1.04)	0.5 (0.13)	17.9 (0.01)	12.9 (0.92)	1.4 (1.62)	0.9 (0.22)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hd720	0.7 (0.16)	0.7 (0.18)	8.0 (0.27)	4.3 (0.28)	1.1 (0.27)	1.1 (0.27)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)
hd2160	5.9 (2.73)	5.9 (2.51)	0.8 (0.05)	0.4 (0.01)	50.2 (3.01)	73.1 (1.91)	6.7 (0.46)	4.9 (0.3)	0.2 (0.01)	0.3 (0.01)

Table 6: Mean (std) of QoE metrics for a YouTube video in different qualities, averaged over 10 runs. 100Mbps, 1% loss. QUIC benefits are clear for high qualities. While the absolute number of rebuffers for QUIC is higher for hd2160, it is able to load and play more of the video in a given time compared to TCP, with fewer (about 30%) rebuffers per playing second. †Buffer/Play Time is the time spent while buffering the video divided by the time playing video.



(a) No added loss or latency (b) 1% Loss (c) 100ms added RTT

Figure 17: QUIC vs. TCP proxied, where red cells indicate that QUIC performs better.



(a) No added loss or latency (b) 1% Loss (c) 100ms added RTT

Figure 18: QUIC with and without proxy when downloading objects with different sizes. Positive numbers (red cells) mean QUIC performs better connecting to the server directly.

TCP in many scenarios, but its benefits diminish or entirely disappear compared to unproxied TCP in low loss/latency cases, and when there is 1% loss. In the case of high delay links, QUIC still outperforms TCP (Fig. 17). Thus, proxies can help TCP to recover many of the benefits of QUIC, but primarily in lossy scenarios, and when the proxy is equidistant from the client and server.

In the case of implementing a QUIC proxy (Fig. 18), we find that a proxy hurts performance for small object sizes (likely due to inefficiencies and the inability to establish connections via 0-RTT), but performance is better under loss for large objects. Taken together, our initial attempt at a QUIC proxy provides mixed results, and identifying any other potential benefits will require additional tuning of the proxy code.

6 CONCLUSION

In this paper, we address the problem of evaluating an application-layer transport protocol that was built without a formal specification, is rapidly evolving, and is deployed at scale with nonpublic configuration parameters. To do so, we use a methodology and testbed that allows us to conduct controlled experiments in a variety of network conditions, instrument the protocol to reason about its performance, and ensure that our evaluations use settings that approximate those deployed in the wild. We used this approach to evaluate QUIC, and found cases where it performs well and poorly—both in traditional desktop environments but also in mobile and proxy scenarios not previously tested. With the help of an inferred protocol state machine and information about time spent in each state, we explained the performance results we observed.

There are a number of open questions we plan to address in future work. First, we will evaluate performance in additional operational networks, particularly in more mobile ones and data centers. Second, we will investigate techniques to improve QUIC's fairness to TCP while still maintaining high utilization. Third, we will automate the steps used for analysis in our approach and port it to other application layer protocols. This includes adapting our state-machine inference approach to other protocols, and we encourage developers to annotate state transitions in their code to facilitate such analysis. We believe doing so can lead to a more performant, reliable evolution for such network protocols.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers and our shepherd Costin Raiciu for their valuable feedback. Jana Iyengar provided comments on early versions of this work. This work is funded in part by NSF grants CNS-1600266, CNS-1617728.

REFERENCES

[1] Android debug bridge. https://developer.android.com/studio/command-line/adb.html.
[2] Chrome debugging protocol. https://developer.chrome.com/devtools/docs/debugger-protocol.
[3] Chromium. https://www.chromium.org/Home.
[4] I. grigorik. deciphering the critical rendering path. https://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/.
[5] IETF QUIC WG. https://github.com/quicwg.
[6] Linux network emulation. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.
[7] Linux traffic control. http://linux.die.net/man/8/tc.
[8] Playing with QUIC. https://www.chromium.org/quic/playing-with-quic.
[9] QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2. https://tools.ietf.org/html/draft-tsvwg-quic-protocol-02.
[10] QUIC at 10,000 feet. https://docs.google.com/document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU.
[11] QUIC Loss Recovery And Congestion Control. https://tools.ietf.org/html/draft-tsvwg-quic-loss-recovery-01.
[12] QUIC Wire Layout Specification. https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U.
[13] Tcp probe. https://wiki.linuxfoundation.org/networking/tcpprobe.
[14] Welch's t-test. https://en.wikipedia.org/wiki/Welch%27s_t-test.
[15] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging Existing Instrumentation to Automatically Infer Invariant-constrained Models. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.
[16] P. Biswal and O. Gnawali. Does quic make the web faster? In *IEEE GLOBECOM*, 2016.
[17] G. Carlucci, L. De Cicco, and S. Mascolo. HTTP over UDP: an experimental investigation of QUIC. In *Proc. of SAC*, 2015.
[18] Chromium Blog. A QUIC update on Google's experimental transport. http://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html, April 2015.
[19] C. Cimpanu. Google Creates New Algorithm for Handling TCP Traffic Congestion Control. http://news.softpedia.com/news/google-creates-new-algorithm-for-handling-tcp-traffic-congestion-control-508398.shtml, September 2016.
[20] S. R. Das. Evaluation of QUIC on web page performance. Master's thesis, Massachusetts Institute of Technology, 2014.
[21] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira. PCC: Re-architecting congestion control for consistent high performance. In *Proc. of USENIX NSDI*, 2015.
[22] N. Dukkipati, N. Cardwell, Y. Cheng, and M. Mathis. Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses. https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01, February 2013.
[23] M. Fischlin and F. Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In *Proc. of ACM CCS*, 2014.
[24] S. Ha and I. Rhee. Taming the elephants: New tcp slow start. In *Comput. Netw.*, 2011.
[25] T. Jager, J. Schwenk, and J. Somorovsky. On the security of TLS 1.3 and QUIC against weaknesses in PKCS# 1 v1. 5 encryption. In *Proc. of ACM CCS*, 2015.
[26] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Kulik, J. Roskind, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The QUIC transport protocol: Design and Internet-scale deployment. In *Proc. of ACM SIGCOMM*, 2017.
[27] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru. How secure and quick is QUIC? provable security and performance analyses. In *Proc. of IEEE Security and Privacy*, 2015.
[28] M. Mathis, N. Dukkipati, and Y. Cheng. Proportional rate reduction for TCP. https://tools.ietf.org/html/rfc6937, May 2013.
[29] S. McQuistin and C. S. Perkins. Is explicit congestion notification usable with udp? In *Proc. of IMC*, 2015.
[30] P. Megyesi, Z. Krämer, and S. Molnár. How quick is QUIC? In *Proc. of ICC*, May 2016.
[31] A. Molavi Kakhki, F. Li, D. Choffnes, A. Mislove, and E. Katz-Bassett. BingeOn under the microscope: Understanding T-Mobile's zero-rating implementation. In *ACM SIGCOMM Internet-QoE Workshop*, Aug. 2016.
[32] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-replay for HTTP. In *Proc. of USENIX ATC*, 2015.
[33] A. Nikravesh, H. Yao, S. Xu, D. R. Choffnes, and Z. M. Mao. Mobilyzer: An open platform for controllable mobile network measurements. In *Proc. of MobiSys*, 2015.
[34] J. Odvarko, A. Jain, and A. Davies. HTTP Archive (HAR) format. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/HAR/Overview.html, August 2012.
[35] I. Swett. QUIC congestion control and loss recovery. https://docs.google.com/presentation/d/1T9GtMz1CvPpZtmF8g-W7j9XHZBOCp9cu1fW0sMsmpoo.
[36] I. Swett. QUIC Deployment Experience @Google. https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf, 2016.
[37] I. Swett. QUIC FEC v1. https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk/edit, February 2016.
[38] A. Vernersson. Analysis of UDP-based reliable transport using network emulation. Master's thesis, Luleå University of Technology, 2015.
[39] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is SPDY? In *Proc. of USENIX NSDI*, 2014.
[40] X. Xu, Y. Jiang, T. Flach, E. Katz-Bassett, D. R. Choffnes, and R. Govindan. Investigating transparent web proxies in cellular networks. In *Proc. PAM*, 2015.
[41] Zhang, Ming and Karp, Brad and Floyd, Sally and Peterson, Larry. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proceedings of the 11th IEEE International Conference on Network Protocols*, ICNP '03, Washington, DC, USA, 2003. IEEE Computer Society.