DOI:10.1145/3330336

Taking a Long Look at QUIC

An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols

By Arash Molavi Kakhki,* Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove

Abstract

Google's Quick UDP Internet Connections (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.

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 Operating System (S) 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 Quick UDP Internet Connections (QUIC) protocol, initially released by Google in 2013,[4] 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).[a] 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.[21 b] In fact, given the popularity of Google services (including search and video), QUIC now represents a substantial fraction (estimated at 7%[15]) 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,[9] there have been very few repeatable studies measuring and explaining the performance of QUIC compared with standard HTTP/2+TCP.[8, 11, 17] In addition to Google's QUIC, an IETF working group established in 2016 is working on standardizing QUIC and there are more than 20 QUIC implementations in progress.[2] Our study focuses on Google's QUIC implementation.

Our overarching goal is to understand the benefits and tradeoffs that QUIC provides. In this work, we address a number of 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

[a] It also implements TLS and SPDY, as described in the next section.

[b] Newer versions of QUIC running on servers are incompatible with older clients, and ISPs sometimes block QUIC as an unknown protocol. In such cases, Chrome falls back to TCP.

The original version of this paper was published in the *Proceedings of the 2017 ACM Internet Measure Conference* (London, U.K., Nov. 1–3, 2017).

* Work done while at Northeastern University.

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.

Summary of findings. Our key findings covered in this article 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.
- QUIC is sensitive to out-of-order packet delivery. QUIC interprets such behavior as loss and performs significantly worse than TCP in many scenarios.
- QUIC's performance gains are diminished on phones due to its reliance on application-layer packet processing and encryption.
- QUIC outperforms TCP in scenarios with fluctuating bandwidth. This is because QUIC's Acknowledgment (ACK) implementation eliminates ACK ambiguity, resulting in more precise RTT and bandwidth estimations.
- When competing with TCP flows, QUIC is unfair to TCP by consuming more than twice its fair share of the bottleneck bandwidth.
- 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

Google's QUIC protocol is an application-layer transport protocol that is designed to provide high performance, reliable in-order packet delivery, and encryption.[4] 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, that is, between September 2015 and January 2017.[c]

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.
- *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 Round-trip Time (RTTs) and detects and recovers from loss more efficiently.

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

Most relevant to this work are the congestion and flow control enhancements over TCP, which have received substantial attention from the QUIC development team. QUIC currently[e] uses the Linux TCP Cubic congestion control implementation,[20] 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[12] to reduce the impact of losses at the end of flows, and proportional rate reduction[16] to mitigate the impact of random loss on performance.

2.1. Related work

QUIC emulation results. Several papers explore QUIC performance and compare it with TCP.[8, 11, 17] However, prior work have a number of shortcomings including lack of proper configuration of QUIC, limited test environments,

[c] 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.

[d] This feature allows QUIC to recover lost packets without needing retransmissions. Due to poor performance it is currently disabled.[22]

[e] Google is developing a new congestion control called BBR,[10] which is not yet in general deployment.

and absence of root cause analysis for reported observations. We refer the reader to out full paper[14] for detailed discussion on these works.

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.[9] In addition, in a recent paper[15] 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 (Section 4) and root cause analysis to explain observed performance*.

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.

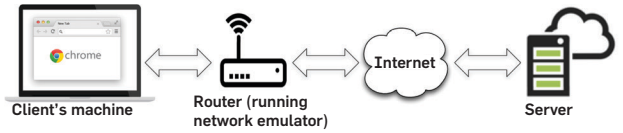
3.1. Testbed

We conduct our evaluation on a testbed that consists of a device machine running Google's Chrome browser[f] connected to the Internet through a router under our control (Figure 1). The router runs OpenWRT (Barrier Breaker 14.07, Linux OpenWrt 3.10.49) and includes Linux's Traffic Control and Network Emulation 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, 8GB memory, Intel Core i5 3.3GHz) and two mobile devices: a Nexus 6 (Android 6.0.1, 3GB memory, 2.7GHz quad-core) and a MotoG (Android 4.4.4, 1GB memory, 1.2GHz quadcore).

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

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



Our servers run on Amazon EC2 (Kernel 4.4.0-34-generic, Ubuntu 14.04, 16GB memory, 2.4GHz 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 Section 4.1, we configure QUIC so it performs identically to Google's production QUIC servers.

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. 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. All tests are automated using Python scripts and Chrome's debugging tools. We use Android Debug Bridge for automating tests running on mobile phones.

Application. We test QUIC performance using the Chrome browser that currently integrates the protocol.

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,[3, 18, 23] the focus of this work is only on transport protocol performance. Our choice of simple pages ensures that PLT 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.

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

[g] 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.

Chrome's remote debugging protocol[1] to load a page and then extract HARs[19] that include all resource timings and the protocol used (which allows us to ensure that the correct protocol was used for downloading an object[g]).

4. 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 PLT as example application metric.

4.1. Calibration and instrumentation

In order to guarantee that our evaluation framework result in sound comparisons between QUIC and TCP, and be able to explain any performance differences we see, we (a) carefully configured our QUIC servers to match the performance of Google's production QUIC server and (b) compiled QUIC client and server from source and instrumented them to gain access to the inner workings of the protocol (e.g., congestion control states and window sizes). Prior work did no such calibration and/or instrumentation, which explains their reported poor QUIC performance in some scenarios, and lack of root cause analysis. We refer the reader to our full paper[14] for detailed discussion on our calibration and instrumentation.

4.2. 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[7] 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 4.3, we only need to investigate the states visited and transitions between them at runtime.

Figure 2a 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 five 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 our full paper.[14] 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.

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 1. 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.

Figure 2. State transition diagram for QUIC's Cubic CC.

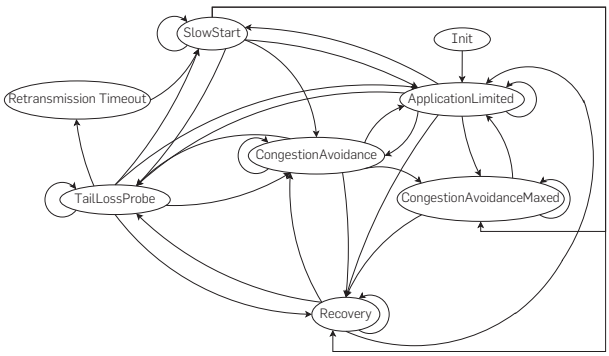


Table 1. Average throughput (5Mbps link, buffer = 30KB, averaged over 10 runs) allocated to QUIC and TCP flows when competing with each other. Despite the fact that both protocols use Cubic congestion control, QUIC consumes nearly twice the bottleneck bandwidth than TCP flows combined, resulting in substantial unfairness.

Scenario	Flow	Avg. throughput (std. dev.)
QUIC vs. TCP	QUIC	2.71 (0.46)
	TCP	1.62 (1.27)
QUIC vs. TCP×2	QUIC	2.8 (1.16)
	TCP 1	0.7 (0.21)
	TCP 2	0.96 (0.3)

- **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 Figure 3a 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.,[8] 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 $N/(M + 1)$ of the bottleneck bandwidth. However, as shown in Table 1, QUIC still consumes more than 50% of the bottleneck bandwidth even with two competing TCP flows. Thus, QUIC is not fair to TCP even assuming two-connection emulation.

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`[5] for TCP, to extract the congestion window sizes. Figure 4a 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 (Figure 4b), 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.

Figure 3. Timeline showing unfairness between QUIC and TCP when transferring data over the same 5Mbps bottleneck link (RTT = 36ms, buffer = 30KB).

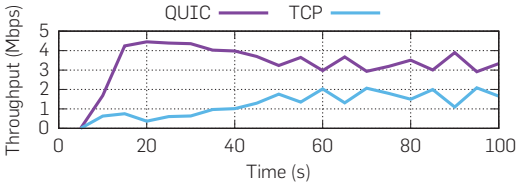
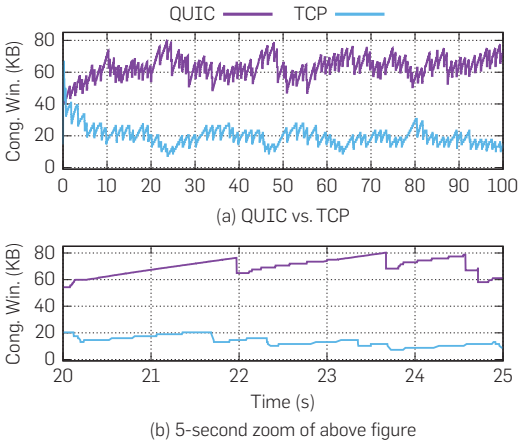


Figure 4. Timeline showing congestion window sizes for QUIC and TCP when transferring data over the same 5Mbps bottleneck link (RTT = 36ms, buffer = 30KB).



4.3. 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 Section 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. 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.

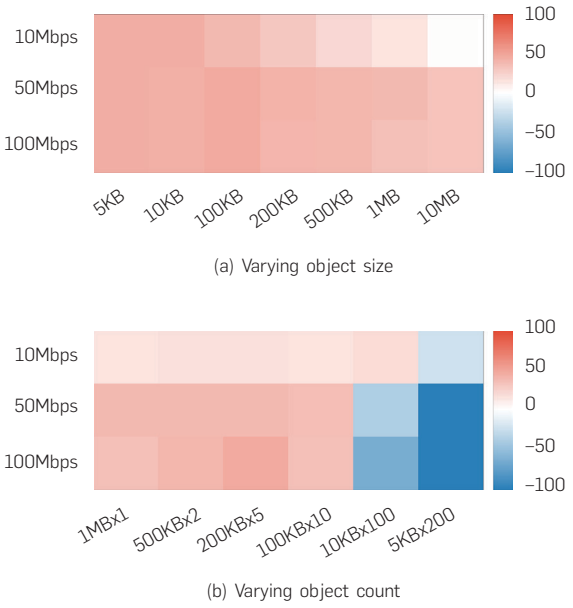
In the results that follow, we evaluate whether the observed performance differences are statistically significant or simply due to noise in the environment. We use the *Welch's t-test*,[6] a two-sample location test which is used to test the hypothesis that two populations have equal means. For each 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%). Figure 5 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 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., one), which worsens performance substantially.

Figure 5. 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—mean the opposite. White cells indicate no statistically significant difference.



Instead, we focused on QUIC's congestion control algorithm and identified that in such cases, QUIC's *Hybrid Slow Start*[13] 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.[h]

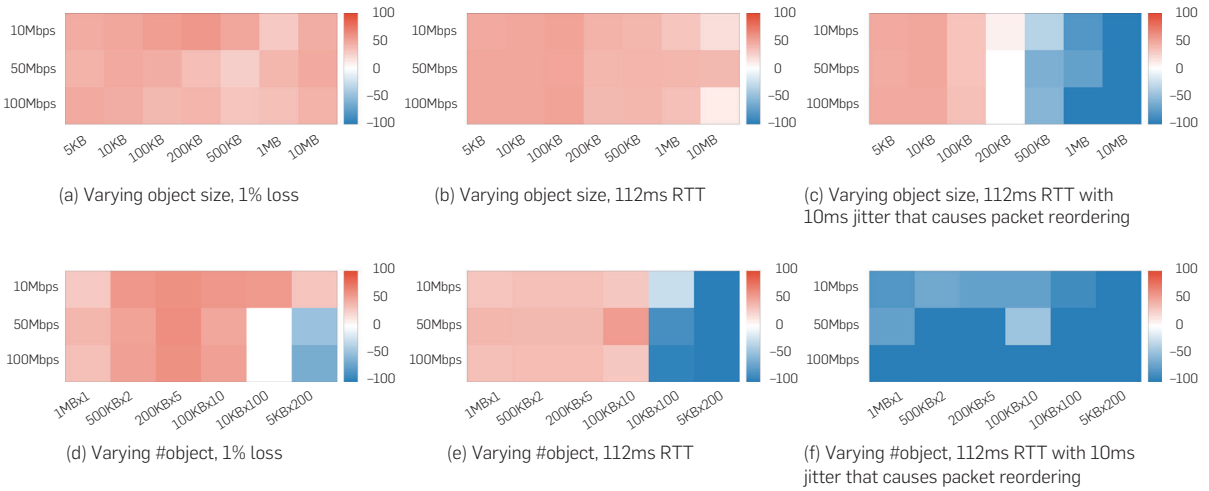
Desktop with added delay and loss. We repeat the experiments in the previous section, this time adding loss, delay, and jitter. Figure 6 shows the results, again using heatmaps.

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. Figure 7 shows the congestion window over time for the two protocols at 100Mbps and 1% loss. Similar to Figure 4, 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

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

Figure 6. 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).



relatively stable delay scenarios. To investigate 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`, which we use for network emulation, 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 reordering.

The reason that QUIC cannot cope with packet reordering is that it uses a fixed threshold for number of Negative Acknowledgment (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.[i] In contrast, TCP uses the Duplicate Selective Acknowledgment (DSACK) algorithm[24] to detect packet reordering 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 Figure 8, 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 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.

Figure 9 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.

Figure 10 shows heatmaps for the two devices when varying bandwidth and object size.[j] 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.

[i] 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 7. Congestion window over time for QUIC and TCP at 100Mbps rate limit and 1% loss.

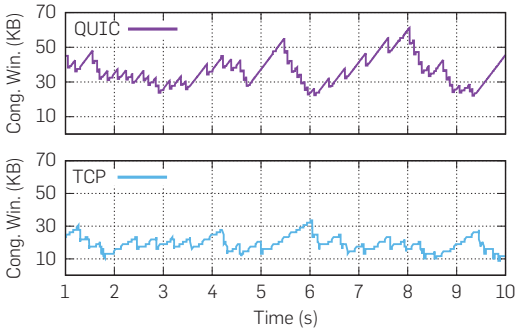
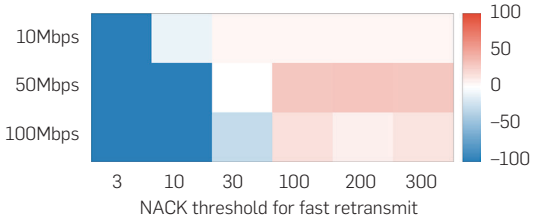


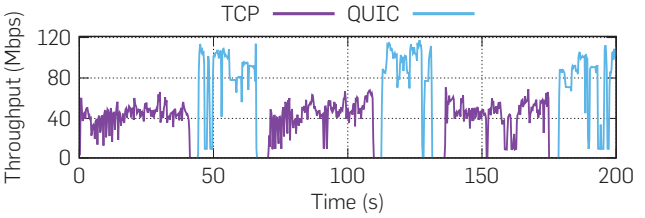
Figure 8. QUIC vs. TCP when downloading a 10MB page (112ms RTT with 10ms jitter that causes packet reordering). Increasing the NACK threshold for fast retransmit allows QUIC to cope with packet reordering.



[j] We omit 100Mbps because our phones cannot achieve rates beyond 50Mbps over WiFi, and we omit results from varying the number of objects because they are similar to the single-object cases.

[k] A parallel study from Google[15] using aggregate data identifies the same performance issue but does not provide root cause analysis.

Figure 9. 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. dev. = 31) while TCP achieves an average throughput of 46Mbps (std. dev. = 12).



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 user-space 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.[k] Table 2 shows the fraction of time (based on server logs) QUIC spent in each state in both environments for 50Mbps with no added latency or loss. By revealing the changes in time spent in each state, such inferred state machines help diagnose problems and develop a better understanding of QUIC dynamics.

5. CONCLUDING DISCUSSION

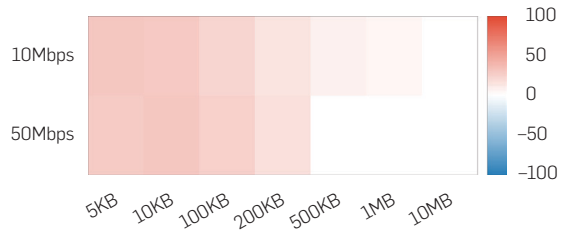
In this paper, we address the problem of evaluating an application-layer transport protocol that was built without a formal specification, is rapidly evolving, and is deployed at scale with nonpublic configuration parameters. To do so, we use a methodology and testbed that allows us to conduct controlled 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 and mobile environments. With the help of an inferred protocol state machine and information about time spent in each state, we explained the performance results we observed.

Additionally, we performed a number of other experiments that were ommited from this paper due to space limitations. These included testing QUIC's performance for video streaming, tests in operational mobile networks, and impact of proxying. For more information on these experiments and our findings, we refer the reader to our full paper.[14]

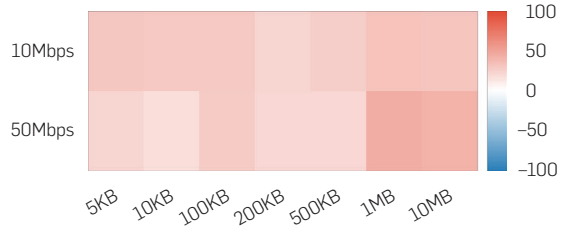
Lessons learned. During our evaluation of QUIC, we identified several key challenges for repeatable, rigorous analyses of application-layer transport protocols in general. Below we list a number of lessons learned while addressing these challenges:

- *Proper empirical evaluation is easy to get wrong*: A successful protocol evaluation and analysis requires proper configuration, calibration, workload isolation, coverage of a wide array of test environments, rigorous statistical analysis, and root cause analysis. While this may seem obvious to the seasoned empiricist, it took us much effort and many attempts to get them right, so we leave these lessons as reminders for a general audience.
- *Models are essential for explaining performance*: Transport protocol dynamics are complex and difficult to summarize via traditional logging. We found that building an inferred state machine model and using transitions between states helped tame this complexity and offer insight into root causes for protocol performance.
- *Plan for change. As the Internet evolves, so too will transport protocols*: It is essential to develop evaluation techniques that adapt easily to such changes to provide consistent and fair comparisons over time.
- *Do not forget to look at the big picture*: It's easy to get caught up in head-to-head comparisons between a flow from one protocol versus another. However, in the wide area there may be thousands or more flows competing over the same bottleneck link. In our limited fairness study, we found that protocol differences in isolation are magnified at scale. Thus, it is important to incorporate analysis of interactions between flows when evaluating protocols.

Figure 10. QUICv34 vs. TCP for varying object sizes on a Nexus6 smartphone (using WiFi). We find that QUIC's improvements diminish or disappear entirely when running on mobile devices.



(a) Nexus6, No added loss or latency



(b) Nexus6, 1% Loss

Table 2. The fraction of time QUIC spent in each state on MotoG vs. Desktop. QUICv34, 50Mbps, no added loss or delay. The table shows that poor performance for QUIC on mobile devices can be attributed to applications not processing packets quickly enough. Note that the zero probabilities are due to rounding.

	Desktop	Mobile
Init	0.01%	0.01%
Slow start	1.65%	0.42%
Application limited	7.05%	58.84%
Congestion avoidance	91.28%	40.55%
Tail loss probe	0.00%	0.00%
Recovery	0.00%	0.18%

References

1. Chrome debugging protocol. https://developer.chrome.com/devtools/docs/debugger-protocol.
2. https://github.com/quicwg/basedrafts/wiki/implementations.
3. I. grigorik. Deciphering the critical rendering path. https://calendar.perfplanet.com/2012/deciphering-the-critical-rendering-path/.
4. QUIC at 10,000 feet. https://docs.google.com/

document/d/1gY9-YNDNAB1eip-RTPbqphgySwSNSDHLq9D5Bty4FSU.
5. TCP probe. https://wiki.linuxfoundation.org/networking/tcpprobe.
6. Welch's t-test. https://en.wikipedia.org/wiki/Welch%27s_t-test.
7. Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M., Ernst, M.D. 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).
8. Carlucci, G., De Cicco, L., Mascolo, S. HTTP over UDP: An experimental investigation of QUIC. In *Proc. of SAC* (2015).
9. Chromium Blog. A QUIC update on Google's experimental transport, April 2015. http://blog.chromium.org/2015/04/a-quic-update-on-googles-experimental.html.
10. Cimpanu, C. Google creates new algorithm for handling TCP traffic congestion control, September 2016. http://news.softpedia.com/news/google-creates-new-algorithm-for-handling-tcp-traffic-congshtml.
11. Das, S.R. Evaluation of QUIC on Web page performance. Master's thesis, Massachusetts Institute of Technology (2014).
12. Dukkipati, N., Cardwell, N., Cheng, Y., Mathis, M. Tail loss probe (TLP): An algorithm for fast recovery of tail losses, February 2013. https://tools.ietf.org/html/draft-dukkipati-tcpm-tcp-loss-probe-01.
13. Ha, S., Rhee, I. Taming the elephants: New TCP slow start. *Comput. Netw.*, 2011.
14. Kakhki, A.M., Jero, S., Choffnes, D., Nita-Rotaru, C., Mislove, A. Taking a long look at QUIC: An approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference* (2017).
15. Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Kulik, J., Roskind, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.-T., Shi, Z. The QUIC transport protocol: Design and Internet-scale deployment. In *Proc. of ACM SIGCOMM* (2017).
16. Mathis, M., Dukkipati, N., Cheng, Y. Proportional rate reduction for TCP, May 2013. https://tools.ietf.org/html/rfc6937.
17. Megyesi, P., Krämer, Z., Molnár, S. How quick is QUIC? In *Proc. of ICC* (May 2016).
18. Nikravesh, A., Yao, H., Xu, S., Choffnes, D.R., Mao, Z.M. Mobilyzer: An open platform for controllable mobile network measurements. In *Proc. of MobiSys* (2015).
19. Odvarko, J., Jain, A., Davies, A. HTTP Archive (HAR) format, August 2012. https://dvcs.w3.org/hg/webperf/raw-file/tip/specs/HAR/Overview.html.
20. Swett, I. QUIC congestion control and loss recovery. https://docs.google.com/presentation/d/1T9GtMz1CvPpZtmF8g-W7j9XHZBOCp9cu1fW0sMsmpoo.
21. Swett, I. QUIC deployment experience "Google, 2016. https://www.ietf.org/proceedings/96/slides/slides-96-quic-3.pdf.
22. Swett, I. QUIC FEC v1, February 2016. https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk/edit.
23. Wang, X.S., Balasubramanian, A., Krishnamurthy, A., Wetherall, D. How speedy is SPDY? In *Proc. of USENIX NSDI* (2014).
24. Zhang, M., Karp, B., Floyd, S., Peterson, L. 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.

Arash Molavi Kakhki (arash@thousandeyes.com), ThousandEyes, Boston, MA, USA.

Samuel Jero (sjero@purdue.edu), Purdue University, West Lafayette, IN, USA.

David Choffnes and Alan Mislove ({choffnes,amislove}@ccs.neu.edu), College of Computer and Information Science, Northeastern University, Boston, MA, USA.

Cristina Nita-Rotaru (c.nitarotaru@neu.edu), College of Computer and Information Science, Northeastern University, Boston, MA, USA.