Understanding Speciation in QUIC Congestion Control

Ayush Mishra, Sherman Lim, and Ben Leong
National University of Singapore

ABSTRACT

The QUIC standard is expected to replace TCP in HTTP 3.0. While QUIC implements a number of the standard features of TCP differently, most QUIC stacks re-implement standard congestion control algorithms. This is because these algorithms are well-understood and time-tested. However, there is currently no systematic way to ensure that these QUIC congestion control protocols are implemented correctly and predict how these different QUIC implementations will interact with other congestion control algorithms on the Internet.

To address this gap, we present *QUICbench*, which, to the best of our knowledge, is the first congestion control benchmarking tool for QUIC stacks. *QUICbench* determines how closely the implementation of a QUIC congestion control algorithm conforms to the reference (kernel) implementation by comparing their respective throughput-delay tradeoffs. *QUICbench* can also be used to systematically compare a new QUIC implementation to previous and different implementations of both QUIC and kernel-based congestion control algorithms. Our measurement study suggests that there is already significant deviation between the existing QUIC implementations of standard congestion control algorithms from the reference implementations. We demonstrate how *QUICbench* can help us identify the implementation differences responsible for these deviations so that they can be suitably corrected.

CCS CONCEPTS

• **Networks** → **Transport protocols**; **Network performance analysis**.

KEYWORDS

IETF QUIC, Congestion Control, Measurement

ACM Reference Format:
Ayush Mishra, Sherman Lim, and Ben Leong. 2022. Understanding Speciation in QUIC Congestion Control. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC '22), October 25–27, 2022, Nice, France*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3517745.3561459

1 INTRODUCTION

First introduced by Google in 2013, the QUIC protocol has matured over the years and is set to replace TCP in HTTP 3.0. QUIC offers many benefits over TCP, including faster connection establishment, baked-in end-to-end encryption, and support for multi-streaming. Since it is implemented in the user space, it allows developers to easily modify and push updates to their QUIC stacks. While this flexibility could potentially allow QUIC to become a more secure alternative to TCP, the converse is also true: it also makes it easier to make mistakes.

The QUIC standard, as described by its many prescriptive IETF RFCs and drafts today [5], implements a protocol that is different from TCP. However, existing QUIC stacks [1] still implement the classic congestion control algorithms (CCA) used by TCP instead of inventing new ones. There is a good reason for this. Classic congestion control algorithms are well understood, predictable, and already have good track records of convergence and stability guarantees thanks to years of research and Internet deployment.

QUIC developers however do not currently have a systematic way to ensure that their implementations are correct and fully compliant with standard kernel implementations. In other words, there is presently no safeguard against *buggy* versions of these standard congestion control algorithms from being deployed on the Internet. To address this gap, we developed *QUICbench*, which to the best of our knowledge is the first congestion control algorithm benchmarking tool for QUIC stacks. For now, *QUICbench* has 2 key applications:

1. **Protocol Conformance.** To allow us to quantify how closely QUIC implementations of existing CCAs conform to the standard kernel (reference) implementations, we introduce a new metric called the *Performance Envelope* (PE). We will use the similarities between the PE of a test QUIC implementation and its corresponding standard kernel implementation as a measure of the QUIC implementation's conformance.
2. **Understanding interactions between implementations.** It would also be helpful for QUIC developers to understand how their CCA implementations will interact with other CCA implementations, both QUIC and in-kernel. This would allow them to tune their QUIC stacks and congestion control algorithms to suit their specific operational requirements. To this end, *QUICbench* also allows us to compare the performance of new QUIC CCA implementations to existing implementations.

In this paper, we describe our methodology to benchmark QUIC stacks by Google[1] [4], Facebook[2] [3], Microsoft[3] [9], and Cloudflare[4] [2] and present the results of our measurement study. We have limited our analysis to these four stacks because of space constraints and because to the best of our knowledge, these are the four most popular open-sourced stacks on the Internet today. *QUICbench* is easily extensible to other QUIC stacks as well and we plan to do so soon. Our results show that there is already significant *speciation*[5] between the CCA implementations in these four QUIC

[1] `chromium` commit hash: 82a3c71cf5bf2502d5ad90489fe20ce8f8cb3fab
[2] `mvfst` commit hash: 65a9c066e742620becacc99b7c0ca86200e6a4c4
[3] `msquic` commit hash: e6110b62cd8e0d84e6436bde2504e6bc0702921a
[4] `quiche` commit hash: 9dfeaafb625b08760218def7beb8db133e3f50cb
[5] Speciation is the evolutionary process by which populations evolve to become distinct species [10].

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
IMC '22, October 25–27, 2022, Nice, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9259-4/22/10.
https://doi.org/10.1145/3517745.3561459

Table 1: QUIC/TCP stacks evaluated and the available congestion control algorithms.

Organization	Stack	CUBIC	BBR	Reno
-	Linux kernel	✓	✓	✓
Facebook	`mvfst` [3]	✓	✓	✓
Google	`chromium` [4]	✓	✓	✗
Microsoft	`msquic` [9]	✓	✗	✗
Cloudflare	`quiche` [2]	✓	✗	✓

stacks and we suspect that some of the deviants could have been intentional to achieve better performance (but at the expense of other CCAs on the Internet). The following are our key findings:

(1) There are significant deviations between the different implementations of congestion control algorithms in QUIC and their respective kernel implementations;
(2) The `mvfst` BBR variant deviates significantly from standard BBR in terms of throughput in shallow buffers, often taking up most of the bottleneck bandwidth against a competing flow. This is because `mvfst` BBR implements an additional scaling factor of 120% on top of BBR's usual sending rate;
(3) The `chromium` CUBIC variant emulates two flows and uses a different β value from the standard kernel implementation. This modification does not have much impact on performance, but it illustrates that *QUICbench* is able to identify minor deviations; and
(4) In general, we have found the performance differences across these different implementations to be *transitive*. By transitive, we mean that if a QUIC implementation α outperforms another implementation β, and β outperforms γ, then α would also generally outperform γ. We have found this to be true for the range of buffer sizes that we investigated.

QUICbench is currently open-sourced and available under the MIT License [21].

2 METHODOLOGY

There are currently at least 25 existing open-sourced implementations of QUIC [1]. We have limited our scope to four of the most common ones because of space constraints. These stacks are all currently being developed actively and deployed widely by major public companies, so we believe that our results are representative of the current QUIC ecosystem deployed in the wild. The CCAs available for these stacks are summarized in Table 1.

Testbed Setup. We ran our experiments on a testbed comprising of two Linux (Ubuntu 20.04, kernel version 5.13.0) machines connected via a 1 Gbps Ethernet cable. We installed the open-source repositories of the QUIC stacks on the two machines and use the sample QUIC clients/servers provided in the repositories to generate the QUIC network flows for our experiments. The open-source tool, iperf3 [6], was used for generating the TCP flows. The bottleneck of the connection and the RTT were emulated using TBF [8] and netem qdisc [7]. The UDP and TCP socket buffer sizes were set to the same value (12,582,912 bytes).

2.1 The Performance Envelope

To understand how well the CCAs implemented in a QUIC stack compare to the standard reference implementations, we needed a way to quantify and visualize the difference. We also want to be able to do so in a code-agnostic way, i.e. we should not have to read the code to pick out any deviations.

One straightforward way to identify deviations would be to compare the `cwnd` evolution to a reference implementation running in identical network conditions [22]. However, we argue that this is impractical in the context of QUIC. Given that QUIC is implemented in the user space, it is unreasonable to expect these implementations to accurately replicate the complex waveforms in algorithms like CUBIC exactly. User space implementations of these algorithms generally have a fast path and a slow path that serves to approximate the behaviour of these algorithms, but they do not exactly match them. If the developers set out to exactly match the `cwnd` evolution of these algorithms, they would not be able to realistically do so in the user space without significant impact on performance.

An alternative is to define a more *coarse-grained* definition for conformance in the context of fairness. If a new implementation of a congestion control algorithm is as fair/unfair to a reference flow as the standard implementation, we can say that it is behaving in a manner *conformant* to the reference implementation. However, we are of the view that such an approach will not adequately capture the finer differences between different congestion control algorithms.

Our key insight is that a good measure of conformance should be based on a metric that captures the different trade-offs of different congestion control algorithms. To this end, we propose the *Performance Envelope* (PE), a multi-dimensional metric for comparing the relative behaviour of different implementations of different congestion control algorithms. The Performance Envelope is a visual representation of the different trade-offs made by different congestion control algorithms.

In this paper, we will evaluate the various implementations of standard congestion control algorithms in QUIC by looking at their throughput and delay trade-offs. We decided to choose these two metrics, because the trade-off between throughput and delay is the key consideration in the design of most modern congestion control algorithms [11, 14–16, 25]. Therefore, even if an implementation is not completely accurate, it should at least offer the same throughput-delay tradeoffs as the reference (kernel) implementation. However, depending on the application, the performance envelope can be adapted to capture the trade-offs between other network metrics as well.

Defining the Performance Envelope. To determine the performance envelope for an algorithm, we launch a control flow and a test flow through a common bottleneck link. In this setting, the control flow represents the standard implementation of the congestion control algorithm and the test flow represents the QUIC implementation that is being tested. We measure the time series throughput and delay data for the test flow. This data is then sampled at regular time intervals of 10 RTTs. We record the instantaneous throughput T Mbps and the queuing delay d for each interval. The *performance envelope* is defined as the 2D region (defined by the convex hull) that contains 95 percent of the operating points (d, T) for the algorithm in the throughput-delay plane. In other words, in plotting the PE, we reject 5% of the data points with the largest euclidean distance from the centroid as outliers.

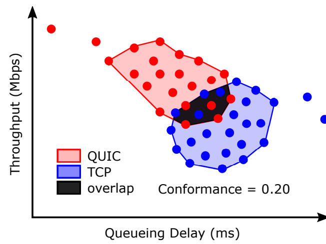


Figure 1: Calculating the *Conformance* between two PEs.

Defining Similarity. Our key insight in determining how closely an implementation conforms to a reference implementation is based on *replace-ability*. That is, a truly conformant implementation should be able to replace the reference implementation without any noticeable impact on its performance envelope *across a whole range of operating conditions*. Therefore, to measure conformance, we plot the performance envelopes for the reference (kernel) implementation and the test QUIC implementation to determine the overlap between them. We quantify this overlap using two metrics:

(1) **Conformance**. A conformance of 1 means the performance envelope of the test implementation completely overlaps with the performance envelope of the corresponding standard kernel implementation. However, not all overlapping areas are the same. Because of the structure of the performance envelopes, it is possible for two overlapping regions to have the exact same area but share different numbers of common data points. To capture this density information, we calculate conformance as the ratio of the number of points within the overlapped region over the total number of sampled points.
(2) **Deviation**. In practice, many of the performance envelopes for QUIC implementations have no overlap with their associated kernel implementation, and thus they have zero conformance. In order to capture the relationship between them, we define an additional metric called *deviation*, which is the euclidean distance between the centroids of two performance envelopes after normalizing the throughput using the maximum observed throughput and the delay using the maximum observed delay. As a result, deviation $\leq \sqrt{2}$.

In Figure 1, we illustrate how the conformance between two performance envelopes is calculated.

2.2 Understanding Interactions between Implementations

It is perfectly plausible and reasonable for QUIC developers to want to implement their own versions of standard congestion control algorithms or even entirely new congestion control algorithms. Since the current crop of QUIC stacks are implemented by companies, it would also not be surprising that they might tune and optimize their stack to improve the performance of their applications. In such a setting, while conformance might not be all that important, we still want to make sure that these *modified* implementations' performance is not so skewed that it begins to hurt other TCP and QUIC flows on the Internet. Therefore, we also conducted a bandwidth-share-based analysis. In particular, we launch experiments where we made two flows from different implementations compete (not limited to the same congestion control algorithm) and measured their throughput ratios.

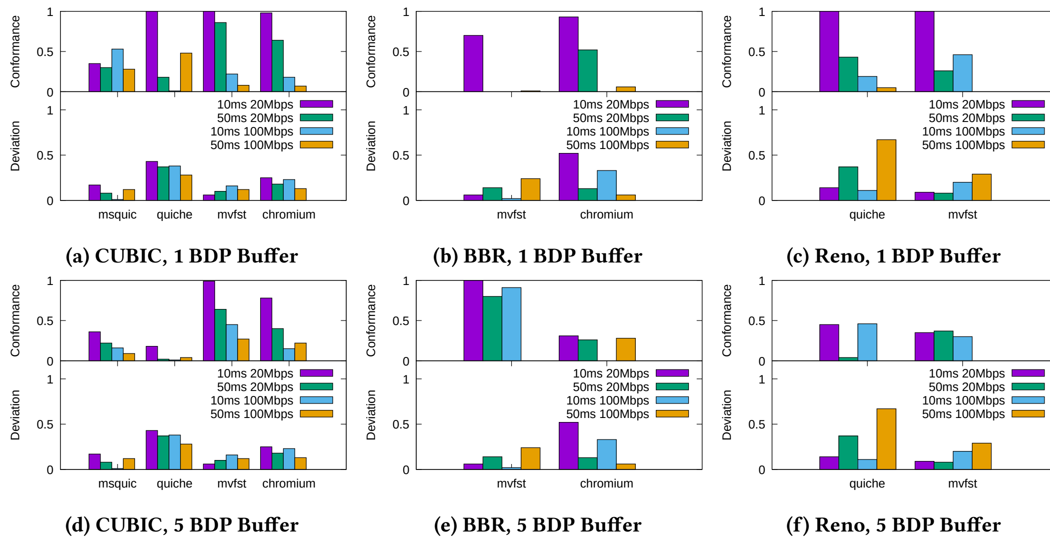
3 RESULTS

In this section, we present the results of our measurement study for the QUIC stacks listed in Table 1.

3.1 Differences in Performance Envelopes

As described in §2.1, we ran two-flow experiments with a test QUIC flow and the corresponding reference TCP (kernel) flow and measured the performance envelope of the test flow. We tested flows with a base RTT of 10 ms and 50 ms (since these are typical RTTs on wired and wireless networks [27] respectively) through bottlenecks with 20 Mbps and 100 Mbps constant link capacity. The flows were run for 2 minutes and the bottleneck buffer size was varied in multiples of the bandwidth-delay product (BDP). The results are shown in Figure 2. The following is a summary of the key observations:

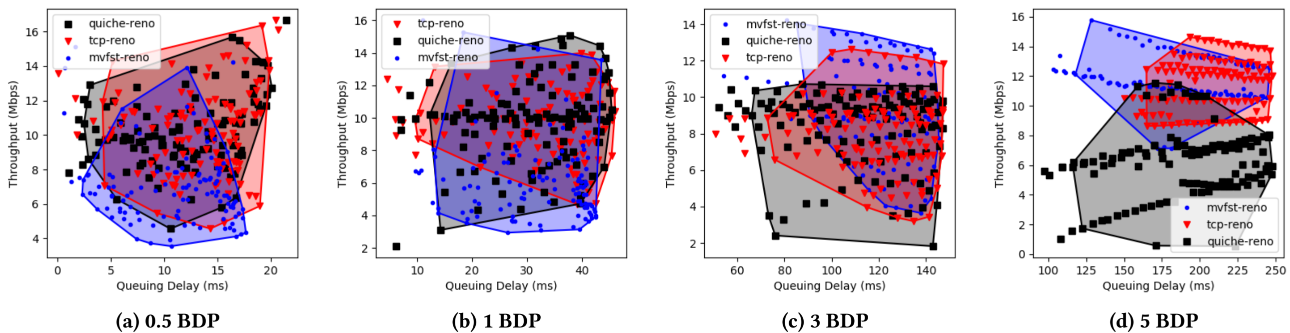
(1) **Unfairness viz-a-viz TCP kernel flows**. It is clear from Figure 2 that current implementations of standard CCAs already deviate significantly from standard kernel implementations. In general, we have found that the performance envelopes of most QUIC implementations lie above their TCP counterparts. This suggests that these QUIC implementations are more aggressive compared to the kernel implementations. This was also observed in a previous study [18]. Wolsing et al. suggested that this gap could be closed by tuning some kernel parameters (larger initial window size, larger `rcv` buffers [31]). However, we found no noticeable improvement in performance after applying the suggested changes. A possible explanation for this is that Wolsing et al. evaluated short flows, while we are evaluating long flows.
(2) **Lack of systematic deviation**. While the tested QUIC implementations in general seem to be more aggressive towards the standard TCP implementations in the kernel, it is unlikely that this is because of something in the QUIC standard. The fact that we are seeing deviations in performance in both directions suggests that most of these deviations are implementation specific. We have also noticed that both conformance and deviation are highly contextual and depend a lot on the network conditions.
(3) **Reno implementations most conformant across stacks.** From Figure 2, we also note that Reno implementations in general show higher conformance and lower deviation than the implementations of other congestion control algorithms like CUBIC and BBR (especially in shallower buffers). This is hardly surprising, since Reno is algorithmically the simplest CCA of the three and is therefore likely the easiest to implement.
(4) **Loss-based vs. Rate-based**. The performance envelopes for the loss-based algorithms, CUBIC and Reno, had a much larger variation in queuing delay compared to that for BBR. This is expected, since loss-based congestion control algorithms frequently back off and then re-fill the bottleneck



(a) CUBIC, 1 BDP Buffer (b) BBR, 1 BDP Buffer (c) Reno, 1 BDP Buffer

(d) CUBIC, 5 BDP Buffer (e) BBR, 5 BDP Buffer (f) Reno, 5 BDP Buffer

Figure 2: The *Conformance* and *Deviation* of various implementations in shallow (1 BDP) and deep (5 BDP) buffers.



(a) 0.5 BDP (b) 1 BDP (c) 3 BDP (d) 5 BDP

Figure 3: Performance Envelopes for various Reno implementations tested on a 20 Mbps link with 50 ms base RTT.

buffer, while BBR is largely loss-agnostic [28]. We also notice that in general for loss-based flows like CUBIC and Reno, conformity worsens in deeper buffer. However, the opposite is true for BBR, with its conformity improving as the bottleneck buffers get deeper. This trend was consistent across all the network conditions tested by us, and most visually apparent in the network setting with 50 ms RTT and 20 Mbps link speed (see Figures 3 and 4).

3.2 Interactions with Other CC Algorithms.

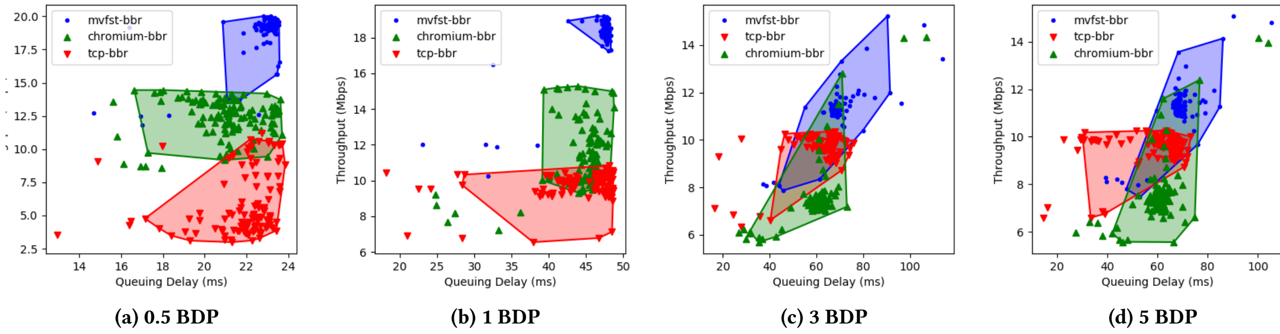
To understand how different QUIC implementations interact and co-exist with each other, we also measured the throughput ratios when 2 flows compete at a common bottleneck. The bottleneck link speed was set to a constant 50 Mbps and the RTT of both the flows was set at 20 ms. We plot a heat map of the pair-wise throughput ratios of the congestion control algorithm implementations studied in Figure 5. The throughput ratio in each square in the heat-map is calculated as $\frac{T_x}{T_x+T_y}$, where T_x and T_y are the throughputs of the corresponding implementations listed along the x and y axes, respectively. Therefore, if the throughput ratio is greater than 0.5, $T_x > T_y$.

While there are varying degrees of unfairness between flows (as expected), two outliers stand out: (i) `mvfst` BBR is *extremely* unfair to all the other tested implementations in shallow buffers (with it taking up to 50× bandwidth compared to the competing flows!); (ii) `chromium` CUBIC seems to be extremely aggressive when competing against `quiche` Reno in deep buffers. In general, we also found the throughput-wise performance to be *transitive*. That is, if implementation α outperforms another implementation β, and β outperforms γ, then α would also generally outperform γ. We also note that all the CCAs implemented in `chromium` are extremely fair to one another.

3.3 Implementation-level differences

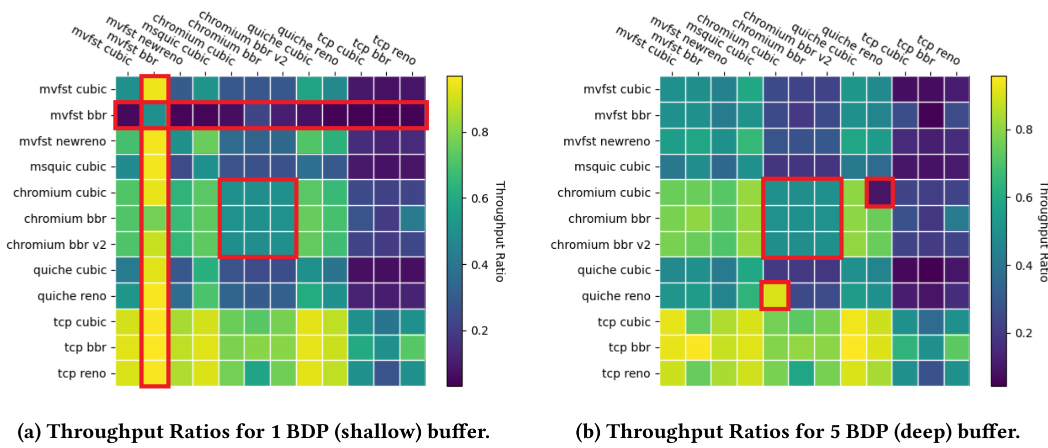
With the help of *QUICbench*, we have been able to identify three implementation-level differences that impact the performance of three QUIC implementations:

(1) `mvfst` **BBR's scaled up sending rate.** We found that `mvfst` BBR applied a scaling factor of 120% to BBR's usual sending rate, which explains why `mvfst` BBR was extremely aggressive compared to other CCA implementations. We verified



(a) 0.5 BDP (b) 1 BDP (c) 3 BDP (d) 5 BDP

Figure 4: Performance Envelopes for various BBR implementations tested on a 20 Mbps with 50 ms base RTT.



(a) Throughput Ratios for 1 BDP (shallow) buffer. (b) Throughput Ratios for 5 BDP (deep) buffer.

Figure 5: Plot of throughput ratios for 2 flows sharing a common bottleneck.

this by removing this scaling factor and evaluating the resulting implementation. The new performance envelope is shown in Figure 6. By returning the gain to 100% as intended, we were able to improve the conformance of `mvfst` BBR from zero to up to 0.8 in some networks conditions.

(2) `chromium` **CUBIC emulating multiple flows.** We found that by default, `chromium` CUBIC emulates 2 CUBIC flows ($N = 2$) and therefore has an adjusted β value. Instead of having $\beta = 0.7$ like in the standard CUBIC algorithm, `chromium` CUBIC sets $\beta = 0.85$. As shown in Figure 7, a change (to $N = 1$) significantly improves `chromium` CUBIC's conformance with the kernel implementation. We also saw its throughput ratio compared to `quiche` Reno reduce from 10.45 (see Figure 5b) to 6.25 in 5 BDP buffers. However, even after making this change `chromium` CUBIC has significantly different performance compared to its TCP counterpart.

(3) `mvfst` **Reno consistently does worse than TCP Reno.** While we did not find anything algorithmically different between the Reno implementations in `mvfst` and in the Linux kernel, the `mvfst` implementation consistently achieved lower throughput when it competed with the Linux implementation (i.e., the throughput ratio was below 0.6 for all tested network conditions). We noticed one implementation difference between the `mvfst` and the kernel implementations: ACK frequency. By default, `mvfst` send ACKs for every 10 packets. When we increased this ACK frequency match the kernel implementation by sending ACKs for every 2 packets, the performance of `mvfst` Reno noticeably improved (i.e., throughput ratio exceeded 0.8 for all tested network conditions), albeit at the cost of a slight increase in CPU utilization.

4 DISCUSSION

Our preliminary investigation into the congestion control implementations in four popular QUIC stacks shows that speciation in QUIC congestion control deserves more careful study as the QUIC standard matures. It is important for developers, as well as the people who deploy these stacks, to be aware of potential differences and the impact that they might have on existing congestion control algorithms.

Adding more QUIC stacks. The fact that performance deviations could go in either direction compared to reference implementations suggests that performance deviations are dependent on stacks' specific implementation differences and are not merely the result of the protocol-level differences between QUIC and TCP. This suggests that QUIC stacks should be evaluated more comprehensively and it is not sufficient to only study the QUIC implementations of

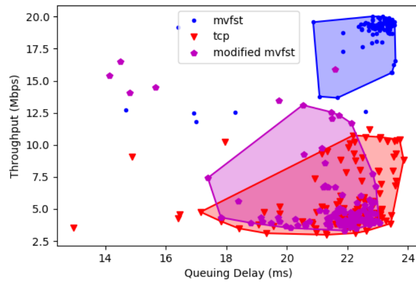


Figure 6: Improving `mvfst` BBR's conformance by modifying its gain to 100% (modified `mvfst`). (0.5 BDP, 20 Mbps, 50 ms)

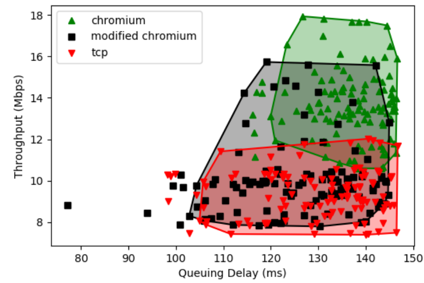


Figure 7: Improving `chromium` CUBIC's conformance by setting $N = 1$ (modified `chromium`). (3 BDP, 20 Mbps, 50 ms)

the congestion control protocols. We have thus far only studied four QUIC stacks, and we plan to extend this project to more available QUIC stacks.

Context matters. We have found that PEs would vary depending on the network parameters. We plan to extend *QUICbench* to benchmark QUIC stacks under other network conditions (like mobile links, wireless) to develop a more comprehensive benchmark in order to understand how the PEs depend on the network parameters. We suspect that with more data, we will also be able to refine our current definitions of the PE, conformance, and deviation.

Stack-level parameter tuning. We expect many of the performance differences between the QUIC stacks to be due to stack-level parameters like ACK frequency and starting window size affecting the congestion control algorithms. We plan to extend *QUICbench* to extract these stack-level parameters (either via measurement or by directly looking for them in the code-base) and suggest fixes for CCA implementations that show low conformance and large deviation.

Root-cause analysis. Finally, we hope to conduct more root-cause analyses to identify the modifications that affect an implementation's performance. As illustrated in §3.3, root-cause analysis can lead to actionable insights. While we were able to find the modification required to bring Facebook's `mvfst` BBR variant closer to the reference implementation, we were less successful with Google's `chromium` CUBIC variant. This shows that even if we can determine that there is a deviation, identifying the root cause is non-trivial. Nevertheless, we have shown that *QUICbench* can potentially help us identify bugs that can lead to unintentional performance trade-offs.

5 RELATED WORK

Over the years, there has been a large body of work dedicated to studying the QUIC standard as it evolved. To the best of our knowledge, Marx et al. [19] were the first to take a holistic approach to evaluate the low-level details in IETF QUIC using *Qvis* [20]. They studied the implementation of key QUIC features like the 0-RTT handshake mechanism, loss notification, and multi-streaming. With respect to congestion control, they evaluated stack-level parameters like initial window size, pacing, and ACK frequency. They found a significant amount of heterogeneity in these parameters across the 15 QUIC stacks that they studied. However, the performance of the different implementations of congestion control algorithms was not evaluated.

There have also been a number of studies that compared the performance of QUIC to TCP [12, 18, 26, 31]. Kakhki et al. found that in general, QUIC outperforms TCP+TLS in most networks [18]. They reported that the performance improvement was mainly the result of the QUIC standard increasing its maximum window size in 2016. Saif et al. showed that even though QUIC can outperform TCP in terms of throughput, its QoS can be worse [26]. Bhat et al. highlighted a similar issue with DASH video workloads, where videos were able to achieve a better QoS with TCP even though they were getting better bandwidth shares with gQUIC [12]. QUIC was evaluated against TCP in the wild with production traffic by Wolsing et al. [31]. Again, they also found that QUIC outperforms TCP in terms of throughput. They made suggestions on how the TCP stack can be tuned to make it more competitive against production QUIC. We tried to replicate these suggested changes, but found that they did not improve TCP's performance in our testbed. They also note that in general, QUIC's performance is an artifact of how the stack has been implemented, and is not a byproduct of the standard itself.

Beyond QUIC, there is a large body of work on the deployability of and fairness between different congestion control algorithms. This was spurred by measurement studies that suggested that the Internet's TCP congestion control landscape itself is extremely heterogeneous [23]. There has been a recent surge in studies evaluating how well CUBIC and BBR co-exist [13, 17, 24, 29, 30]. The consensus from this body of work is that there can be varying degrees of unfairness between CCAs depending on the network conditions. While these works provide important insights into how different algorithms interact with each other, the focus of our work is to study the similarity between standard kernel CCAs and their associated QUIC implementations.

6 CONCLUSION

In our work, we sought to study the congestion control algorithm (CCA) implementations of QUIC stacks by benchmarking their performance against the reference kernel implementations using *QUICbench*. Our analysis uncovered significant performance differences between the user-space CCAs and their reference implementations in the kernel due to implementation-level differences. This suggests that some level of speciation is happening with regard to the CCAs used in QUIC stacks, which deserves further study. We plan to expand our analysis to more QUIC stacks and evaluate their CCAs over a wider range of network conditions.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers, Ali Razeen, Joshua Koo, and our shepherd, Mirja Kuehlewind, for providing their valuable feedback. This work was supported by the Singapore Ministry of Education grant T1 251RES1917.

REFERENCES

[1] 2021. *Active QUIC implementations.* https://github.com/quicwg/base-drafts/wiki/Implementations
[2] 2021. *Cloudflare's QUIC implementation, quiche.* https://github.com/cloudflare/quiche
[3] 2021. *Facebook's QUIC implementation, mvfst.* https://github.com/facebookincubator/mvfst
[4] 2021. *Google's QUIC implementation, chromium.* https://www.chromium.org/quic/playing-with-quic
[5] 2021. *IETF QUIC Working Group.* https://datatracker.ietf.org/wg/quic/
[6] 2021. *iPerf, the Speed Test Tool for TCP.* https://iperf.fr/iperf-doc.php
[7] 2021. *Linux Traffic Control, netem qdisc.* https://wiki.linuxfoundation.org/networking/netem
[8] 2021. *Linux Traffic Control, Token Bucket Filter qdisc.* https://linux.die.net/man/8/tc-tbf
[9] 2021. *Microsoft's QUIC implementation, msquic.* https://github.com/microsoft/msquic
[10] 2022. Speciation. https://en.wikipedia.org/wiki/Speciation
[11] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *Proceedings of NSDI*.
[12] Divyashri Bhat, Amr Rizk, and Michael Zink. 2017. Not so QUIC: A Performance Study of DASH over QUIC. 13–18. https://doi.org/10.1145/3083165.3083175
[13] Yi Cao, Arpit Jain, Kriti Sharma, Aruna Balasubramanian, and Anshul Gandhi. 2019. When to use and when not to use BBR: An empirical analysis and evaluation study. In *Proceedings of the IMC*.
[14] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2017. BBR: congestion-based congestion control. *Commun. ACM* 60, 2 (2017), 58–66.
[15] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *NSDI*.
[16] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.
[17] Per Hurtig, Habtegebreil Haile, Karl-Johan Grinnemo, Anna Brunstrom, Eneko Atxutegi, Fidel Liberal, and Åke Arvidsson. 2018. Impact of TCP BBR on CUBIC traffic: A mixed workload evaluation. In *30th International Teletraffic Congress (ITC 30)*.
[18] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In *Proceedings of IMC*.
[19] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ)*.
[20] Robin Marx, Wim Lamotte, Jonas Reynders, Kevin Pittevils, and Peter Quax. 2018. Towards QUIC debuggability. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ)*.
[21] Ayush Mishra and Sherman Lim. 2022. QUICBench. https://github.com/NUS-SNL/QUICbench
[22] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. 2019. The Great Internet TCP Congestion Control Census. 59–60. https://doi.org/10.1145/3393691.3394221
[23] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. 2019. The Great Internet TCP Congestion Control Census. In *Proceedings of SIGMETRICS*.
[24] Ayush Mishra, Jingzhi Zhang, Melodies Sims, Sean Ng, Raj Joshi, and Ben Leong. 2021. Conjecture: Existence of Nash Equilibria in Modern Internet Congestion Control. In *APNet*.
[25] Vern Paxson and Mark Allman. 2009. TCP Congestion Control. RFC 5681.
[26] Darius Saif, Chung-Horng Lung, and Ashraf Matrawy. 2020. An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse.
[27] Satadal Sengupta, Hyojoon Kim, and Jennifer Rexford. 2022. Continuous in-network round-trip time monitoring. In *Proceedings of SIGCOMM*.
[28] Bruce Spang, Serhat Arslan, and Nick McKeown. 2021. Updating the theory of buffer sizing. *Performance Evaluation* 151 (2021), 102232.
[29] Belma Turkovic, Fernando A Kuipers, and Steve Uhlig. 2019. Interactions between congestion control algorithms. In *Network Traffic Measurement and Analysis Conference (TMA)*.
[30] Ranysha Ware, Matthew K. Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's Interactions with Loss-Based Congestion Control. In *Proceedings of IMC*.
[31] Konrad Wolsing, Jan Rüth, Klaus Wehrle, and Oliver Hohlfeld. 2019. A performance perspective on web optimized protocol stacks: TCP+ TLS+ HTTP/2 vs. QUIC. In *Proceedings of the Applied Networking Research Workshop*.

A ETHICS

This work does not raise any ethical issues.