Containing the Cambrian Explosion in QUIC Congestion Control

Ayush Mishra
National University of Singapore

Ben Leong
National University of Singapore

ABSTRACT

Since its introduction in 2015, QUIC has seen rapid adoption and is set to be the default transport stack for HTTP3. Given that developers can now easily implement and deploy their own congestion control algorithms in the user space, there is an imminent risk of the proliferation of QUIC implementations of congestion control algorithms that no longer resemble their corresponding standard kernel implementations.

In this paper, we present the results of a comprehensive measurement study of the congestion control algorithm (CCA) implementations for 11 popular open-source QUIC stacks. We propose a new metric called *Conformance-T* that can help us identify the implementations with large deviations more accurately and also provide hints on how they can be modified to be more conformant to reference kernel implementations. Our results show that while most QUIC CCA implementations are conformant in shallow buffers, they become less conformant in deep buffers. In the process, we also identified five new QUIC implementations that had low conformance and demonstrated how low-conformance implementations can cause unfairness and subvert our expectations of how we expect different CCAs to interact. With the hints obtained from our new metric, we were able to identify implementation-level differences that led to the low conformance and derive the modifications required to improve conformance for three of them.

CCS CONCEPTS

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

KEYWORDS

IETF QUIC, Congestion Control, Measurement

ACM Reference Format:
Ayush Mishra and Ben Leong. 2023. Containing the Cambrian Explosion in QUIC Congestion Control. In *Proceedings of the 2023 ACM Internet Measurement Conference (IMC '23), October 24–26, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3618257.3624811

1 INTRODUCTION

QUIC was first introduced by Google in 2015 to address the limitations of TCP and to add security enhancements to HTTP [31]. It has since seen rapid adoption and has been designated as the default transport stack for HTTP3. QUIC is estimated to already constitute some 30% of downstream traffic in EMEA (Europe, Middle East, and Africa) and 16% of downstream traffic in North America [43].

While the original motivation behind QUIC was to improve security, QUIC's protocol designers also used this opportunity to redesign many legacy aspects of TCP, like the handshake, and to introduce multi-streaming. However, for congestion control, most QUIC developers chose to be conservative and re-implemented standard congestion control algorithms (CCAs) like CUBIC [28], Reno, and BBR [25]. This is not surprising since these algorithms are well understood and predictable, thanks to many years of deployment on the Internet. Their stability properties are especially important since QUIC already takes up a significant share of today's Internet traffic.

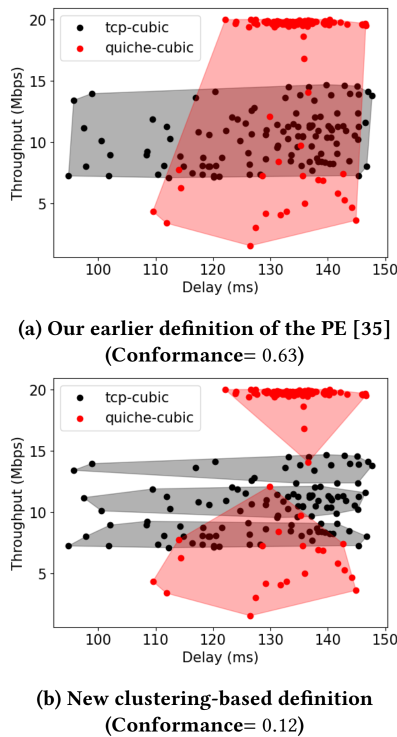
In general, for compatibility with existing CCAs, we expect these QUIC CCA implementations to have the following two properties:

(1) **Behave like their kernel counterparts.** Since the entire motivation behind re-implementing standard congestion control algorithms is to achieve predictability and stability, we want QUIC implementations to resemble their corresponding kernel implementations. In particular, we expect QUIC CCAs to not only achieve delays and throughputs similar to their kernel counterparts but also to interact with existing CCAs qualitatively in the same way.
(2) **TCP friendliness.** We want new QUIC congestion control implementations to co-exist well with existing Internet traffic and be friendly towards other QUIC and TCP kernel implementations. In particular, it would be disastrous if new implementations cause significant degradation or starvation of existing flows.

In this paper, we investigated how closely the CCA implementations for the 11 popular open-source QUIC stacks listed in Table 1 adhere to these expectations. These stacks were selected because they are all open source, stable, and deployed on the Internet. While it is easy to evaluate TCP-friendliness and qualitative interactions between different implementations, it is much harder to define the *ideal* behavior of a congestion control algorithm. Even if we use a given implementation as the standard reference for a congestion control algorithm, it is not entirely straightforward how we can quantify how well a new implementation conforms to this reference implementation. For example, a simple fairness-based measure of conformance would not be desirable as it would fail to capture the algorithmic nuances between different CCA implementations. We had argued earlier that any measure of similarity should capture the following two properties (relative to a reference implementation) [35]:

(1) **Replaceability.** How easily can a third-party observer tell if we replaced the reference implementation with the new implementation?

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 '23, October 24–26, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0382-9/23/10.
https://doi.org/10.1145/3618257.3624811



(a) Our earlier definition of the PE [35] (Conformance= 0.63)

(b) New clustering-based definition (Conformance= 0.12)

Figure 1: A single convex hull for the PE does not fully capture low conformance in `quiche` CUBIC.

(2) **Inherent performance trade-offs.** Different congestion control algorithms represent different trade-offs in a network. Does the new implementation operate in the same trade-off space as the standard reference implementation?

To this end, we proposed a metric called the *Performance Envelope* (PE) that was built on these two ideas [35]. To investigate a new QUIC CCA implementation, we sample the delay (d) and throughput (T) of the new implementation while it competed with the reference (kernel) implementation in a controlled network environment. These (d, T) pairs were then plotted on a delay-throughput plane to visualize the delay-throughput trade-off space for the new QUIC CCA implementation. The region defined by the convex hull of this point cloud was referred to as the PE of the new QUIC CCA implementation.

To measure the replaceability of a QUIC implementation, the PE of the QUIC implementation competing with a reference implementation was compared with the PE of a reference implementation competing with itself. The area of the overlap between these two PEs was then used as a measure of similarity (called *conformance*) between the two implementations (see Figure 1a).

While this preliminary approach was promising and uncovered some low-conformance QUIC implementations of CUBIC and BBR, it had some limitations. For example, we found that plotting the performance envelope with a single convex hull does not provide us with sufficient granularity to identify low conformance in some cases. One such example is illustrated in Figure 1.

The maximum possible value for *conformance*, the overlap between the PEs of the reference and QUIC implementations, is 1. We can clearly see that using a single convex hull for the PE does not fully capture low conformance in the `quiche` implementation of CUBIC in Figure 1a. This is because while the overlap between the PEs in Figure 1 is relatively large, most of it comprises empty space without any data points. This will inadvertently cause us to overestimate the similarity between these implementations, even though the two implementations may exhibit very different behaviors. We address this issue by clustering the delay-throughput pairs before constructing the final PEs (see Figure 1b).

Furthermore, our earlier conformance metric did not provide any hints on how the conformance of a QUIC implementation could be improved. We found several QUIC implementations where the conformance can be improved by merely tuning their intentionally misconfigured cwnd and sending rate parameters (§5). In such cases, the PE of the reference and tested QUIC implementations generally have the same shape and the same number of clusters but are translated to a different region in the delay-throughput plane.

To identify such implementations, we propose an additional metric *Conformance post-Translation* (or *Conformance-T*), that is the maximum conformance that can be achieved by translating the PE of a QUIC CCA implementation. Generally, a high Conformance-T would indicate that an implementation's conformance can be improved significantly with simple parameter tuning. We show that this parameter tuning can be guided by the translation vector (Δ-throughput, Δ-delay).

In summary, our work advances the state of the art in our understanding of speciation [32, 35] in QUIC CCAs with the following contributions:

(1) We present a comprehensive study of 11 QUIC stacks using this enhanced definition of the PE in both controlled testbeds and on the Internet. Our results show that while most QUIC CCA implementations are conformant in shallow buffers, they become less conformant in deep buffers (§4.1). In the process, we identified five *new* QUIC implementations that had low conformance;
(2) We propose new metrics *Conformance-T*, Δ-throughput, and Δ-delay, that can provide hints on the root cause of low conformance for a QUIC implementation (§3.2);
(3) We demonstrate how low-conformance implementations can cause unfairness (§4.3) and subvert our expectations of how we expect different CCAs to interact (§4.4); and
(4) We identify implementation-level differences that led to the low conformance and propose modifications to improve conformance for three QUIC CCA implementations (§5). We were also able to identify instances where low conformance arises when features that are a part of the TCP stack and not the CCA itself (like Hystart (RFC 9406) [23]) are not implemented in QUIC stacks.

The traces and source code for all our experiments is available at [34].

Given the large number of QUIC stacks, it seems inevitable that there will be an increasing number of non-conformant QUIC CCA implementations. Our proposed metrics will provide developers

Table 1: List of QUIC/TCP stacks studied and their available CCAs.

Organization	Stack	Version/Commit Hash	CUBIC	BBR	Reno
Linux kernel	TCP	Linux 5.13.0-44-generic	✓	✓	✓
Facebook	mvfst [6]	65a9c066e742620becacc99b7c0ca86200e6a4c4	✓	✓	✓
Google	chromium [8]	82a3c71cf5bf2502d5ad90489fe20ce8f8cb3fab	✓	✓	✗
Microsoft	msquic [12]	e6110b62cd8e0d84e6436bde2504e6bc0702921a	✓	✗	✗
Cloudflare	quiche [5]	9dfeaafb625b08760218def7beb8db133e3f50cb	✓	✗	✓
LiteSpeed	lsquic [11]	108c4e7629a8c10b9a73e3d95be0a1652e620fb9	✓	✓	✗
Go	quicgo [9]	424a66389c01d10678bfb980cfe6faa8524b42b6	✓	✗	✓
H2O	quicly [10]	d44cc8b21ed0d27ab6d209d0775c3961b2f89f38	✓	✗	✓
Rust	quinn [14]	f86dd7596d4df31370b294c35501cec8da48b393	✓	✗	✓
Amazon Web Services	s2n-quic [4]	17826d9df1c59903beadd1733bbe79ed7d67647e	✓	✗	✗
Alibaba	xquic [3]	00f622885d91e02c879f8531bc04af7a584faed4	✓	✓	✓
Mozilla	neqo [13]	07c2019988a8f0a37f87cbd90f95e906e7b53258	✓	✗	✓

with a means to understand how their CCA implementations deviate from standard kernel implementations and with hints to make the required modifications to ensure that their CCA implementations are conformant, thus reducing the likelihood of mistakes that might cause instability and performance degradation to the Internet.

While we had initially set out to study how we can ensure that future QUIC CCA implementations are conformant to the standard kernel implementations, we have come to realize that the standard kernel implementations are also moving targets that will evolve with time. While it is beyond the scope of this paper, the question of how QUIC CCA implementations can keep up with new RFCs and evolving kernel implementations is also an important concern.

2 RELATED WORK

The widespread adoption of QUIC by major tech companies has inspired numerous studies in recent years [24, 26, 30, 31, 33]. These works include studies on the effectiveness of its new mechanisms, interoperability, differences in parameterization, and general performance.

Comparing gQUIC with TCP. Most of the earlier studies evaluating the performance of QUIC against TCP were done using gQUIC[31], Google's original version of QUIC before the IETF QUIC standard was released, as it was the only implementation available. Langley et al. completed an extensive study on gQUIC's performance for Google's large-scale deployment of QUIC and reported gQUIC's performance against TCP CUBIC [31]. Langley et al. reported that gQUIC outperformed TCP in metrics such as Google Search's latency, Youtube's video latency, and Youtube's video rebuffer rate. For example, YouTube's video rebuffer rate was reduced by 18.0% for desktop users and 15.3% for mobile users when TCP was replaced with gQUIC. A limitation of this study is that it measured the performance of specific applications (and not QUIC CCA implementations) and the results were obtained from proprietary data that is not easily reproducible. In our work, we study the general transport-layer performance of CCA implementations for a large number of QUIC stacks and not just gQUIC. Our source code is available at [34] and our experiments are fully reproducible.

Other gQUIC studies evaluating gQUIC's performance against TCP mostly used page-load time measured via controlled experiments as the main metric to compare their application-layer performance [24, 26, 33]. Carlucci et al. found that gQUIC achieved higher goodput and smaller page-load times in networks with small buffers or high packet loss rates [26]. Biswal et al. found that gQUIC outperformed TCP in networks with low bandwidth, high RTT, or high packet loss rates. Megyesi et al. had similar conclusions, except that they reported, to the contrary, that TCP performs better in networks with high packet loss rates. These studies all evaluated the QUIC's protocol performance against TCP using application-level metrics without any attempt at root-cause analysis. Our work is focused on the transport-layer performance of the QUIC CCAs and we show that our methodology provides hints that allow us to (i) deduce potential reasons for differences in behavior, and (ii) to verify if modifications made would make a QUIC CCA implementation more conformant to the reference kernel implementation.

Another study on gQUIC by Kakhki et al. focused on evaluating gQUIC's transport-layer performance against TCP [30]. In their study, Kakhki et al. performed a root-cause analysis using execution traces captured by instrumenting gQUIC's source code. They discovered that gQUIC's default parameter values were not tuned and proceeded to calibrate their gQUIC implementation in their experiments so that they would perform similarly to those deployed in production. Kakhki et al. highlighted that this calibration was not done in prior studies and led to them wrongly concluding that gQUIC would under-perform in high bandwidth networks.

Kakhki et al. observed that the congestion control implementation in QUIC was likely different from TCP despite both of them implementing the same CUBIC CCA because the gQUIC flow was observed to have twice the bandwidth of a competing CUBIC flow at the same bottleneck link. They concluded that this is because gQUIC's CUBIC increased its congestion window (cwnd) more frequently and by a larger amount than TCP CUBIC. Although Kakhki et al. showed that there was a deviation and even found the root cause, those findings are not relevant today as the QUIC standard published by IETF (IETF QUIC) has many significant differences compared to the old gQUIC standard. Moreover, the QUIC ecosystem has grown much larger and includes many more stacks. At some level, this limitation also extends to the results of the above studies. To the best of our knowledge, our work is the most comprehensive measurement study covering the largest number of open-sourced QUIC implementations to date.

Evaluating IETF QUIC implementations. There are other more recent studies that evaluated IETF QUIC stacks, but most of them have limited scope, and focus on application-layer metrics [38,

40, 42]. Saif et al. compared the `quiche` stack against TCP and found that `quiche` QUIC has greater average throughput but worse user quality of experience metrics as measured by the Lighthouse tool [7, 42]. A key limitation of these studies is that they only report the results for a single QUIC stack. As demonstrated in our latest study, QUIC's performance can differ significantly across implementations and these differences are often artifacts of the implementations and not a result of the QUIC protocol.

To the best of our knowledge, Marx et al. were the first to report speciation in different QUIC stacks [32]. Their study highlighted differences in implementation details of 15 IETF QUIC stacks. These differences were uncovered by analyzing the inner workings of the QUIC stacks through visualizations produced by the Qvis tool. They found significant differences in domains where the QUIC standard had minimal specifications, i.e. congestion control and flow control. Differences were found even for parameters that were specified in the QUIC standard. For example, 3 of the stacks did not follow the QUIC standard's initial congestion window value specification and 10 of the stacks did not follow the QUIC standard's recommendation of 2 for the acknowledgment frequency. Marx et al. did not investigate how the performance differences arose from these implementation deviations. More recently, we had earlier reported significant deviations between the different QUIC implementations of existing congestion control algorithms and their respective kernel implementations for 4 common QUIC stacks [35]. In this paper, we build upon our earlier work by investigating the transport-layer performance differences for a much larger number of modern QUIC stacks.

3 METHODOLOGY

Our goal is to quantify the conformance of QUIC implementations of CUBIC, BBR, and Reno for the QUIC stacks in Table 1 using an enhanced definition of the Performance Envelope (PE). In this section, we describe our original definition of the PE [35], our proposed enhancements, as well as the new metrics Conformance-T, Δ-throughput, and Δ-delay.

3.1 Background: Performance Envelope

To determine the PE for a test implementation, a flow running the test implementation is launched alongside a competing flow that runs the corresponding reference (Linux kernel TCP) implementation. Both flows are set to the same RTT and run through a bottleneck with a constant link capacity and a fixed-size droptail buffer. The flows run for 120 seconds to ensure that they have sufficient time to converge to steady state. The start and end of the flow traces are also truncated by 10% to remove the transient behaviors. The throughput and delay time series data (computed offline via packet trace) of the test implementation is then sampled every 10 RTTs and plotted pair-wise (d, T) on a delay-throughput plane as a point cloud. The region defined by the convex hull of this point cloud is the PE for the implementation. Empirically, we have found that sampling the time-series throughput and delay data at this rate is sufficient to capture an implementation's PE. In other words, sampling more frequently does not substantially affect the shape of the PE for a CCA implementation.

Conformance. The *conformance* of an implementation is determined by calculating the overlap its PE has with the PE of the reference implementation. The overlap is weighted by the number of points present in the overlapping region. In particular,

$$\text{Conformance} = \frac{\text{\# of points in the overlapping region}}{\text{total \# of points in both PEs}}$$

Clearly, the maximum possible value of conformance is 1 (complete overlap) and the minimum value is 0 (no overlap).

3.2 Improving the Performance Envelope

We propose a number of enhancements to the earlier definition of the PE from [35] to address some of its limitations.

Handling outliers. Since the data points that are used to determine the PE are instantaneous values, there will be outliers. Earlier, we removed these outliers by eliminating 5% of the points with the largest Euclidean distance from the centroid of the PE. However, we found that there is no guarantee that the points furthest from the centroid are necessarily outliers and by doing so, we risk artificially reducing the variance in the PE. Ideally, we want to remove the outliers that are points that arise from natural network variation across trials and not artifacts of the implementation itself. Therefore, we decided that a more principled way to remove outliers was to capture (d, T) pairs over multiple trials and use the intersection of the convex hulls produced by all these trials to be the final PE. In practice, it turns out that our approach also removes roughly 5% of the points on average for our experiments.

One convex hull is not enough. From Figure 1, we can see that the distribution of the points in the point cloud is often not uniform. Hence, if we use only a single convex hull, it is plausible that we may include large regions of empty space that do not contain any points. In other words, using only a single convex hull will often result in the overestimation of an implementation's conformance. To address this issue, instead of a single convex hull, we use a clustering algorithm to group data points into clusters and then calculate convex hulls for each individual cluster. The final PE is then the set of all convex hulls.

How many clusters is enough? We use the standard k-means clustering algorithm [29] to compute clusters from our set of data points in the throughput-delay plot. Usually, the number of clusters for the k-means algorithm is determined using the elbow method, which selects the inflection point for the mean squared error of the resulting clusters. However, in our case, we found that the regular elbow method was not satisfactory because there was no obvious inflection point if we considered the mean squared error.

On the other hand, we can see in Figures 2 and 3 that a PE often has a "natural" number of clusters arising from the characteristics of the CCA. For BBR, this natural number is generally 2 (because of its distinct `ProbeBW` and `ProbeRTT` phases, see Figure 2). For CUBIC and Reno, natural clusters still exist, but there does not seem to be a fixed number (see Figure 3).

In order to determine this "natural" number of clusters algorithmically, we ran the k-means algorithm for all $k \geq 1$ for each trial. For each k, each trial will produce a PE with k convex hulls. The final PE for each k is then computed as the intersection for all the convex hulls over all the trials. For each of these PEs, we compute and plot the intersection over union (IOU) R, which we define as

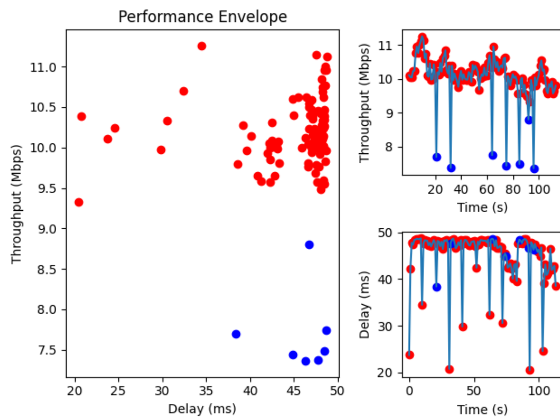
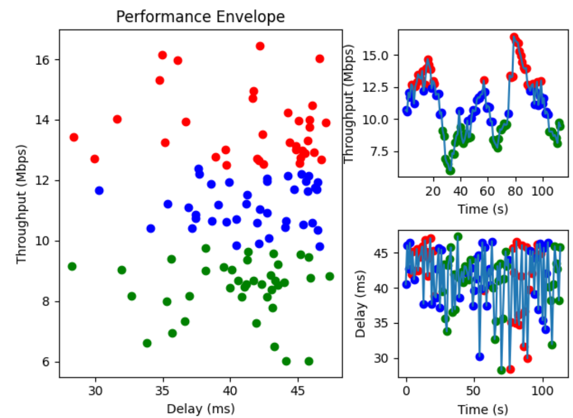
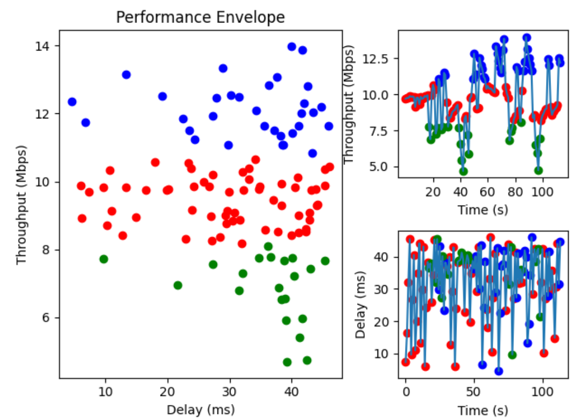


Figure 2: Two distinct clusters corresponding to TCP BBR's `ProbeBW` (red) and `ProbeRTT` (blue) phases.



(a) TCP CUBIC



(b) TCP Reno

Figure 3: Clusters for CUBIC and Reno are less distinct and tend to form around different throughput levels.

the proportion of the total data points for all the trials contained in the PE. This is effectively the amount of information retained in the PE for each value k.

What we found was that R is a strictly decreasing function of the number of clusters k, since as the number of clusters increases, the size of each cluster becomes smaller and the final intersection will contain fewer data points. Because it was indeed true that there was a "natural" number of clusters for each implementation, we found that R drops most steeply at some k for all the instances that we studied. We use the value of k before the drop as the number of clusters for the final PE. We illustrate this iterative process in Figure 4, with the blue and red point clouds representing data points from two trials of the same measurement. We removed the outliers from each trial *before* we computed the number of clusters for a PE so that the outliers do not impact the number of clusters in the PE for a CCA implementation.

3.3 Improving the Conformance Metric

Other than measuring the conformance of different CCA implementations more accurately, we also want a way to deduce how it might be possible to improve the conformance of an implementation via simple parameter tuning. Modern congestion control algorithms and QUIC stacks can be quite complex, and therefore manually checking and tuning all their parameters is not tractable. What we need are hints for the implementor on what tuning might be needed to improve a CCA implementation's conformance. We found many instances where this is possible. In such cases, the PE of an implementation generally has the same shape and number of clusters as the reference implementation but is merely translated to another region in the delay-throughput plane.

Conformance-T. To identify implementations that can likely be fixed with parameter tuning, we compute the translation that is necessary so that we maximize the intersection between the respective clusters of data points and recalculate the *Conformance post-Translation* (or *Conformance-T*). In general, a high Conformance-T indicates that an implementation's conformance can be significantly improved through simple parameter tuning alone.

To understand how Conformance-T works, consider the following experiment: we know that BBR multiplies its BDP estimate with a constant called `cwndgain` to determine its cwnd. By default, `cwndgain` is set to 2 in the Linux kernel. We modified the kernel version of BBR by changing its `cwndgain` and measured the Conformance and Conformance-T values for modified implementations with a range of `cwndgain` values from 1.0 to 4.0. We plot the resulting values for Conformance-T in Figure 5. Unsurprisingly, Conformance and Conformance-T are highest when `cwndgain` is 2.0. As the gap in `cwndgain` in the modified implementation increases, the Conformance drops as expected, even though the algorithmic behavior of the modified version is almost identical to that of the vanilla kernel BBR implementation. On the other hand, the Conformance-T remains relatively high. This suggests that Conformance-T is a relatively robust way to capture shifts in observed behavior arising from parameter tuning.

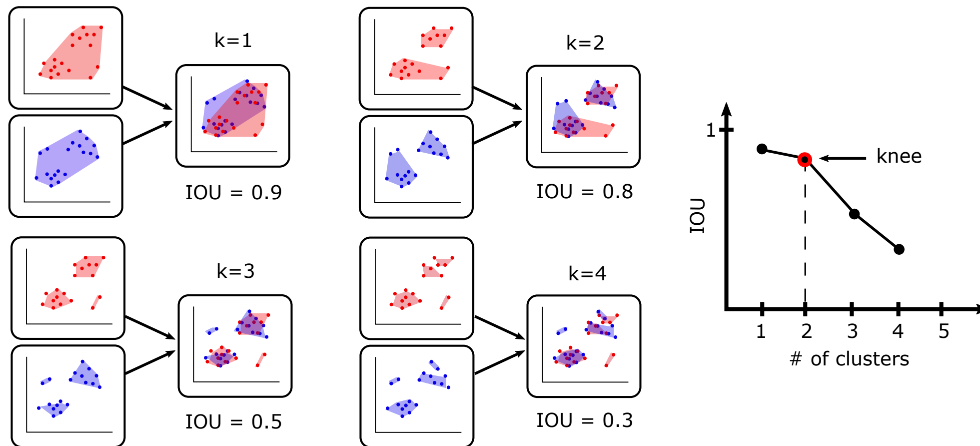


Figure 4: Determining k, the number of clusters for a Performance Envelope. IOU = Intersection over Union.

Parameter tuning. It turns out that the translation required to compute Conformance-T also provides us with hints on the systematic difference between a QUIC implementation and its corresponding reference implementation. We capture the 2 components in the required translation as the translation vector (Δ-throughput, Δ-delay).

Consider the two *knobs* a congestion control algorithm usually uses to regulate how aggressive it is: (i) its sending rate and (ii) its cwnd. For QUIC implementations showing low conformance but a high value of Conformance-T, we can deduce which of these knobs have been improperly tuned and correct for them. For example, if the cwnd of an implementation is more than it should be, it would typically have higher throughput and higher delay since it puts more packets in flight. This would show up as a large positive Δ-throughput and a large positive Δ-delay. We see this trend in Figure 5 where we see both Δ-throughput and Δ-delay increase as the cwndgain is increased.

On the other hand, if the implementation sets a correct cwnd but sends its packets at a larger sending rate than it should, we would see a large positive Δ-throughput but a negligible increase in Δ-delay. We see this behavior in mvfst BBR (See Figure 9) which we had earlier identified to have set its pacing gain to a value higher than the default [35].

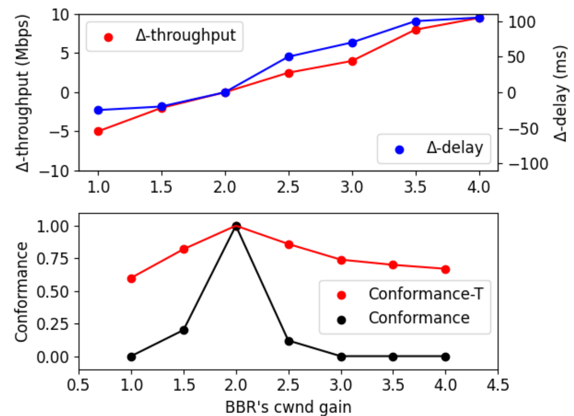


Figure 5: Conformance and Conformance-T values for modified versions of TCP BBR.

3.4 Experiment Setup

Our experiments were conducted on a testbed with two Linux machines (Ubuntu 20.04, kernel version 5.13.0-44-generic) connected via a 1 Gbps Ethernet cable. To generate the QUIC flows, we installed the open-source QUIC stacks on both machines and used the test clients/servers provided. To generate the TCP flows, we used the iperf3 [2] tool.

We used a modified version of our open-source tool QUIC Bench [34] to run our experiments and compute our new metrics Conformance-T, Δ-throughput, and Δ-delay. We configured the socket buffer sizes for both UDP and TCP to be 12,582,912 bytes in order to have a fair comparison between TCP and QUIC.

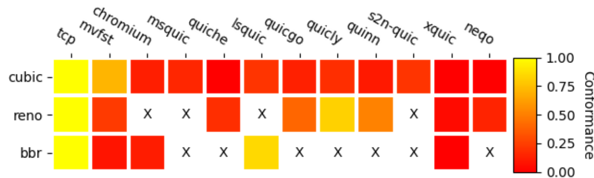
As shown in Table 2, there are currently at least 22 modern QUIC stacks [1] available. However, in this paper, we only evaluated 11 of them. We picked these 11 stacks because they are either used by major public companies, such as Google [8], Facebook [6], and Microsoft [12] or are the de-facto standard QUIC libraries in programming languages such as Go [9] or Rust [14]. The remaining stacks that we did not evaluate are either closed-sourced, had no stable version available, or did not implement congestion control.

4 MEASUREMENT RESULTS

In this section, we present the results of our evaluation of the 11 QUIC stacks listed in Table 1. In §4.1, we compare the PEs of the QUIC implementations of CUBIC, Reno, and BBR to their corresponding Linux TCP implementations (hereafter referred to as the reference implementations) and calculate their conformance with respect to these reference implementations.

Table 2: List of Known IETF QUIC/TCP stacks.

Organization	Stack	Open Source?	Implements CCA?	Stable?	Deployed?	Evaluated?
Facebook	mvfst [6]	✓	✓	✓	✓	✓
Google	chromium [8]	✓	✓	✓	✓	✓
Microsoft	msquic [12]	✓	✓	✓	✓	✓
Cloudflare	quiche [5]	✓	✓	✓	✓	✓
LiteSpeed	lsquic [11]	✓	✓	✓	✓	✓
Go	quicgo [9]	✓	✓	✓	✓	✓
H2O	quicly [10]	✓	✓	✓	✓	✓
Rust	quinn [14]	✓	✓	✓	✓	✓
Amazon Web Services	s2n-quic [4]	✓	✓	✓	✓	✓
Alibaba	xquic [3]	✓	✓	✓	✓	✓
Mozilla	neqo [13]	✓	✓	✓	✓	✓
Akamai	akamaiquic [16]	✗	-	-	-	✗
Apple	applequic [39]	✗	-	-	-	✗
Apache	ats [17]	✓	✓	✓	✗	✗
F5	f5 [27]	✓	✗	✗	✗	✗
Haskell	haskellquic [17]	✓	✗	✗	✗	✗
Java	kwik [18]	✓	✗	✗	✗	✗
nghttp	ngtcp2 [20]	✓	✗	✗	✗	✗
nginx	nginx [19]	✓	✗	✗	✗	✗
Pico	picoquic [21]	✓	✓	✗	✗	✗
Python	aioquic [15]	✓	✗	✓	✓	✗
Quant	quant [22]	✓	✓	✗	✗	✗



(a) 5 BDP (deep) buffer

(b) 1 BDP (shallow) buffer

Figure 6: Conformance becomes significantly worse in 5 BDP (deep) buffers. (10 ms RTT, 20 Mbps)

To investigate the general fairness between different implementations, we also performed a bandwidth-share-based analysis of all pairwise combinations of the 11 QUIC stacks. We present the results in §4.3. We also discuss how we can expect low-conformance QUIC implementations to subvert our expectations of how CUBIC and BBR interact in §4.4.

All evaluations were done under a variety of network conditions that were emulated by varying the network parameters as follows:

(1) RTT (10 ms and 50 ms);
(2) bottleneck bandwidth (20 Mbps and 100 Mbps); and
(3) bottleneck buffer size (0.5, 1, 3, 5 times the BDP)

Table 3: Summary of low-conformant implementations (1 BDP Buffer).

Stack	Type	Conf-old[a]	Conf	Conf-T	Δ-tput	Δ-delay
chromium[b]	CUBIC	0.65	0.6	0.74	+3 Mbps	0 ms
neqo	CUBIC	0	0	0.62	−6 Mbps	−5 ms
quiche	CUBIC	0.48	0.08	0.55	+5.5 Mbps	0 ms
xquic	CUBIC	0.6	0.55	0.64	0 Mbps	−5 ms
mvfst[b]	BBR	0	0	0.7	+9 Mbps	0 ms
xquic	BBR	0.37	0.15	0.42	+4 Mbps	0 ms
xquic	Reno	0.43	0.38	0.81	−4 Mbps	−3 ms

[a] Conformance calculated using our earlier definition in [35].
[b] Earlier identified and fixed in [35].

To ensure that buffer sizes are comparable across all the RTT and bottleneck bandwidth combinations, we normalize them as multiples of the Bandwidth-Delay Product (BDP). All network parameters were set using *tc* and Mahimahi [37] and each experiment was repeated 5 times.

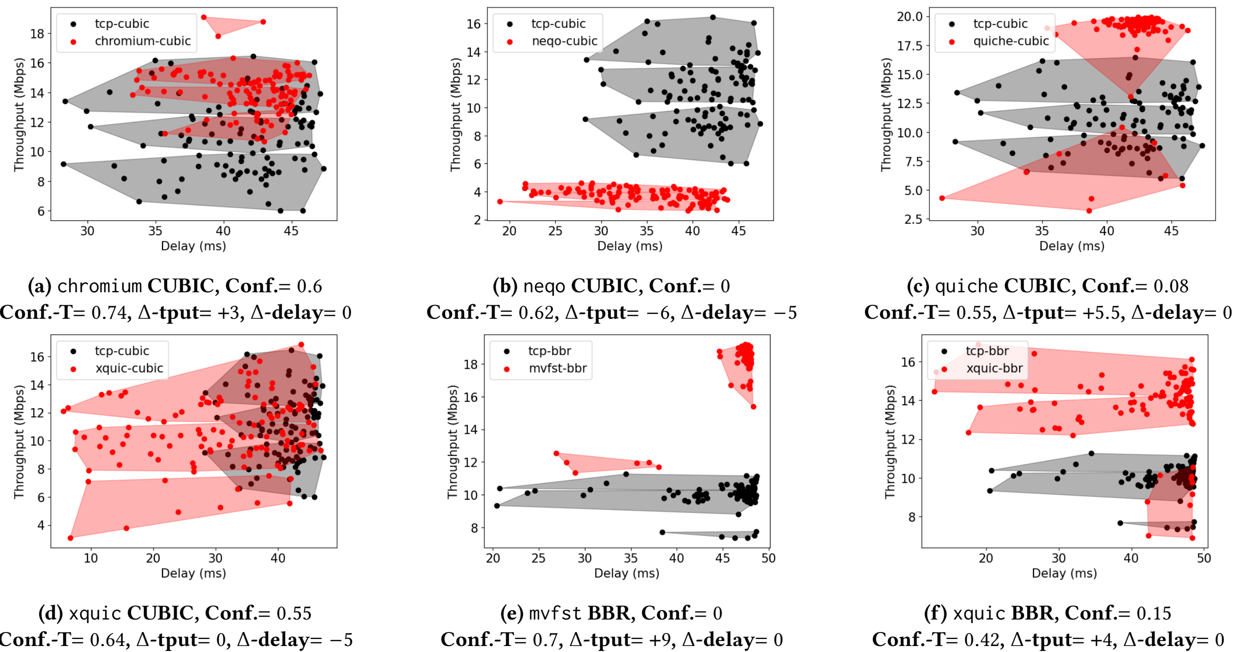
We note here that all our Performance Envelope and conformance measurements are done over relatively stable network profiles with constant bottleneck bandwidths and simple droptail buffers. This is because while trying to understand how well QUIC implementations of standard CCAs resemble their kernel counterparts, we want any deviations to arise from the implementation and the QUIC stack itself, and not from the inherent variability of the network profile. It is for this reason that we evaluate the Performance Envelope with simple 2-flow experiments without any background traffic.

4.1 Conformance of CCA implementations of mainstream QUIC stacks

As we can see in Figure 6, the bottleneck buffer size has a significant impact on conformance. In particular, we can see from Figure 6a, that all implementations have poor conformance in 5 BDP (deep) buffers. Since this is a trend consistent across all stacks, it is plausible that the root cause of low conformance in deeper buffers is some artifact of the QUIC standard that becomes more pronounced when the buffers are larger. The investigation of this hypothesis remains as future work.

In general, the majority of the stacks are relatively conformant in shallow buffers as shown Figure 6b. There are several outliers to this trend, with some implementations showing very low conformance (<0.5) even in shallow 1 BDP buffers. From Figure 6b, we can identify the 7 low-conformant stacks in red. The results for these stacks (in 1 BDP buffers) are summarized in Table 3. xquic CUBIC is included in the list, despite having conformance marginally greater than 0.5, because it was found to be extremely unfair to other implementations (§4.3).

Two of the 7 implementations in Table 3 (chromium CUBIC and mvfst BBR) were earlier identified to be low-conformant and modifications were proposed to make them more conformant [35]. This means that we have identified 5 *new* non-compliant QUIC implementations. While these new non-conformant implementations were not evaluated in our earlier study [35], it is clear that the enhanced definition of conformance presented in this paper was



(a) `chromium` **CUBIC, Conf.**= 0.6
Conf.-T= 0.74, **Δ-tput**= +3, **Δ-delay**= 0

(b) `neqo` **CUBIC, Conf.**= 0
Conf.-T= 0.62, **Δ-tput**= −6, **Δ-delay**= −5

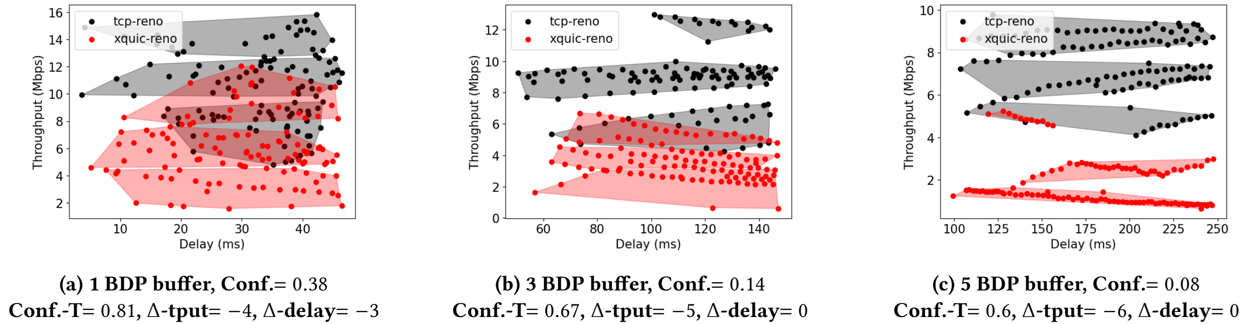
(c) `quiche` **CUBIC, Conf.**= 0.08
Conf.-T= 0.55, **Δ-tput**= +5.5, **Δ-delay**= 0

(d) `xquic` **CUBIC, Conf.**= 0.55
Conf.-T= 0.64, **Δ-tput**= 0, **Δ-delay**= −5

(e) `mvfst` **BBR, Conf.**= 0
Conf.-T= 0.7, **Δ-tput**= +9, **Δ-delay**= 0

(f) `xquic` **BBR, Conf.**= 0.15
Conf.-T= 0.42, **Δ-tput**= +4, **Δ-delay**= 0

Figure 7: QUIC CUBIC and BBR implementations with low conformance for 1 BDP buffers.



(a) 1 BDP buffer, Conf.= 0.38
Conf.-T= 0.81, **Δ-tput**= −4, **Δ-delay**= −3

(b) 3 BDP buffer, Conf.= 0.14
Conf.-T= 0.67, **Δ-tput**= −5, **Δ-delay**= 0

(c) 5 BDP buffer, Conf.= 0.08
Conf.-T= 0.6, **Δ-tput**= −6, **Δ-delay**= 0

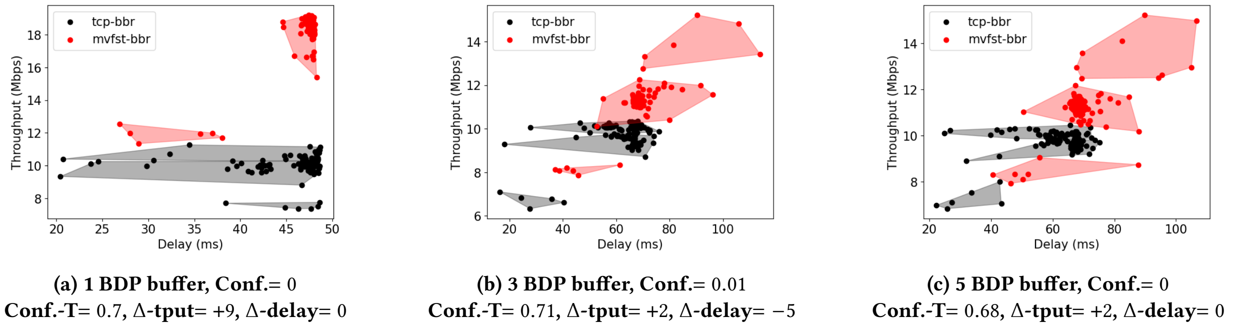
Figure 8: Performance envelopes for `xquic` Reno for different bottleneck buffer sizes.

helpful in highlighting these variants. Comparing our new definition of conformance to the earlier definition (*Conformance-old*) in Table 3, we see that our new definition makes the low conformance for some QUIC implementations more obvious (cf. `quiche` CUBIC and `xquic` BBR).

In §5, we describe modifications that can make `xquic` BBR and `quiche` CUBIC more conformant; for `xquic` CUBIC, we were able to identify the root cause for the low conformance. The PEs for non-compliant CUBIC and BBR QUIC implementations are shown in Figure 7 and the PEs for the sole non-compliant Reno implementation are plotted in Figure 8. While we performed measurements over a large variety of network configurations as described in §4, link speed and RTT had a marginal impact on the results and so we only produce representative plots varying the buffer sizes unless stated otherwise. It is likely that link speed and RTT do not have a pronounced effect because we normalize our buffer sizes by the BDP in our relatively stable network profiles. This trend may not hold in networks with highly volatile bandwidth variations, like 5G networks.

4.1.1 CUBIC. The implementations of CUBIC in `chromium`, `neqo`, `quiche`, and `xquic` had low conformance. We had earlier found the modifications needed to make `chromium` CUBIC more conformant [35]. To build on our earlier work, we describe how we can mitigate the low conformance for `quiche` CUBIC and `xquic` CUBIC in §5.

4.1.2 BBR. We found BBR implementations in `mvfst` and `xquic` to have low conformance. The conformance for `mvfst` was better for deep buffers (see Figure 9), while for `xquic`, the lack of conformance became worse in deep buffers (see Figure 10). However, both these

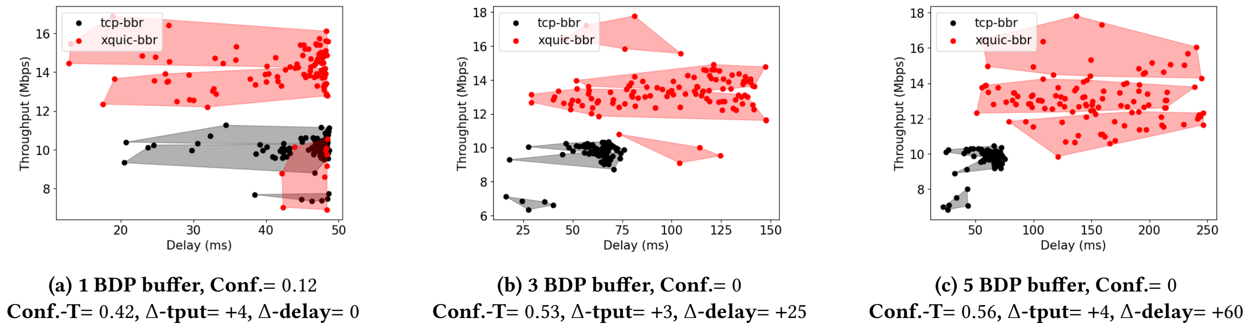


(a) 1 BDP buffer, Conf.= 0
Conf.-T= 0.7, **Δ-tput**= +9, **Δ-delay**= 0

(b) 3 BDP buffer, Conf.= 0.01
Conf.-T= 0.71, **Δ-tput**= +2, **Δ-delay**= −5

(c) 5 BDP buffer, Conf.= 0
Conf.-T= 0.68, **Δ-tput**= +2, **Δ-delay**= 0

Figure 9: Performance envelopes for `mvfst` BBR. (Conf.=Conformance)



(a) 1 BDP buffer, Conf.= 0.12
Conf.-T= 0.42, **Δ-tput**= +4, **Δ-delay**= 0

(b) 3 BDP buffer, Conf.= 0
Conf.-T= 0.53, **Δ-tput**= +3, **Δ-delay**= +25

(c) 5 BDP buffer, Conf.= 0
Conf.-T= 0.56, **Δ-tput**= +4, **Δ-delay**= +60

Figure 10: Performance envelopes for `xquic` BBR. (Conf.=Conformance)

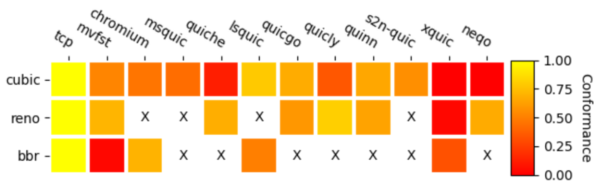


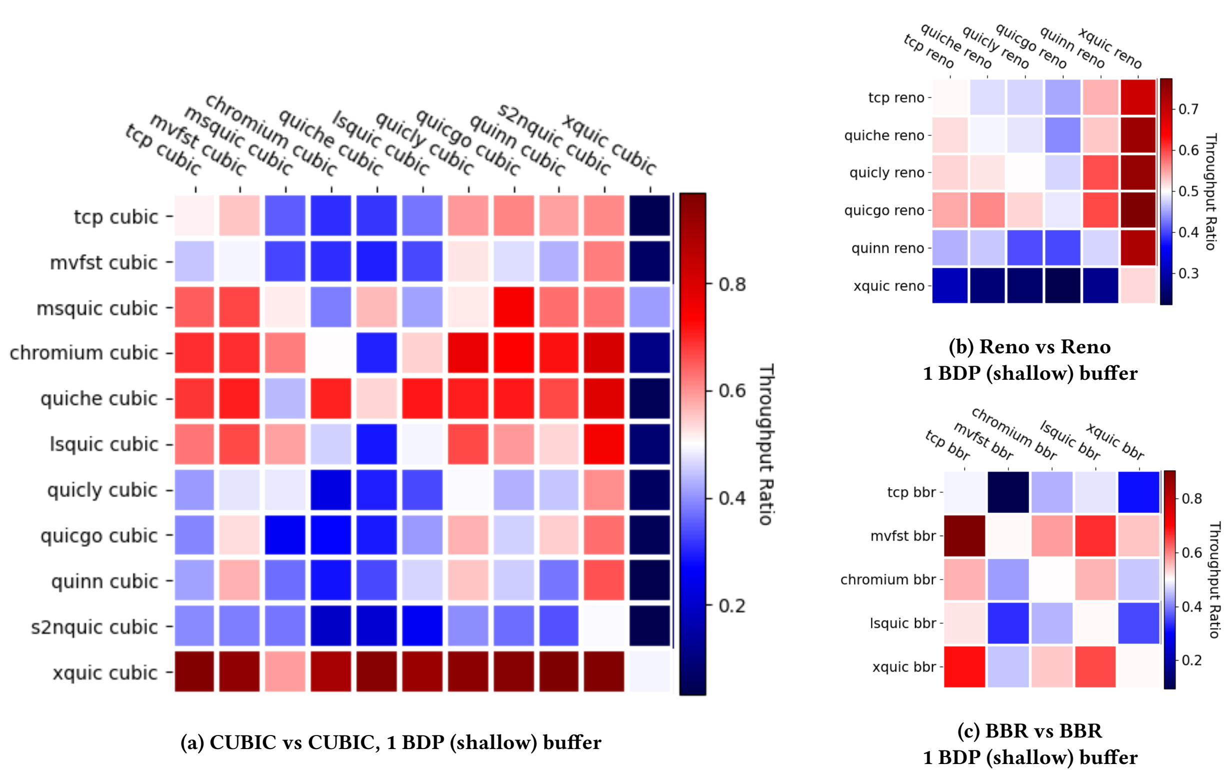
Figure 11: Conformance of various QUIC stacks when tested on AWS. Link speed was locally limited to 100 Mbps.

implementations show significantly high Conformance-T values. The positive Δ-throughput for both these implementations suggests that they might be reasonably conformant implementations of BBR with some parameter tuning. We had earlier highlighted that `mvfst` BBR multiplies its final sending rate by 120% in order to improve throughput [35]. We verified that reducing the send rate to 100%, `mvfst` BBR will become more conformant. We show in §5 that a similar modification can also improve the conformance of `xquic` BBR.

4.1.3 Reno. QUIC Reno implementations are generally conformant for most QUIC stacks. The conformance in deeper buffers is also relatively better than CUBIC and BBR. This is likely because Reno is the simplest algorithm among the three CCAs investigated, and is thus easier to implement correctly. The only exception among them is `xquic`, as shown in Figure 8. The fact that even a simple CCA like Reno is non-conformant for `xquic` suggests that there might be a larger issue with the `xquic` stack itself, given that all its CCA implementations show poor conformance. Upon investigation, we could not find anything that was clearly wrong with the `xquic` CCA implementations, suggesting that the root cause was beyond algorithmic parameters and correctness of the CCA implementation. Determining the exact cause of the observed differences remains future work.

4.2 Investigating Conformance "in the Wild"

We repeated our experiments in §4.1 on the Internet. In these measurements, the senders were run on `aws` instances and connected to receivers on physical servers in our lab. We limited the link speed to 100 Mbps at the server. We measured the ping latency before every experiment and added additional delay using Mahimahi [37] to keep the RTT constant at 50 ms across all trials. Like before, the results of these experiments are plotted as a heatmap in Figure 11. We found the conformance numbers to be similar to our results for 1 BDP buffer in our testbed (see Figure 6b). While we were tempted to conclude that this hints that the buffers on the Internet are shallow, we refrain from making any such claim since a CCA's performance on the Internet can be impacted by other network artifacts as well.

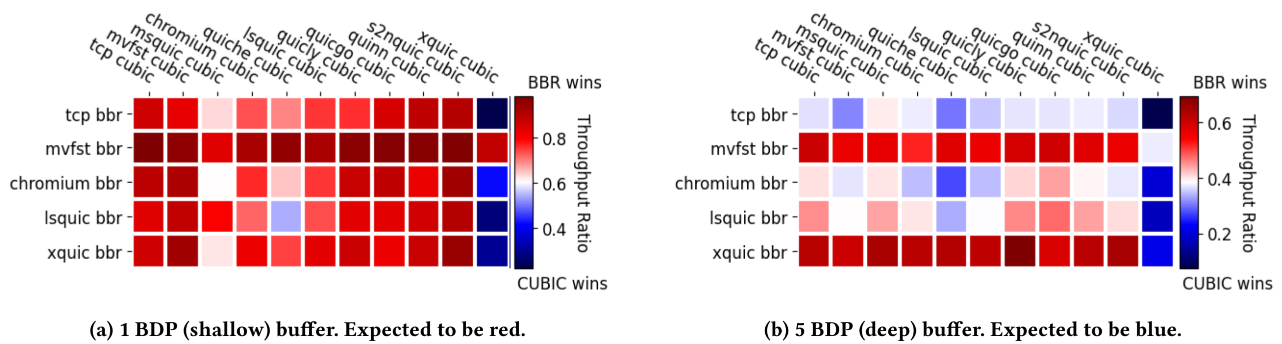


(a) CUBIC vs CUBIC, 1 BDP (shallow) buffer

(b) Reno vs Reno 1 BDP (shallow) buffer

(c) BBR vs BBR 1 BDP (shallow) buffer

Figure 12: Throughput ratios for competing implementations on CUBIC, Reno, and BBR (20 Mbps, 50ms RTT).



(a) 1 BDP (shallow) buffer. Expected to be red.

(b) 5 BDP (deep) buffer. Expected to be blue.

Figure 13: Different implementations of CUBIC and BBR competing with each other (a throughput ratio of 1 means the BBR flow starves the CUBIC flow.)

4.3 Fairness between Implementations

The PE only compares the time series behavior of QUIC implementations to their kernel counterparts, which can provide us an estimate of how faithfully QUIC CCA implementations conform to the reference implementations. However, no matter how hard developers might try to reproduce the behaviors of the kernel implementations, there will be overheads, since the QUIC stack CCA implementations run in the user space. There will also likely be some inherent unfairness between algorithms. It is also inevitable that some organizations will want to modify and tune their congestion control algorithms for their own applications and therefore lead to low conformance by design.

For such cases, we would still be keen to ensure that new implementations can co-exist with other kernel and QUIC CCA implementations without causing instability or significant degradation. To determine if new implementations are friendly to other implementations, we do a simple bandwidth-share-based analysis of the pairwise combinations of all 23 QUIC CCA implementations investigated in this paper. Moreover, this serves as an extension to

the conformance analysis presented in §4.1 and a sanity check that high conformance is highly correlated to fairness to other implementations.

In these experiments, the two competing flows share a 20 Mbps bottleneck link, 50 ms RTT, and a 1 BDP buffer. The bandwidth share is computed as $\frac{T_x}{T_x+T_y}$ and $\frac{T_y}{T_x+T_y}$ where T_x and T_y are the throughputs of the two competing flows averaged across 5 trials. If the bandwidth share is greater than 0.5 for any flow, it implies that the flow has more than a fair share of the bandwidth.

Since it is well known that different congestion control algorithms can be unfair to each other [36, 45], we looked into fairness between implementations of the same CCAs. We plot the throughput ratios for competing implementations of CUBIC, BBR, and Reno in Figure 12. If we compare these results to the implementations earlier identified as being low-conformant in Table 3, it is not surprising to see which QUIC implementations of CUBIC, Reno, and BBR are overly aggressive. `chromium` CUBIC, `quiche` CUBIC, and `xquic` CUBIC (all previously identified low-conformant implementations of CUBIC) were unfair to all other implementations of CUBIC.

Similar trends exist for `xquic` BBR, `mvfst` BBR, and `xquic` Reno, which also show low conformance. We also note that `lsquic` CUBIC also shows some degree of unfairness despite having high conformance (0.76). This suggests that while low conformance is likely to lead to unfairness, high conformance does not necessarily result in fairness, so it is still important to do a bandwidth-share-based analysis for new QUIC CCA implementations.

4.4 Contradicting known trends in inter-CCA fairness

Given the current heterogeneous congestion control landscape on the Internet, it is important for interactions between different congestion control algorithms to be consistent and predictable. In particular, the interactions between CUBIC and BBR, the two most dominant congestion control algorithms on the Internet, has been a hot topic in congestion control research over the past few years [36, 44, 45].

In particular, it is well known that BBR will achieve higher bandwidth than CUBIC when they compete in shallow buffers due to CUBIC backing off frequently and BBR being largely loss-agnostic. Also, CUBIC is expected to achieve higher throughput than BBR in deep buffers since CUBIC is a buffer-filler [36, 45]. In other words, Figure 13a is expected to be all red, and Figure 13b is expected to be all blue.

However, we see in Figure 13 that some QUIC implementations of CUBIC and BBR do not conform to these expectations. In particular, we see that in shallow buffers `xquic` CUBIC outperforms most BBR implementations (Figure 13a); in deep buffers, `xquic` BBR and `mvfst` BBR outperforms other CUBIC implementations. All three of these implementations were earlier identified as showing very low conformance (Table 3). This shows that in addition to introducing unfairness, low-conformant implementations can potentially subvert our expectations of how we expect standard congestion control algorithms to interact.

5 "FIXING" LOW-CONFORMANCE IMPLEMENTATIONS

QUIC CCA implementations might not behave like their Linux TCP counterparts for a number of reasons: (i) the implementation might not conform to existing standards; (ii) the algorithm parameters might be set differently; or (iii) because of implementation artifacts within the QUIC stack. As we discussed in §3.3, our new proposed metrics Δ-throughput (abbreviated as Δ-tput), and Δ-delay are often helpful in providing us with hints on the root cause of low conformance. In this section, we describe how we managed to improve the conformance of some of the low-conformance implementations identified earlier in §4. Most of our modifications required only a small number of lines of code (LoC). We summarize our findings in Table 4.

Differences in implementation. We had earlier proposed modifications to make `chromium` CUBIC and `mvfst` BBR more conformant [35]. We verified that these modifications are valid using our enhanced definition of the PE and Conformance. In addition, we also found modifications that could make `xquic` BBR and `quiche` CUBIC more conformant.

The conformance-T value for `xquic` BBR was almost triple its conformance, which suggested that `xquic` BBR could potentially be made significantly more conformant by parameter tuning. Upon investigation, we found its `cwndgain` was set to 2.5 instead of the RFC-recommended value of 2. By setting the `cwndgain` to 2, we were able to marginally improve conformance as shown in Figure 14.

After reviewing the implementation of `quiche` CUBIC, we discovered that RFC 8312 [41] was implemented. This draft proposes the rolling back of any back-off in the `cwnd` if a packet loss was deemed to be spurious. This mechanism has in fact not yet been implemented in the Linux kernel. When we disabled it in `quiche` CUBIC, we saw an immediate improvement in its conformance from 0.08 to 0.55 as shown in Figure 15. In general, we would expect QUIC CCA implementations to lag behind developments in the kernel. In this case, we have found a QUIC CCA implementation that leads kernel development.

Missing Mechanism. When we analyzed the implementation of `xquic` CUBIC, we found that TCP HyStart (RFC 9406) [23] was not implemented. TCP HyStart is a mechanism present in the Linux kernel that implements a modified slow start for CUBIC, where we will exit Slow Start when we see an increase in end-to-end delay. This makes the Hystart dramatically less aggressive than the traditional slow start. To verify that this missing mechanism was the main cause of low conformance, we evaluated the conformance of `xquic` CUBIC with respect to TCP CUBIC with HyStart disabled. As shown in Table 4, the conformance was indeed much higher. We did not attempt to implement HyStart in `xquic` CUBIC to make it more conformant because HyStart is relatively complicated and we had already identified the root cause of low conformance.

Indications of wider stack-level issues. When we reviewed the implementations of `xquic` Reno and `neqo` CUBIC, we found them to be compliant with the standard algorithms. The parameter settings are also correct. This suggests that the low conformance is likely due to some artifact(s) in the QUIC stack rather than in the CCA implementation. This means that `xquic` developers need

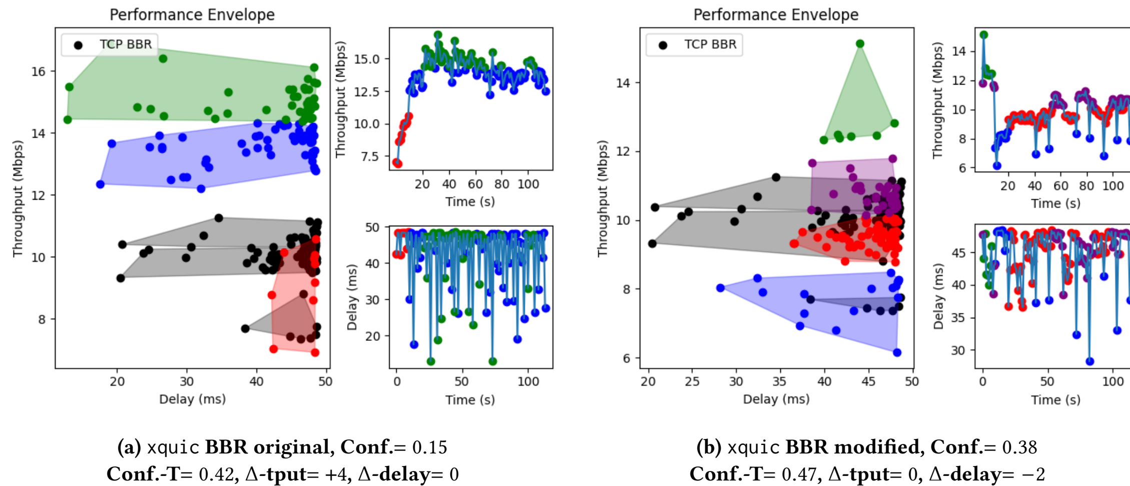
Table 4: Summary of successful modifications to low-conformant implementations (1 BDP buffer).

			Original implementation				Modified implementation					
Fixed?	Stack	Type	Conf	Conf-T	Δ-tput	Δ-delay	Conf	Conf-T	Δ-tput	Δ-delay	ΔLoC[#]	Remarks
✓	`chromium`[+]	CUBIC	0.6	0.74	+3 Mbps	0 ms	0.78	0.85	0 Mbps	3 ms	1	Emulated flows reduced from 2 to 1
✓	`mvfst`[+]	BBR	0	0.7	+9 Mbps	0 ms	0.8	0.8	0 Mbps	0 ms	2	pacing gain reduced from 1.25 to 1
✓	`xquic`	BBR	0.15	0.42	+4 Mbps	0 ms	0.38	0.47	0 Mbps	-2 ms	2	cwnd gain reduced from 2.5 to 2
✓	`quiche`	CUBIC	0.08	0.55	+5.5 Mbps	0 ms	0.55	0.66	+2 Mbps	0 ms	14	Disabled RFC8312 [46]
✗[*]	`xquic`	CUBIC	0.55	0.64	0 Mbps	-5 ms	-	-	-	-	-	`xquic` does not implement HyStart [23]
			0.72	0.81	-2 Mbps	0 ms	-	-	-	-	-	Compared to TCP CUBIC w/o HyStart
✗	`xquic`	Reno	0.38	0.81	-4 Mbps	-3 ms	-	-	-	-	-	Implementations verified to be compliant with existing standards.
✗	`neqo`	CUBIC	0	0.62	-6 Mbps	-5 ms	-	-	-	-	-	

[#] Lines of code in required modification.

[+] Earlier identified and fixed [35].

[*] Implementation difference identified but not fixed. Implementation found to be conformant to TCP CUBIC with HyStart disabled.



(a) `xquic` **BBR original, Conf.**= 0.15
Conf.-T= 0.42, **Δ-tput**= +4, **Δ-delay**= 0

(b) `xquic` **BBR modified, Conf.**= 0.38
Conf.-T= 0.47, **Δ-tput**= 0, **Δ-delay**= −2

Figure 14: `xquic` **BBR's conformance before and after reducing** `cwnd` **gain from 2.5 to 2.**

to pay attention not only to the implementations of the CCA but also to the implementation of the QUIC stack in order to achieve high conformance. The investigation into the low conformance of `xquic` Reno and `neqo` CUBIC is left as future work.

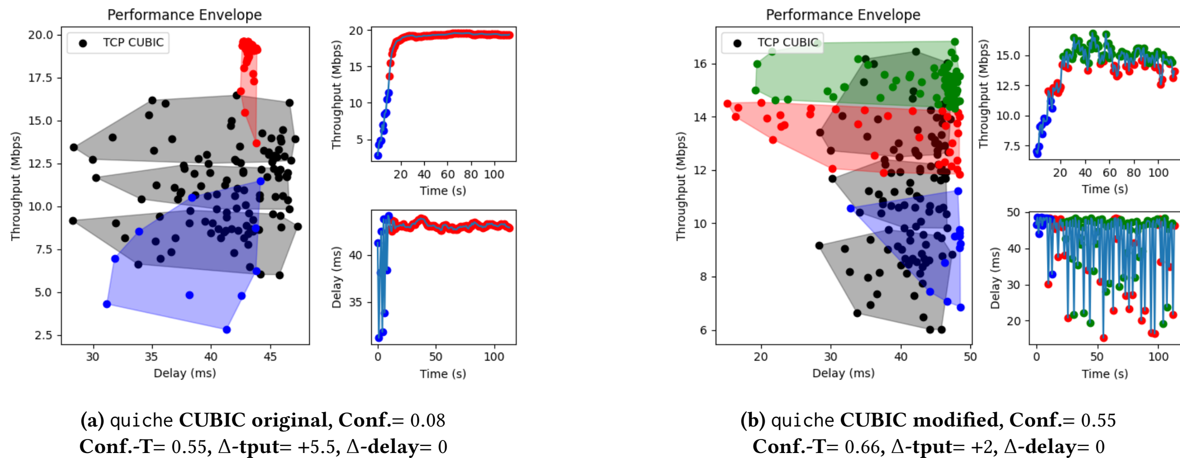
6 DISCUSSION AND FUTURE WORK

Given the low conformance that we have identified in modern QUIC stacks, we believe that QUIC congestion control research deserves more attention and further study as the results of classic CCA research may no longer apply. Fully understanding the interactions and impact of new QUIC stacks is likely to be a continuous process as these implementations morph with time. Also, we recognize that there are some limitations in our current measurement study.

Refining bandwidth-share analysis. The throughput ratios discussed in §4.3 provide an estimate of how well implementations can coexist with each other. However, in the future, we would like to refine our approach to measuring coexistence and general intra-CCA friendliness. In addition to running experiments over a larger range of network conditions, we would also like to experiment with different applications and measure their application-level metrics (such as QoE for video streaming). In our experiments, we launch both flows together. It is likely helpful to understand the impact of different start times and different flow durations on fairness.

Extending the Performance Envelope to other applications. Besides benchmarking congestion control applications, the performance envelope also has the potential to serve as a tool for helping application choose their desired congestion control algorithms. Different applications usually value different network metrics. For example, live-streaming applications will generally value low latency, in contrast to applications that perform bulk downloads and value high throughput. Such applications can possibly leverage the performance envelope to identify the trade-off space they want to operate in and then select a congestion control algorithm whose performance envelope has the maximum overlap with their desired performance envelope.

Systematic Root Cause Analysis. While the methodology applied in this paper has largely been successful in identifying low-conformance implementations of congestion control algorithms, we would like to do more to aid the debugging of these implementations. We feel that time series graphs (such as the ones in Figure 15) and Conformance-T are a good starting point in investigating which aspects an implementation may be differing in (such as `cwnd` or the sending rate). In the future, we would also like to



(a) quiche **CUBIC original, Conf.**= 0.08
Conf.-T= 0.55, Δ-**tput**= +5.5, Δ-**delay**= 0

(b) quiche **CUBIC modified, Conf.**= 0.55
Conf.-T= 0.66, Δ-**tput**= +2, Δ-**delay**= 0

Figure 15: quiche **CUBIC's conformance before and after disabling its detection of spurious packet losses (RFC8312 [46]).**

automatically extract key parameters from implementations and try to correlate them with Δ-throughput and Δ-delay values of their Performance Envelopes. There is also scope for differentiating between implementation-level and stack-level differences for these QUIC implementations. For example, if we find that the same qualitative deviation in the PE across all the evaluated CCAs for a given QUIC stack (for example, say all the CCAs in a QUIC stack achieve lower throughput than their kernel counterparts), it may suggest that the root cause of this non-conformance lies in how the underlying QUIC stack is implemented, rather than in the implementation of the individual CCAs.

Transitivity. In our earlier study of 4 QUIC stacks [35], we found that the performance of the CCA implementations was transitive, i.e. if a CCA X achieved higher throughput competing with CCA Y and CCA Y achieved higher throughput competing with CCA Z, then CCA X would achieve higher throughput when competing with CCA Z. However, from our results of the 11 QUIC stacks evaluated in this paper, we found that the relative performance of QUIC implementations is *not* transitive between different CCAs. For example, lsquic CUBIC beats msquic CUBIC and msquic CUBIC beats chromium BBR, but lsquic CUBIC does not beat chromium BBR when they compete in deep buffers. However, among the QUIC stacks that we evaluated, the intra-CCA performance seems to be transitive. That is, transitivity is likely to exist between QUIC implementations for the same congestion control algorithm. A more detailed study of transitivity between CCAs remains as future work.

Comparing Fairly Across Different CCAs. Currently, while measuring the conformance of a QUIC implementation, we run it alongside its corresponding TCP implementation (or the reference flow). This is because our idea of conformance is built around *replaceability*–that is, we want to determine how easily a QUIC implementation can mimic a standard TCP implementation in terms of performance and behavior. However, this also means that our calculated PEs are only comparable to the PEs of other implementations implementing the same congestion control algorithm. In the future, we would like to define and experiment with running all QUIC implementations alongside the same standard background flow. This would allow us to have a fair basis to compare implementations of *different* congestion control algorithms.

Keeping up with the kernel. Even though the Linux kernel's TCP stack is a relatively stable reference for measuring the conformance of QUIC implementations, it is still a moving target. The kernel will also continue to evolve as new RFCs are proposed and implemented, as we have already seen with the implementation of Hystart in §5. In fact, RFC8312 [46], whose implementation in quiche reduced the conformance of its CUBIC implementation (Figure 15), is scheduled to be deployed only in the next stable version of the Linux kernel. These developments make a case for conducting regular conformance tests for QUIC implementations every time a new *milestone* kernel version with significant changes to the TCP stack is released.

7 CONCLUSION

In this paper, we present the results of a measurement study of the congestion control algorithms in 11 popular open-source QUIC stacks. To the best of our knowledge, our measurement study is likely the most comprehensive evaluation of congestion control algorithms in modern QUIC stacks to date. We address the limitations of previous approaches that evaluate QUIC congestion control and propose a new metric *Conformance-T* for identifying implementations where there is scope for improving conformance via parameter tuning. Our measurement study significantly advances the state of the art in our understanding of speciation [32, 34] and raises new research questions on the evolution of CCAs for QUIC.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Vaibhav Bajpai for their valuable feedback and helpful comments. This work was supported by the Singapore Ministry of Education grant T1 251RES1917 and the NUS-AWS Cloud Credits for Research Grant.

ETHICS

This work does not raise any ethical issues.

REFERENCES

[1] 2021. *Active QUIC implementations.* https://github.com/quicwg/base-drafts/wiki/Implementations
[2] 2021. *iPerf, the Speed Test Tool for TCP.* https://iperf.fr/iperf-doc.php
[3] 2022. *Alibaba's QUIC implementation, xquic.* https://github.com/alibaba/xquic
[4] 2022. *Amazon Web Services's QUIC implementation, s2n-quic.* https://github.com/aws/s2n-quic
[5] 2022. *Cloudflare's QUIC implementation, quiche.* https://github.com/cloudflare/quiche
[6] 2022. *Facebook's QUIC implementation, mvfst.* https://github.com/facebookincubator/mvfst
[7] 2022. *Google Chrome, Lighthouse.* https://github.com/GoogleChrome/lighthouse
[8] 2022. *Google's QUIC implementation, chromium.* https://www.chromium.org/quic/playing-with-quic
[9] 2022. *Go's QUIC implementation, quic-go.* https://github.com/lucas-clemente/quic-go
[10] 2022. *H2O's QUIC implementation, quicly.* https://github.com/h2o/quicly
[11] 2022. *LiteSpeed's QUIC implementation, lsquic.* https://github.com/litespeedtech/lsquic
[12] 2022. *Microsoft's QUIC implementation, msquic.* https://github.com/microsoft/msquic
[13] 2022. *Mozilla's QUIC implementation, neqo.* https://github.com/mozilla/neqo
[14] 2022. *Rust's QUIC implementation, quinn.* https://github.com/quinn-rs/quinn
[15] 2023. *aioquic: QUIC network protocol in Python.* https://github.com/aiortc/aioquic
[16] 2023. *Akamai QUIC.* https://akaquic.com/
[17] 2023. *IETF QUIC implementation in Haskell.* https://github.com/kazu-yamamoto/quic
[18] 2023. *IETF QUIC implementation in Java.* https://bitbucket.org/pjtr/kwik/src/master/
[19] 2023. *nginx-quic.* https://hg.nginx.org/nginx-quic/
[20] 2023. *ngtcp2.* https://github.com/ngtcp2/ngtcp2
[21] 2023. *picoquic.* https://github.com/private-octopus/picoquic
[22] 2023. *quant.* https://github.com/NTAP/quant
[23] P. Balasubramaniam, Y. Huang, and M. Olson. 2023. *HyStart++: Modified Slow Start for TCP.* https://www.rfc-editor.org/rfc/rfc9406
[24] Prasenjeet Biswal and Omprakash Gnawali. 2016. Does QUIC Make the Web Faster?. In *Proceedings of IEEE Global Communications Conference (GLOBECOM).* IEEE, 1–6.
[25] 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.
[26] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. 2015. HTTP over UDP: an Experimental Investigation of QUIC. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC).* 609–614.
[27] F5. 2023. *Overview of the BIG-IP HTTP/3 and QUIC profiles.* https://support.f5.com/csp/article/K60235402
[28] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *Operating Systems Review* 42 (07 2008), 64–74. https://doi.org/10.1145/1400097.1400105
[29] John A Hartigan and Manchek A Wong. 1979. Algorithm AS 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28, 1 (1979).
[30] 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 Internet Measurement Conference (IMC).* 290–303.
[31] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the conference of the ACM special interest group on data communication (SIGCOMM).* 183–196.
[32] 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.* 14–20.
[33] Péter Megyesi, Zsolt Krämer, and Sándor Molnár. 2016. How quick is QUIC?. In *Proceedings of International Conference on Communications (ICC).* IEEE, 1–6.
[34] Ayush Mishra and Sherman Lim. 2022. QUICBench. https://github.com/NUS-SNL/QUICbench
[35] Ayush Mishra, Sherman Lim, and Ben Leong. 2022. Understanding speciation in QUIC congestion control. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC).* 560–566.
[36] Ayush Mishra, Wee Han Tiu, and Ben Leong. 2022. Are we heading towards a BBR-dominant Internet?. In *Proceedings of the 22nd ACM Internet Measurement Conference (IMC).* 538–550.
[37] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of USENIX Annual Technical Conference (ATC).* 417–429.
[38] Mirko Palmer, Thorben Krüger, Balakrishnan Chandrasekaran, and Anja Feldmann. 2018. The QUIC Fix for Optimal Video Streaming. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC.* 43–49.
[39] Tommy Pauly. 2021. *QUIC usage at Apple.* https://tinyurl.com/applequic
[40] James Pavur, Martin Strohmeier, Vincent Lenders, and Ivan Martinovic. 2021. QPEP: A QUIC-Based Approach to Encrypted Performance Enhancing Proxies for High-Latency Satellite Broadband. In *Proceedings of NDSS.*
[41] I Rhee, L Xu, S Ha, A Zimmermann, L Eggert, and R Scheffenegger. 2018. RFC 8312: CUBIC for Fast Long-Distance Networks. https://datatracker.ietf.org/doc/html/rfc8312
[42] Darius Saif, Chung-Horng Lung, and Ashraf Matrawy. 2021. An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse. In *IEEE International Conference on Communications (ICC).* IEEE, 1–6.
[43] Canada Sandvine Inc. Waterloo, ON. 2022. *The 2022 Global Internet Phenomena Report.* https://www.sandvine.com/phenomena
[44] Simon Scherrer, Markus Legner, Adrian Perrig, and Stefan Schmid. 2022. Model-Based Insights on the Performance, Fairness, and Stability of BBR. In *Proceedings of the 22nd ACM Internet Measurement Conference* (Nice, France) *(IMC '22).* Association for Computing Machinery, New York, NY, USA, 519–537. https://doi.org/10.1145/3517745.3561420
[45] Ranysha Ware, Matthew K Mukerjee, Srinivasan Seshan, and Justine Sherry. 2019. Modeling BBR's interactions with loss-based congestion control. In *Proceedings of the Internet Measurement Conference (IMC).* 137–143.
[46] L. Xu, s. Ha, Vidhi Goel, and Lars Eggert. 2023. *Spurious Congestion Events.* https://tools.ietf.org/id/draft-ietf-tcpm-rfc8312bis-00.html#section-4.9