QUIC(k) Enough in the Long Run? Sustained Throughput Performance of QUIC Implementations

Michael König*, Oliver P. Waldhorst‡, Martina Zitterbart*
*Institute of Telematics, Karlsruhe Institute of Technology, {m.koenig, martina.zitterbart}@kit.edu
‡Institute of Applied Research, Karlsruhe University of Applied Sciences, oliver.waldhorst@h-ka.de

***Abstract*—QUIC aims to become a general-purpose transport protocol, and numerous implementations of the QUIC protocol already exist. Earlier evaluations often examined QUIC in conjunction with HTTP/3.0 or focused on latency metrics. The measurement studies in this paper focus on actual QUIC implementations with respect to their ability to achieve high sustained throughput in network scenarios with data rates of 10 Gbit/s. We compare six popular QUIC implementations developed in different programming languages with TCP. Our findings show significant performance improvements in several QUIC implementations compared to prior evaluations. However, it is not a homogeneous picture, as current QUIC implementations often behave quite differently. We observed that in environments with low RTTs or an increased number of packet losses, most of the surveyed QUIC implementations struggle unexpectedly and cannot compete with TCP regarding sustained throughput performance.**

***Index Terms*—QUIC, Performance Analysis, Benchmark, Throughput, Transport Protocol, TCP, UDP**

I. Introduction

QUIC is a new transport protocol implemented on top of UDP. Since its standardization by the IETF in May 2021 [1], QUIC is becoming increasingly popular. According to [2][3], a significant proportion of Internet traffic is already transferred using QUIC today. Furthermore, based on the HTTParchive dataset[1], more than 18.7 % of crawled websites already support QUIC and HTTP/3.0. In 2012, Google initiated QUIC as a more effective alternative to the conventional TCP+TLS+HTTP web stacks. Nevertheless, according to its RFC [1], QUIC aims to be a general-purpose transport protocol. Previous works show that QUIC often outperforms TCP+TLS regarding latency, particularly for small transfers typical for common web traffic, profiting from its improved handshake (i.e., integrated TLS negotiation and 0-RTT handshake). However, less research exists evaluating QUIC's performance regarding sustained throughput performance within environments supporting high bandwidth links (i.e., 10 Gbit/s) and low round trip times (RTTs) that are prevalent in data center networks or prospectively facilitated by the usage of content delivery networks (CDNs). Furthermore, to our knowledge, no evaluations exist about the impact of packet loss, corruption, and reordering upon the throughput performance of current QUIC implementations.

The contribution of this work is twofold:

- First, we survey the current performances of popular QUIC implementations regarding sustained throughput in various scenarios. We show that the implementations have evolved and improved significantly, considering former evaluations. Compared to TCP, however, the performance of the tested QUIC implementations is still significantly lower.
- Second, we explore possible causes for performance bottlenecks and highlight actions where today's QUIC implementation could be improved.

The remainder of this paper is structured as follows. Section II discusses existing performance evaluations of QUIC implementations. We present our methodology and the rationale behind selecting the examined QUIC implementations in Section III. Our benchmark findings are presented and interpreted in Section IV. Furthermore, in Section V, we summarize our evaluation results and previous findings and discuss possible steps to improve current QUIC implementations. Section VI concludes the paper.

II. Related Work

Transport layer performance has experienced much progress. For example, nowadays, TCP can reliably achieve throughput rates of up to 25 Gbit/s for a single flow when slightly tuning Kernel parameters, as demonstrated by the authors of [4]. Furthermore, using jumbo frames (i.e., MTU of 9000 Byte), the same authors report throughput rates of up to 40 Gbit/s for a single connection.

QUIC should also benefit from progress in transport layer performance, as it aims to be a general-purpose transport protocol. A significant amount of QUIC performance studies focus either on response time latencies [5] [6], on page load times [7], or the performance numbers represent application protocol-specific behavior. Thus, no clear conclusion can be drawn from these works about the throughput performance of the pure QUIC protocol itself.

Furthermore, [8][9] conducted throughput measurements with small bottleneck bandwidth rates (i.e., 100 Mbit/s) or primarily shortly-lived flows, as only small amount of data (i.e., 10 MB), were transferred.

One of the few performance studies that examine the throughput achieved in high bandwidth scenarios is [10]. The authors compared four QUIC implementations within a 10 Gbit/s network environment. They present a CPU usage

This work was supported by the bwNET2020+ project which is funded by the Ministry of Science, Research and the Arts Baden-Württemberg (MWK). The authors alone are responsible for the content of this paper.

[1]https://httparchive.org/reports/state-of-the-web#h3

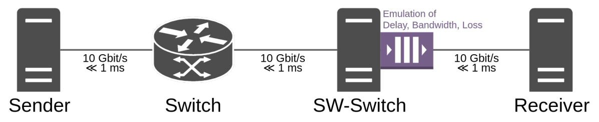


Fig. 1: Testbed

breakdown for the sender and receiver side, and found that packet I/O, crypto, ACK processing, and packet reordering represent the dominant CPU resource-consuming code units. Furthermore, the authors reported the impact of packet losses and packet reordering on throughput performance and demonstrated that most QUIC implementations react highly sensitively to packet reorder events. However, Yang et al. conducted their measurements in 2020. Thus, it is an open question if the QUIC implementations have evolved performance-wise since then. We will answer this question for *picoquic* and *mvfst*.

In [11], the authors compare the throughput performance of four QUIC implementations and TCP against their improved QUIC implementation using DPDK. We evaluated two of their four tested QUIC implementations, namely *picoquic* and *msquic*, as well.

Thus, this paper presents a comprehensive performance evaluation regarding the sustained throughput of recent QUIC implementations in high bandwidth environments.

III. Methodology

In the following, we present the methodology we used for our benchmarks, the rationale behind selecting the examined QUIC implementations, and depict our evaluation setup.

A. QUIC Implementations under Test

From the many existing QUIC implementations available[2], we focused on the performance of popular (i.e., GitHub stars and activity) QUIC implementations that are executable on Linux, and whose source code is publicly available for further investigation. In the following, we compare Microsoft's *msquic*, LiteSpeed's *lsquic*, *picoquic* (all three written in C), Facebook's *mvfst* (written in C++), *quinn*, and Amazon's *s2n-quic* (both written in Rust). As they are written in different low-level programming languages, we prioritized QUIC implementations that provide traffic generators using their QUIC library (often referred to as perf-clients by QUIC developers). This avoids the necessity to write traffic generators for each QUIC implementation in multiple different programming languages on our own. We presume that the authors of the QUIC implementation know their library best, thus utilizing their libraries within their traffic generators appropriately and, therefore, being indicative of the general performance of their QUIC library. All QUIC implementations were compiled in production mode (i.e., compiler optimizations enabled).

As a comparison baseline, we use the two traffic generators *iperf3* and *netperf* to generate TCP traffic. To avoid biases due to the usage of different congestion control algorithms (CCA),

[2]https://github.com/quicwg/base-drafts/wiki/Implementations

TABLE I: Varied Link Perturbations Across Scenarios

Scenario $\mathcal{A}$	Scenario $\mathcal{B}$	Scenario $\mathcal{C}$	Scenario $\mathcal{D}$
Unmodified	RTT	Loss	Reordering
–	0…500 ms	0…2.5 %	0…2.5 %

we opted to use Cubic [12] for all QUIC implementations as well as for both TCP traffic generators since other CCAs, such as BBR [13], are not supported in all QUIC implementations.

Since we are comparing raw TCP without TLS with QUIC, which uses encryption by default, additional computational costs would have to be imputed to the TCP throughput results. Consequently, we conducted additional QUIC experiments with encryption disabled where possible.

B. Measurements Setup

The testbed setup we used in our tests is depicted in Figure 1. It consists of four devices: A sender, a hardware switch, a software switch, and a receiver. 10 Gbit/s links interconnect all systems. The software switch uses NetEm on the onward path to the receiver to introduce artificial delays or packet loss or to limit the bottleneck bandwidth. Table I depicts the settings for the different scenarios $\mathcal{A}$–$\mathcal{D}$ we emulate with NetEm. On average, the inherent round trip time between the sender and receiver was 0.43 ms. The sender, receiver, and software switch use identically configured hardware and software (i.e., Intel Xeon W-2145 with 16 logical cores and CPU frequency scaling turned off, 128 GiB of DDR4 RAM, Intel X550-T2 network cards, running Ubuntu 22.04.1 LTS with Kernel 5.15.0-56-generic) and are configured according to recommendations in [4]. We used CPUnetLOG with a sample resolution of 100 ms to record throughput rates and CPU utilization. If not stated otherwise, hardware offloading (i.e., hand over up to 64 KByte data in super-sized packets from the application to Kernel in one go) was enabled, and a value of 1500 Byte was used for the MTU for all UDP, TCP, and QUIC experiments. To obtain average values, we performed each experiment 10 times, and error bars represent the standard deviation of these runs.

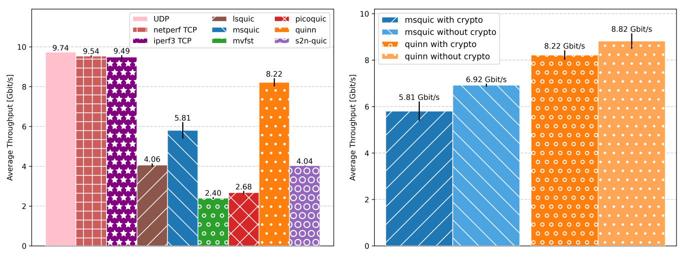
IV. Performance Results

In this section, we present the performance results of six QUIC implementations and, as a comparison, pure UDP and TCP performance results for different network scenarios.

A. Throughput and Cause Analysis

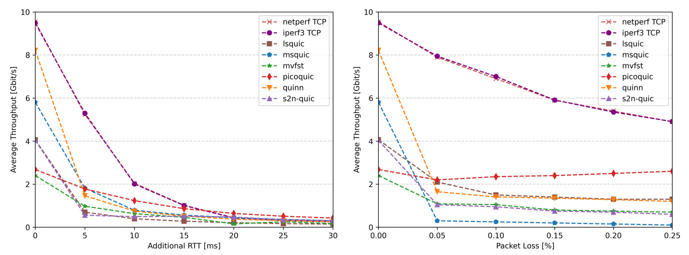
To evaluate the average throughput and possible causes for throughput limitations, a single connection sends data from the sender to the receiver in scenario $\mathcal{A}$. In this scenario no perturbations are emulated (i.e., no artificial delay, loss).

1) Maximum Average Throughput: Figure 2a illustrates the different senders' throughput. The results show that a single UDP connection driven by *netperf* can achieve a throughput of around 9.74 Gbit/s without exhibiting packet loss, thereby saturating the link bandwidth of 10 Gbit/s almost completely.



(a) Maximum Average Throughput (b) Impact of En-/Decryption

Fig. 2: Throughput Comparison (Scenario $\mathcal{A}$)



(a) RTTs (Scenario $\mathcal{B}$) (b) Random Loss (Scenario $\mathcal{C}$)

Fig. 3: Average Throughput for Different Link Characteristics

Thus, the data path for UDP through the Linux Kernel theoretically supports these high data rates and does not already limit QUIC implementations to values significantly below 10 Gbit/s of throughput.

Furthermore, as expected, TCP achieves average throughput rates of about 9.49 Gbit/s in the case of *iperf3* and 9.54 Gbit/s in the case of *netperf*. This is expected and consistent with results from previous studies with higher link bandwidths [4].

The results of all QUIC implementations contrast this. None of the tested six QUIC implementations can fully saturate the available link bandwidths. *Quinn* is the fastest of the tested evaluations and manages to achieve, on average, up to 8.22 Gbit/s of throughput. It is followed by *msquic* with an average throughput rate of up to 5.81 Gbit/s. Furthermore, the remaining implementations all achieve throughput rates of less than 4.06 Gbit/s (e.g., *msquic* only achieves 2.4 Gbit/s).

However, compared to previous results reported in 2020 by [10], *mvfst* as well as *picoquic*, significantly evolved regarding throughput performance. Back then, they attained 325 Mbit/s for *mvfst* and 489 Mbit/s for *picoquic* in throughput. Thus, an improvement for *mvfst* by factor 7.38 and for *picoquic* by 5.48x.

Despite the significant throughput improvements over time, the results show that the examined QUIC implementations struggle to date compared to TCP for scenarios with high bandwidth links and low RTTs.

2) Cryptography: To measure the overhead associated with QUIC's inherently used encryption, we compare *msquic*'s and *quinn*'s throughput performance for scenario $\mathcal{A}$ with and without encryption enabled. *msquic* and *quinn* have the only traffic generators we evaluated that support disabling encryption. Figure 2b shows significant performance gains when disabling cryptographic. The average throughput significantly improves by 22 % for *msquic* and by 7.3 % for *quinn*. However, even without encryption enabled, both QUIC implementations cannot achieve the throughput rates of TCP.

3) CPU Limitations and Scheduling: To correlate the throughput performance of the six QUIC and two TCP implementations, we recorded their per-core CPU utilization for transfers in scenario $\mathcal{A}$. While the different average CPU utilization is relatively low and similar (i.e., between 5.5 % and 9 %), the peak utilization value of the hottest core (i.e., the CPU core that is used most actively) of *lsquic*, *msquic*, *mvfst*, and *quinn* of 100 % hint at possible CPU resource contention. For example, *quinn*'s throughput performance seems limited by single-core performance, as it constantly uses primarily one core to 100 % for the whole experiment duration.

Furthermore, scheduling between the CPU cores can affect throughput performance negatively. For example, if no scheduling between the CPU cores occurs within the duration of an experiment run, *lsquic* achieves a throughput of 4.15 Gbit/s on average. However, in case the execution of the sender application gets scheduled to another core, a degradation in performance is observable (i.e., 3.7 Gbit/s).

B. Impact of Different Link Characteristics

To investigate the impact of different link characteristics on the throughput performance of the QUIC implementations, we used scenario $\mathcal{B}$–$\mathcal{D}$. We we emulate different delays and packet loss, corruption, and reordering rates by configuring the software switch with values listed in Table I.

1) Round Trip Time (RTT): Figure 3a depicts the implementations' average throughput for RTT values between 0 ms and 30 ms (Scenario $\mathcal{B}$). The results show that increasing RTT values negatively impact the throughput performance of all implementations.

TCP achieves higher throughput rates than the QUIC implementations for small RTTs (i.e., below 20 ms). However, with increasing RTT values above 20 ms, four of the QUIC implementations (i.e., *msquic*, *mvfst*, and *picoquic*) outperform TCP. Above 25 ms of RTT with *s2n-quic*, another QUIC implementation outperforms TCP. This trend generally persists for even larger RTT values.

Henceforth, for very low RTT environments such as within data center networks or prospectively facilitated by the usage of CDNs, TCP drastically outperforms current QUIC implementations regarding sustained throughput performance. In contrast, above RTTs of 20 ms multiple of the QUIC implementations do fair significantly better.

2) Packet Loss, Corruption and Reordering: In our packet loss experiments, we introduced artificial packet loss on the link between the software switch and receiver (Scenario $\mathcal{C}$). We evaluate the throughput performance when random or burst packet losses occur.

As depicted in Figure 3b, the average throughput of all implementations degrades with increased loss rates, likely caused by Cubic's inefficiencies when challenged with non-congestion-based losses as it is misinterpreting these losses falsely as an indication of congestion, thus reducing its congestion window unnecessarily. However, most QUIC implementations are more severely impacted by this than both TCP implementations. In particular, *quinn*, *s2n-quic*, and *msquic* already display a drastic decline in throughput performance when random losses occur with a low probability of 0.05 %. Moreover, *picoquic* exhibits a strange behavior as it increases its throughput rate above a random loss rate of 0.05 % and continues to do so for larger random packet loss rates.

In additional experiments with burst instead of random loss rates, and for scenarios with introduced packet corruption by applying single-bit flips, all tested implementations display similar behavior to our random loss experiment.

Moreover, the average throughput performances in the presence of out-of-order packets (Scenario $\mathcal{D}$) mirror the previously attained results when artificially introducing random packet losses on the same link. Except for *picoquic* and *msquic*, all QUIC implementations significantly degrade in average throughput performance. These results reaffirm the findings of [10] that many QUIC implementations falsely treat out-of-order packets as lost, even though they arrive before the packet loss timeout expires.

V. Summary & Discussion

The results draw an inhomogeneous picture of QUIC implementations' throughput performances. On the one hand, the performance of two QUIC implementations, namely *picoquic* and *mvfst*, has significantly improved compared to previous works. On the other hand, traffic perturbations such as random packet losses, packet corruption, or packet reordering negatively impact the performances of the QUIC implementations much more significantly than TCP. In particular, a packet loss rate of 0.05 % already severely degrades most of the evaluated QUIC implementations. Moreover, regardless of the scenario, but especially in low RTT environments, typical for data centers or when using CDNs, none of the tested QUIC implementations managed to be on par with TCP throughput-wise. The fastest QUIC implementation we tested, *quinn*, achieves an average throughput of 8.22 Gbit/s with QUIC's inherent encryption enabled and without encryption 8.82 Gbit/s. In contrast, TCP achieves an average throughput of 9.54 Gbit/s (i.e., with *netperf*). Thus, it is, in principle, possible to attain high throughput rates with QUIC. However, the other QUIC implementations have significant headroom to improve.

Possible causes for QUIC throughput limitations are manifold. First, the inherent encryption of QUIC traffic represents significant performance overhead. Disabling encryption leads to an increase in throughput of about 22 % in the case of *msquic* and 7.3 % in the case of *quinn*. Furthermore, multiple tested implementations execute predominantly in a single-threaded way, only utilizing one of the available CPU cores, thus being limited by single-core CPU performance. Moreover, scheduling of the sender application between CPU cores can adversely affect performance.

Since the performance of QUIC implementations is limited by multiple underlying factors, several optimization approaches would be feasible. End-users employing these QUIC libraries should enable CPU core pinning for their applications to avoid scheduling events between CPU cores that might lead to degraded performance results. Authors of QUIC libraries could try to overcome the single-core performance limitations by employing a multi-thread programming paradigm, thus facilitating the workload distribution across multiple CPU cores. Furthermore, kernel-bypass techniques such as *picoquic-dpdk* [11] employs seem promising to avoid overhead associated with copying data between user and kernel space. Finally, offloading QUIC's inherent crypto routines and other performance-critical tasks, such as handling packet reordering, to hardware accelerators, similar to [10], could be effective.

VI. Conclusion

In this paper, we presented a throughput performance evaluation of six QUIC implementations and UDP and TCP for different network scenarios. Although we outline that the evaluated QUIC implementations' performances have improved considerably compared to older evaluation studies, in many scenarios, particularly in scenarios with low RTT or on links that exhibit packet loss, packet corruption, or reordering, TCP outperforms current QUIC implementations regarding sustained throughput performance. The differences between QUIC and TCP are partly caused by QUIC's inherent cryptography, but primarily by how CPU resources are used. Overhead by context switches between user and kernel space and the lack of multicore support, while simultaneously being affected by scheduling between CPU cores, severely limits the tested QUIC implementations' throughput performance.

References

[1] "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000.
[2] J. Rüth *et al.*, "A first look at quic in the wild," in *PAM*, 2018.
[3] J. Zirngibl *et al.*, "It's over 9000: analyzing early QUIC deployments with the standardization on the horizon," in *ACM IMC*, 2021.
[4] M. Hock *et al.*, "TCP at 100 Gbit/s – Tuning, Limitations, Congestion Control," in *IEEE LCN*, 2019.
[5] J. Rüth and othersr, "Perceiving QUIC: Do users notice or even care?" in *CoNEXT*, 2019.
[6] D. Saif *et al.*, "An early benchmark of quality of experience between HTTP/2 and HTTP/3 using lighthouse," in *IEEE ICC*, 2021.
[7] S. Cook and all, "QUIC: Better for what and for whom?" in *IEEE ICC*, 2017.
[8] T. Shreedhar *et al.*, "Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads," *IEEE ToNaSM*, 2021.
[9] A. Yu *et al.*, "Dissecting Performance of Production QUIC," in *Web Conference 2021*, 2021.
[10] X. Yang *et al.*, "Making QUIC Quicker With NIC Offload," in *Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2020.
[11] N. Tyunyayev *et al.*, "A high-speed QUIC implementation," in *3rd Intern. CoNEXT Student Workshop*, 2022.
[12] S. Ha *et al.*, "CUBIC: a new TCP-friendly high-speed TCP variant," *ACM SIGOPS*, 2008.
[13] N. Cardwell *et al.*, "Bbr: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time," *Queue*, vol. 14, no. 5, 2016.