Examining the Heterogeneous Throughput Performance Landscape of QUIC Implementations

Michael König[§*], Sebastian Rust[‡*], Martina Zitterbart[§], Björn Scheuermann[‡]
[§]Institute of Telematics, Karlsruhe Institute of Technology, Karlsruhe, Germany, {m.koenig, martina.zitterbart}@kit.edu
[‡]Technical University of Darmstadt, Darmstadt, Germany, {sebastian.rust@, scheuermann@kom}.tu-darmstadt.de

***Abstract*—QUIC, a UDP-based transport protocol that integrates TLS for security and reduces connection latency, has gained widespread adoption and is now underpinning a substantial share of data traffic for major platforms like Cloudflare, Google, and Facebook. Given its growing deployment across major Internet platforms, there is growing attention on the performance potential of QUIC implementations. This paper provides an in-depth study of different QUIC implementations on a hardware testbed with 10 Gbit/s links. Our focus is on the achievable goodput in different scenarios and with different implementations. In contrast to other performance studies of QUIC, we investigated QUIC together with multiple versions of HTTP and used multiple streams for the data transfer. Our results show that merely choosing a different application protocol (i.e., HTTP/3 versus HTTP/0.9) can reduce goodput by as much as 27 %. Dedicated traffic generators can further significantly boost achievable goodput, in cases more than doubling the throughput obtained via HTTP. Moreover, our analysis reveals that increasing the number of QUIC streams may potentially double the throughput of multi-segment data transfers, depending on the implementation. Additionally, certain QUIC implementations can saturate a 10 Gbit/s link by increasing packet sizes, indicating that QUIC packet processing speed, rather than raw transmission capacity, is a primary bottleneck. These findings highlight QUIC's capabilities, limitations, and implementation heterogeneity. The differences between QUIC and QUIC+HTTP throughput emphasize the need for dedicated performance tests. Understanding these distinctions is crucial for analyzing, optimizing, and maximizing QUIC's performance.**

***Index Terms*—QUIC, Transport Protocols, Performance, Benchmark**

I. Introduction

The QUIC protocol, initially developed by Google and later standardized by the IETF, was explicitly designed to minimize latency, boost security, and eliminate head-of-line blocking that affects the traditional TCP+TLS stack. Now widely adopted by major players like Cloudflare, Google, and Facebook, QUIC is the foundational protocol for HTTP/3, making it a crucial component of Internet traffic.

In high-bandwidth scenarios, such as in data center networks where data is transferred at speeds of 10 Gbit/s and more, current research indicates that none of the open-source QUIC implementations match the throughput achieved by conventional TCP+TLS stacks. While these extreme environments may not be commonplace on the broader Internet, they are prevalent in specific contexts. For instance, Microsoft's specification of QUIC for the SMB file-sharing protocol [1] highlights the practical interest in using QUIC for high-bandwidth bulk data transfer outside of HTTP.

Previous studies on QUIC performance in high-bandwidth environments have highlighted the significant role of implementation heterogeneity in influencing throughput. We refine and expand upon these findings by demonstrating that performance variations are not solely driven by the QUIC implementation itself but also by how different applications leverage the implementations. Our experiments show that choosing HTTP/3 over HTTP/0.9 can lead to a performance penalty of up to 27 %, depending on the specific implementation. Conversely, utilizing multiple streams in HTTP/3 can enhance performance, underscoring the importance of thoughtful application design when selecting and structuring QUIC-based data transfers and performance measurements.

Additionally, our findings show that QUIC implementations exhibit distinct responses to fundamental network conditions, such as round-trip time (RTT) and packet loss, as well as to hardware acceleration features. For example, increasing the MTU from 1500 to 9000 Bytes allowed some implementations to thoroughly saturate a 10 Gbit/s link, while others showed only modest improvements. These findings underscore the impact of deployment choices on data transfer performance, such as selecting the application protocol and determining data chunk sizes. Additionally, the significant overhead of QUIC packet processing highlights the need for QUIC-specific offloading techniques.

Moreover, the substantial differences between QUIC-only and QUIC+HTTP throughput emphasize the necessity of dedicated performance tests for each use case. Performance results from QUIC+HTTP do not necessarily reflect the pure transport-layer capabilities of QUIC implementations, and vice versa. These factors should be incorporated into evaluation criteria when optimizing deployments for maximum throughput and assessing the performance of QUIC implementations.

The following are the main contributions of this work:

First, we analyze the sustained throughput of popular open-source QUIC implementations across different layers of the network stack. Second, we identify potential performance bot-

This work was supported by the bwNET2.0 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.

[*]Equal contribution

tlenecks and highlight opportunities for optimization across these layers.

- *Application Layer:* We find that the application layer introduces substantial overhead compared to pure QUIC traffic. Additionally, our analysis reveals significant performance asymmetries between different QUIC implementations, depending on the combination used on the sender and receiver sides. Furthermore, we observe significant differences in throughput performance between HTTP/0.9 and HTTP/3, depending on the implementation.
- *QUIC Layer:* Our findings reveal that throughput performance varies across different QUIC implementations and is influenced by the choice of traffic generators. Additionally, we assess performance under diverse network conditions, including increased delay and packet loss. Our evaluation indicates that QUIC implementations have evolved and improved considerably compared to previous studies. However, despite these advancements, their performance remains noticeably lower than TCP's while demanding significantly higher CPU resources.
- *Lower Network Layers:* We evaluate the impact of offloading and increased packet sizes, demonstrating that these optimizations can significantly enhance QUIC efficiency.

The remainder of this paper is organized as follows. In Section II, we review existing performance evaluations of QUIC implementations. Section III provides the necessary background information relevant to our study. Our methodology and the rationale behind our in-depth analysis across network layers are detailed in Section IV. We present and interpret our experimental results for QUIC+HTTP in Section V and for QUIC-only in Section VI. In Section VII, we summarize and discuss the key insights derived from our study.

II. Related Work

As QUIC was developed as the mandatory transport protocol for HTTP/3, early performance measurements focused on web-centric scenarios [2] [3] [4].

The authors of [5] modify the QUIC Interop Runner (QIR) to support QUIC performance measurements in various implementation combinations, a strategy that we also use in our work. However, their approach uses HTTP/3 as the primary application protocol, meaning that their results reflect the combined performance of both HTTP/3 and QUIC. In contrast, our work examines how different HTTP versions affect achievable throughput and, by eliminating confounding factors, provides a clearer perspective on raw QUIC performance.

In [6], the authors present detailed performance measurements by implementing the methodology from [7] specifically for the *msquic* implementation. However, it is important to note that the resulting data are exclusively for *msquic* and, therefore, do not reflect the complex and complicated interaction between different implementations.

The authors of [8] introduce a kernel bypass mechanism using the *Data Plane Development Kit (DPDK)* in *picoquic*. This modification nearly triples the achievable goodput in high-bandwidth environments, reaching speeds of up to 20 Gbit/s.

The authors of [9] evaluate four QUIC implementations in a 10 Gbit/s network environment, analyzing CPU usage on the sender and receiver sides. Their study identified packet I/O, cryptographic operations, ACK processing, and packet reordering as the primary contributors to CPU consumption. Additionally, they examined the impact of packet loss and reordering on throughput, revealing that most QUIC implementations are susceptible to packet reordering events. However, these measurements were conducted in 2020, and the landscape of QUIC implementations has since evolved.

In [10], we evaluate the sustained throughput performance of six QUIC implementations in a high-bandwidth environment with 10 Gbit/s links. Our findings reveal that while both TCP and pure UDP can fully utilize the available bandwidth, none of the tested QUIC implementations achieves the same level of performance, with throughput rates ranging from 2.4 Gbit/s to 8.22 Gbit/s. To further analyze performance bottlenecks, we disable QUIC's cryptographic routines in two implementations. While this improves the performance, our results suggest that cryptographic overhead is not the sole factor driving performance differences. Additionally, when introducing link perturbations such as packet loss and packet reordering, QUIC implementations exhibit significant performance degradation compared to TCP. Finally, our CPU utilization analysis indicates that throughput limitations stem primarily from single-core performance constraints. Moreover, process scheduling across CPU cores was found to negatively impact transfer rates, further highlighting the challenges in optimizing QUIC implementations for high-speed networks.

III. Background

This section provides the necessary background information relevant to our study.

A. QUIC Streams and Frames

One of the design goals for QUIC is to mitigate head-of-line blocking, a problem often seen with TCP in conjunction with HTTP/2 multiplexing. To this end, QUIC establishes *streams* to provide applications with an ordered and reliable byte-stream interface. Each stream functions independently, so delays in one stream do not block the progress of others, effectively reducing head-of-line blocking. Beneath these streams, QUIC encapsulates data into packets composed of discrete *frames*, which act as lightweight transport abstractions carrying diverse payloads such as user data, acknowledgments, and connection management information.

B. Network Stack

Figure 1 illustrates the distinct paths a TCP segment and a QUIC packet take from the receiving network interface card (NIC) to the application, along with the corresponding ACK segment or frame. When a TCP segment arrives at the NIC, the packet is processed by the network driver, and the actual payload is transferred to the TCP socket for the application. In

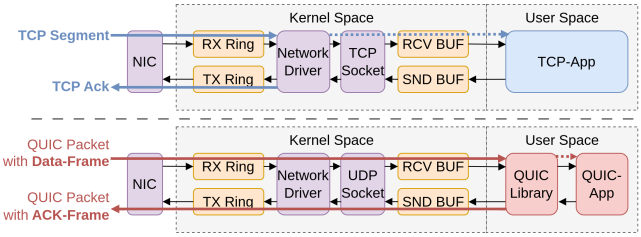


Fig. 1: Simplified Comparison of QUIC and TCP Network Stack Traversal. Solid colored lines trace the path of network packets through the stack, while dotted lines depict the flow of application data.

response, the acknowledgment (ACK) for the received segment is triggered within the kernel by the network driver, without the need for context switches. In contrast, a QUIC packet must traverse the entire network stack before reaching the QUIC library via its UDP socket. The QUIC library then generates an acknowledgment as an ACK frame, encapsulates it within a new QUIC packet, and sends it within a QUIC packet to the remote endpoint. Unlike TCP, which can process acknowledgment more directly, this procedure requires two context switches on the sender and one on the receiver—three context switches in total—for each generated acknowledgment. Unlike TCP, which benefits from kernel-level implementation that allows direct, in-kernel acknowledgment processing, QUIC implementations in user space cannot easily adopt a similar strategy. This limitation stems from the fact that ACK frames are sent as part of the payload of QUIC packets and may be multiplexed with other frames. As a result, the entire packet must be processed in user space, introducing additional overhead.

C. Offloading

Offloading techniques can minimize the overhead of context switches during QUIC packet transmission and reception. These techniques primarily operate at the UDP layer and are not specific to QUIC. For example, *Generic Segmentation Offload (GSO)* [11] allows the sender to batch multiple UDP datagrams and write them to the UDP socket with a single context switch. The NIC subsequently segments this batch into individual IP packets, although the QUIC implementation must still generate a proper QUIC header for each packet. Similarly, *Generic Receive Offload (GRO)* reduces context switches on the receiving side by batching multiple UDP datagrams. As with GSO, GRO works at the UDP level, so the QUIC implementation is responsible for processing each individual QUIC packet.

As discussed in Section VI-C, support for these offloading techniques generally improves throughput in QUIC implementations, though the magnitude of the benefit can vary significantly among different implementations.

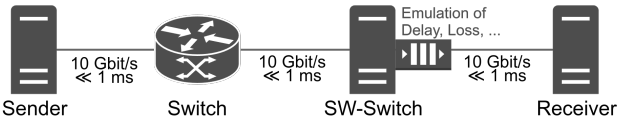


Fig. 2: Testbed

TABLE I: Testbed Hard- and Software

CPU	Intel Xeon W-2145, 3.7–4.5 GHz, 8 Cores / 16 Threads
RAM	128 GB (4x 32 GB DDR4 with 2666 MT/s)
NIC	Intel X550-T2
OS	Linux Ubuntu 22.04.1 LTS
Kernel	5.15.0-56-generic

IV. Evaluation Methodology

To investigate the performance of the QUIC implementations, we set up a 10 Gbit/s testbed as depicted in Figure 2. The testbed consists of two endpoints, one software switch and one hardware switch. The sender endpoint is connected via the hardware switch to the software switch, and both forward packets between sender and receiver endpoint. 10 Gbit/s links interconnect all systems. The sender, receiver, and software switch use identically configured hardware and software, shown in Table I, and are configured according to recommendations in [12] [13] [14]. The software switch uses NetEm [15] on the path from the sender to the receiver to introduce artificial delays, packet loss, or to limit the bottleneck bandwidth. We cross-validated the results attained by using NetEm with TLEM [16] and achieved the same qualitative results. Therefore, in the following, we only report the results of NetEm used on the software switch. Using the software switch, we configure distinct characteristics for the network link to the receiver. These network characteristics, denoted as scenario $\mathcal{A}$–$\mathcal{E}$, correspond to the values listed in Table II.

For all QUIC implementations, we used Cubic [17] as the congestion control algorithm (CCA) since other CCAs, such as BBR [18], are not supported in all QUIC implementations. Furthermore, we increased the send and receive buffer sizes for UDP, as the default values are known to be insufficient for optimal QUIC performance [19]. Unless otherwise specified, we used an MTU of 1500 bytes. To log throughput and CPU utilization, we use *pidstats* [20] and *CPUnetLOG* [21], and we use *TCPlog* [22] to record TCP internal metrics.

To evaluate the performance of the integrated QUIC+HTTP stack, we use the Quic Interop Runner (QIR) introduced by Seemann and Iyengar [23], a framework designed for testing the interoperability of QUIC implementations. Jaeger et al. [5] subsequently extended the QIR to support bulk data transfer on bare-metal systems instead of simulated networks, thereby highlighting the impact of various QUIC implementation pairs on overall transmission performance.

Building on Jaeger et al.'s modifications, we enhance our test framework with a toggle for selecting HTTP/3 or HTTP/0.9, an option to transfer multiple files over an established QUIC

connection for both protocols, and a configurable parameter for concurrent stream control. This enables us to cross-test different QUIC implementations in respect to their performance in combination with HTTP, and the impact that segmenting files into multiple chunks and controlling for the allowed number of concurrent streams has. Note that the implementation of HTTP/0.9 in the tested QUIC implementations is solely intended for compatibility testing with QIR and is not designed for production use. The transfer test used in QIR requires implementing an HTTP-GET request using HTTP/0.9 syntax, along with the appropriate ALPN to establish the connection. All transmissions occur over a single QUIC connection, with each HTTP/0.9 request using one QUIC stream. QIR allows multiplexing of different requests, but does not require it. The results of our experiments, which compare different HTTP versions, segmented file transfers, and concurrent stream limits, are presented in Section V-A and Section V-B.

To isolate QUIC performance from the influence of HTTP, we employ dedicated QUIC traffic generators to assess the QUIC implementations independently. These generators are available in two variants: integrated and generic. Some implementations include an integrated traffic generator developed alongside the QUIC implementation, meaning it works exclusively with that particular implementation. In contrast, generic tools, such as quicperf [24], support multiple QUIC backends. Both types of generator allow for the measurement of sustained goodput for supported QUIC implementations. Not all QUIC implementations include an integrated traffic generator or are supported by quicperf; support is denoted in Table III. A comparison between the goodput achieved with the QUIC+HTTP stack and that measured using traffic generators is presented in Section VI-A, and Section VI-B discusses the impact of different traffic generators on the performance of the same implementation.

Building on our initial QUIC traffic generator experiments, we further investigate how hardware offloading and larger packet sizes affect the throughput performance of various QUIC implementations (see Section VI-C and Section VI-D) For this purpose, we repeat the traffic generator experiments with GSO and GRO enabled and increase the network MTU to 9000 bytes. Furthermore, we study the impact of different link characteristics on the throughput performance of the QUIC implementations, we varied the RTT, probability of packet losses, packet reordering, and packet errors (i.e., bit flips) across the various scenarios by configuring the software switch with values listed in Table II.

Lastly, we compare the CPU usage patterns of the integrated traffic generator and quicperf to identify potential bottlenecks in Section VI-F and examine throughput improvements over time for various QUIC implementations in Section VI-G.

V. Combined QUIC & HTTP Performance

To evaluate QUIC's performance across HTTP versions, we employ QIR to cross-test various QUIC implementations by retrieving a single file using HTTP/3 or HTTP/0.9, respectively.

TABLE II: Varied Link Characteristics Across Scenarios

Scenario $\mathcal{A}$	Scenario $\mathcal{B}$	Scenario $\mathcal{C}$	Scenario $\mathcal{D}$	Scenario $\mathcal{E}$
Unmodified	RTT	Loss	Reordering	Errors
–	0…300 ms	0…2.5 %	0…2.5 %	0…2.5 %

TABLE III: Evaluated QUIC Implementations

Implementation	Language	Integrated Traffic Generator	Quicperf Generator
lsquic [25]	C	✓	✓
picoquic [26]	C	✓	✓
ngtcp2 [27]	C	–	✓
quiche (Cloudfare) [28]	Rust	–	✓
quic-go [29]	Go	–	–

In addition, we examine how partitioning the data into multiple chunks and increasing the number of concurrent streams influence overall throughput.

A. Single File

To establish a performance baseline for further evaluation, we conducted throughput tests on single-file transfers over HTTP/3 by replicating the methodology described by Jaeger et al. The transmitted file is 8 GB large. Our network configuration was implemented using scenario $\mathcal{A}$. Figure 3 shows the achieved goodput between various QUIC implementations. The server implementation is on the x-axis and the client on the y-axis. The achieved goodput varies significantly depending on the client–server pairing, with *ngtcp2–ngtcp2* delivering the highest throughput at 4172 Mbit/s and the slowest pairing measured is *quic-go–quiche* with 1220 Mbit/s.

Figure 4 shows the results of our experiments comparing the different HTTP versions. The heatmap shows the average change in throughput observed when switching from HTTP/3 to HTTP/0.9 in the QIR experiment runs. Positive numbers denote an increase and negative numbers a reduction in throughput. To evaluate the significance of these improvements, we performed a t-test and rejected the null hypothesis when $p > 0.05$. Non-significant differences are indicated by patterned fields in the

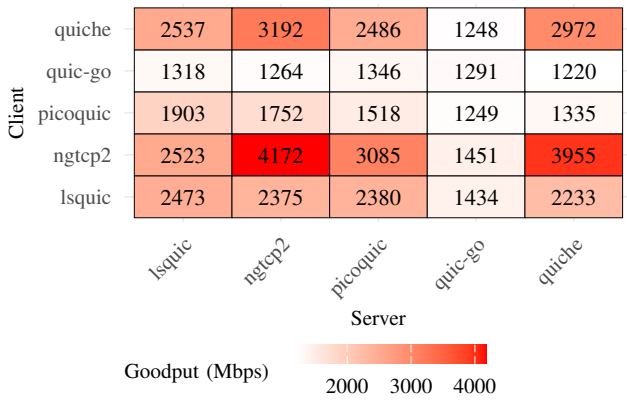


Fig. 3: HTTP/3 Single File experiment results

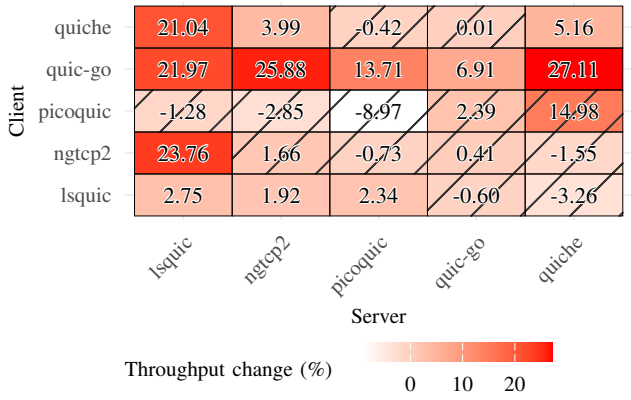


Fig. 4: Improvement in throughput for using HTTP/0.9 over HTTP/3. Patterned fields indicate that the measured difference is not statistically significant.

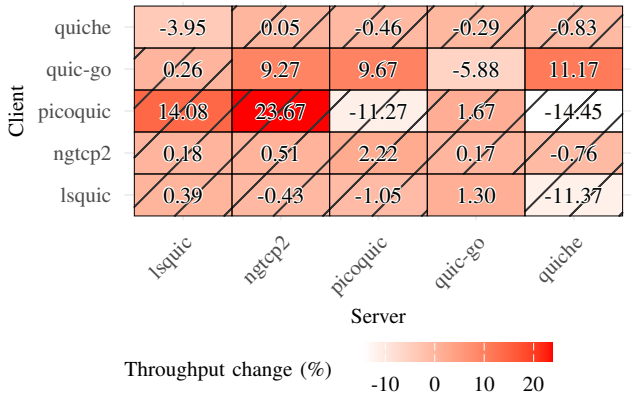


Fig. 5: Comparison of Throughput Performance: Requesting 8 GB of data distributed across 100 files versus a single file. Patterned fields indicate that the measured difference is not statistically significant.

heatmap. In general, switching to HTTP/0.9 results in performance gains. All pairs with statistically significant differences showed improvements ranging from 1.92 % for the *lsquic-ngtcp2* pair to 27.11 % for the *quic-go-quiche* pair. During our experiments, we observed that none of the implementations tested complied with HTTP/0.9 behavior but only with protocol messages. Instead of establishing a new connection for every HTTP/0.9 request, implementations mimick their HTTP/3 behavior, ranging from sequentially requesting files sequentially to fetching all files concurrently. Consequently, any performance loss is attributable to the specific implementation of HTTP/3, given that the underlying transport-level behavior remains unchanged.

Take away: The application protocol and its implementation can have a profound impact on performance measurements. It is essential to control both the application protocol and its specific implementation to obtain accurate results for the performance of a QUIC implementation.

B. Multiple Files and Streams

Figure 5 shows the change in throughput when requesting 8 GB of data, split into 100 chunks of size 80 MB each, over HTTP/3. Positive numbers indicate an increase, while negative numbers denote a decrease in throughput. Although most of the measured differences are not statistically significant, we observe that especially *quic-go* as a client achieves up to 11.17 % more throughput with chunks. This observation is not consistent with all server pairs. With *lsquic* as the server, we do not measure significant differences, and with *quic-go* as the server, we even measure a decrease of $-5.88\,\%$.

To assess the impact of stream count on measurements, we controlled the maximum concurrent streams during transmission. We used *lsquic*, *quiche*, and *quic-go*, as they enforced the set stream limit. Due to time constraints, testing was limited to identical QUIC implementations for both client and server. Figure 6 presents the results of downloading segmented chunks using multiple streams. The x-axis represents the maximum

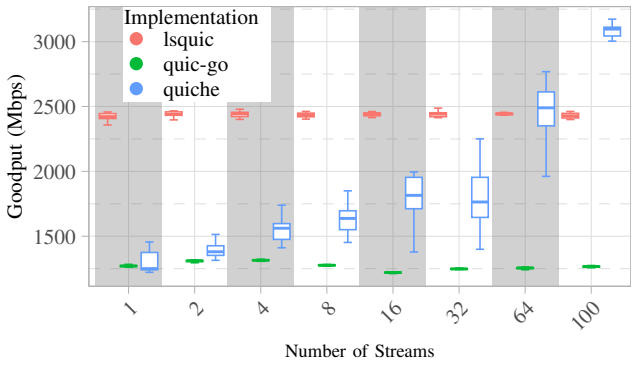


Fig. 6: Average throughput for downloading 100 chunks à 80 MB with a varying number of streams

streams used, while the y-axis shows goodput. Box plots illustrate data distribution: boxes indicate the 25th–75th percentiles, the line marks the median, and whiskers extend to the 5th and 95th percentiles, covering 90% of the data.

Utilizing more than one stream yields better performance for all tested implementations. Notably, downloading 100 chunks over a single concurrent stream reduces *quiche*'s throughput by over 50% compared to transferring a single large file on one stream in the HTTP/3 experiment. With single-stream constraint, throughput falls from 3.097 Gbit/s to below 1.5 Gbit/s. However, when using concurrent 100 streams, the throughput is comparable to the HTTP/3 single-file transmission. *Quic-go* showed slight gains with 2–8 streams, which vanished with higher stream counts. *Lsquic* numbers remained unchanged.

Figure 7 shows the CPU usage during the client-server tests. The measurements were taken using *pidstat* [20], which aggregates CPU usage for processes and their threads but does not indicate core allocation; this detail is important since core pinning can boost performance [19]. *Quic-go* leverages multiple cores, spiking CPU usage to about 400 % with 32 or more

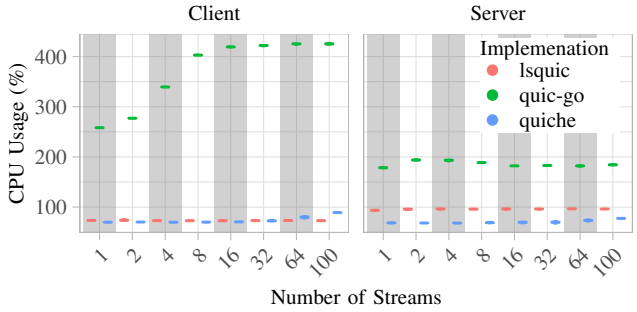


Fig. 7: Average CPU utilization across varying numbers of concurrent streams.

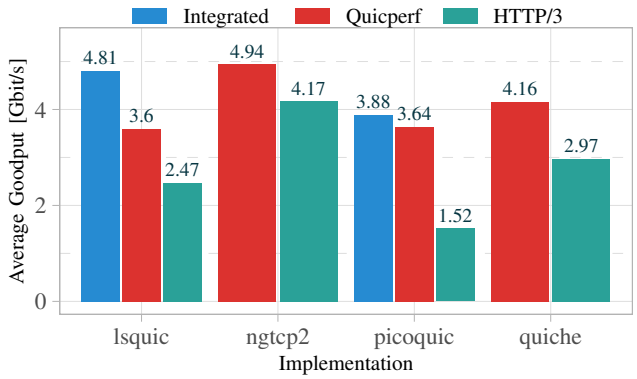


Fig. 8: Impact of HTTP and Traffic Generator on QUIC Throughput (Scen. $\mathcal{A}$). Lack of support causes missing bars.

streams, while the server remains steady between 160 % and 190 %. Nevertheless, as shown in Figure 6, the increased CPU usage does not translate to higher throughput. In contrast, both *lsquic* and *quiche* do not exceed 100% CPU, indicating a lack of multithreading, yet they achieve higher goodput than *quic-go*. The *quiche* client and server average between 50 % and 90 % CPU usage with increasing streams, with the server slightly lower than the client, while *lsquic* reaches 100 % on the server when using more than one stream and uses slightly less on the client.

Take away: Splitting a fixed data volume into multiple segments can yield an increase in overall throughput of up to 11 % during transmission. Furthermore, increasing the number of concurrent streams has been observed to nearly double the achieved throughput, as demonstrated in the *quiche* implementation. These minor modifications significantly affect throughput, making it essential to control for them in any QUIC performance assessment.

VI. QUIC Performance

To evaluate QUIC performance without the potential overhead introduced by HTTP-specific code, we tested four QUIC implementations using QUIC traffic generators.

A. Performance of QUIC vs. QUIC+HTTP/3

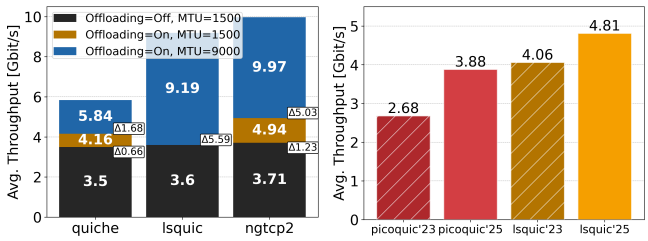
Figure 8 shows the throughput of *lsquic*, *ngtcp2*, and *picoquic* when using traffic generators versus QUIC+HTTP/3 using the QIR from Section V-A. For *lsquic* and *picoquic*, integrated traffic generators are used; quicperf is used for all. In every case, the traffic generators achieve higher throughput than QUIC+HTTP/3, though the extent of improvement varies by implementation. *picoquic*'s throughput more than doubles, from 1.52 Gbit/s to 3.88 Gbit/s, whereas *ngtcp2* shows a moderate increase from 4.17 Gbit/s to 4.94 Gbit/s. This suggests that using the QUIC traffic generators circumvents some of the processing overhead and inefficiencies inherent in the corresponding HTTP/3 implementation, enabling the QUIC transport layer to achieve higher throughput.

Take away: We conclude that throughput comparisons of QUIC+HTTP traffic within the QUIC Interop Runner (QIR) may not accurately reflect the achievable transport-layer performance of QUIC implementations, and vice versa. Throughput values can vary significantly even for the same QUIC implementation, highlighting the need for distinct evaluation methods for each use case.

B. Role of Traffic Generators

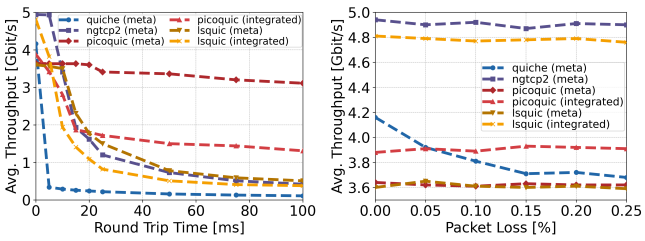
To evaluate the impact of different traffic generating applications, we compared the throughput of *lsquic* and *picoquic* by generating traffic with both the respective integrated traffic generators and the generic traffic generator *quicperf*. We patched *quicperf* to support more recent versions of the QUIC libraries, as well as to support GSO/GRO offloading. For both traffic generation methods, we used identical versions of each QUIC implementation and architecture-specific build optimizations. The results for both *lsquic* and *picoquic* indicate better throughput performance when using the traffic generators provided by the respective QUIC library authors compared to the generic *quicperf* generator (shown in Figure 8). *Lsquic* performs slightly better with its integrated generator (i.e., from 3.64 Gbit/s to 3.88 Gbit/s), while *picoquic*'s throughput significantly improves by over 1.2 Gbit/s (i.e., from 3.6 Gbit/s to 4.81 Gbit/s) in comparison. Nevertheless, the maximum throughput of 4.94 Gbit/s is achieved by combining *ngtcp2* with the generic *quickperf* generator. Profiling results using *perf* indicate that *quicperf* experiences a higher number of soft interrupts, performing up to 25 % more syscalls. This may be attributed to suboptimal buffer allocations, leading to more frequent `malloc` calls for QUIC packet handling. Thus, similar to the variations caused by different HTTP implementations and their overheads, differences in traffic generator quality and their interaction with QUIC libraries impact throughput results.

Take Away: To better compare the transport-layer performance across different QUIC implementations, a standardized traffic generator—similar to *iperf3* for TCP—is essential, as variations within specific generators can introduce additional variability. Furthermore, application developers must be mindful of potential throughput losses resulting from inefficient integration of QUIC implementations.



(a) Offloading & Packet Sizes (b) Different Versions

Fig. 9: Impact of Offloading, Packet Sizes, and Versions



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

Fig. 10: Avg. Throughput for Different Link Characteristics

C. Impact of GSO/GRO Offloading

To evaluate the impact of offloading, we compared the average throughput for three QUIC implementations *lsquic*, *ngtcp2*, and *quiche* that support enabling and disabling offloading. Implementations with offloading support can transfer up to 64 KiB in a single operation. By reducing the time spent transferring numerous small packets, enabling offloading improves the throughput of two of the three tested implementations.

Figure 9a shows the average throughput with GSO/GRO offloading enabled and disabled. In this setup, *quiche* achieves an improvement of 0.66 Gbit/s, from 3.5 Gbit/s to 4.16 Gbit/s and *ngtcp2* gains 1.2 Gbit/s, from 3.7 Gbit/s to 4.94 Gbit/s. *lsquic* shows no differences as it does not support GSO/GRO offloading. Offloading yields smaller gains than larger packet sizes, likely due to the lack of QUIC-specific support. While GRO and GSO help aggregate and segment UDP packets, QUIC libraries still create and process headers for each 1500-byte packet, matching the standard MTU.

Take away: In QUIC implementations that support them, generic offloading techniques such as GSO and GRO reduce processing overhead and improve throughput. However, QUIC-specific offloading techniques could most certainly yield even more significant throughput gains.

D. Impact of Packet Size

For QUIC implementations that support jumbo frames by using 9000-byte packets instead of the standard 1500-byte packets, we compared throughput performance for both sizes. Figure 9a shows the average throughput of the three QUIC implementations *lsquic*, *ngtcp2*, and *quiche* for both packet sizes. All implementations show higher throughput rates when using jumbo frames, as less time is spent processing smaller packets (i.e., generating QUIC headers). However, the results reveal significant differences in performance. For instance, while *quiche* shows only a modest improvement (i.e., from 4.16 Gbit/s to 5.84 Gbit/s), *lsquic* demonstrates a much more substantial increase, more than doubling its throughput from 3.6 Gbit/s to 9.19 Gbit/s. *Ngtcp2* also profits from larger packets and doubles the achieved throughput from 4.94 Gbit/s with only offloading enabled to 9.97 Gbit/s with offloading and an MTU of 9000 bytes, thus finally saturating the 10 Gbit/s link.

Take away: Compared to previous studies, our packet size evaluations quantitatively demonstrate the significant overhead of QUIC packet processing and underscore the performance gains of using jumbo frames, as well as the potential benefits of future QUIC-specific offloading techniques.

E. Throughput with Different Link Characteristics

Previous studies have reported significant throughput degradations in the presence of increased round trip times (RTTs), packet loss, reordering, or errors [9][10]. To examine the impact of different link characteristics regarding throughput, we emulate different delays, packet losses, and reordering rates by configuring the software switch with values listed in Table II (Scenarios $\mathcal{B}$–$\mathcal{E}$).

1) Round Trip Times (RTTs): Figure 10a illustrates the average throughput of the tested QUIC implementations as RTTs increase. All implementations experience a decline in throughput as the artificially introduced RTT grows. However, the impact varies significantly across the implementations. While the throughput of *ngtcp2* and *lsquic* decreases gradually and consistently as RTT increases, *quiche* sees a sharp decline as soon as 5 ms of additional delay are introduced, dropping from 4.16 Gbit/s to just 0.15 Gbit/s.

2) Packet Losses, Reordering, and Errors: The average throughput remains relatively stable or decreases only slightly for all but one implementation (*quiche*) when packet loss becomes more frequent, as illustrated in Figure 10b. Similar qualitative trends were observed with increasing packet reordering and packet errors (i.e., bit flips), but are not shown here. Compared to the severe throughput degradations reported in previous studies [9][10], our results highlight significant performance improvements in these scenarios just by using more recent versions of the QUIC implementations.

Take away: Our results indicate that most tested QUIC implementations handle increased RTTs, packet loss, packet reordering, and packet errors significantly better than in previous evaluations [9][10], even when tested under the same hardware and software setup with only more recent implementation versions.

F. Role of CPU Resources

We used Linux's profiler *perf* and *pidstat* to gather statistics about CPU usage, cache efficiency, and various hardware and software events to help diagnose performance bottlenecks.

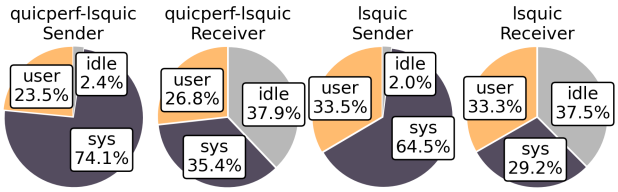
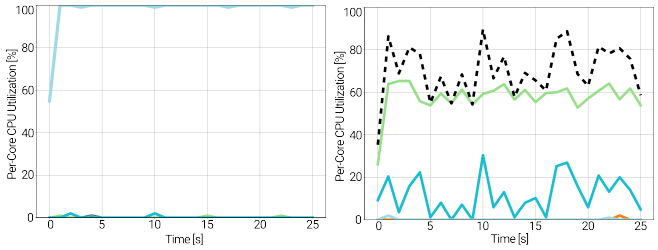


Fig. 11: CPU Time Distribution: *quicperf-lsquic* vs. *lsquic* (Each with Sender & Receiver)



(a) Sender (b) Receiver

Fig. 12: CPU Utilization of *lsquic* (One Run in Scenario $\mathcal{A}$)

For all our experiments, we recorded the CPU utilization per (logical) CPU core on the sender and receiver sides.

Figure 11 illustrates the CPU time distribution across idle, user, and kernel (sys) time for *lsquic*. From left to right, it compares time distributions for traffic generated by *quicperf* (quicperf-lsquic) and *lsquic*'s integrated traffic generator (lsquic). For both traffic generators, the figure presents CPU time allocation on both the sender and receiver sides.

The first observation is that receivers spend a significant portion of CPU time idling (37.5–37.0 %), whereas both sender systems remain nearly fully utilized, with only 2–2.4 % idle time. This suggests that throughput limitations stem from the sender rather than the receiver. Furthermore, CPU utilization, as shown in Figure 12a, reinforces this finding, with one CPU core on the sender side reaching 100 % utilization throughout the transfer. Nevertheless, only core is actively used. In contrast, the receiver's CPU utilization, shown in Figure 12b, exhibits two active logical cores, with overall utilization fluctuating (depicted by the dashed black line) between 60 % and 85 %.

Moreover, on the sender side, *quicperf-lsquic* shows significantly higher kernel (sys) time (74.1 %) compared to *lsquic* (64.5 %). This is likely due to suboptimal buffer allocations, leading to a higher frequency of system calls, such as `malloc()`. For transfers with larger packet sizes (e.g., 9000 bytes instead of 1500 bytes) as discussed in Section VI-D, we observed a significant reduction in user-space processing time, while CPU time spent in the kernel—primarily for sending—increased. In contrast, our offloading experiments in Section VI-D showed only a modest increase in kernel time relative to the user time when GSO/GRO offloading was enabled. Our findings confirm that generic offloading improves performance, but much of QUIC processing still focuses on packet handling, so tailored offloading could significantly boost throughput.

Take away: The throughput is primarily limited by the sender side rather than the receiver. Specifically, the sender side is constrained by single-core performance, as the implementations do not leverage multi-core capabilities. Furthermore, while a substantial portion of CPU time is spent on UDP packet processing, an even larger share of the QUIC implementation's CPU cycles is consumed by QUIC packet processing.

G. Performance Improvements Over Time

In addition to the improved throughput observed when handling loss and reordering events (as described above), we compared the maximum throughput performance of *picoquic* and *lsquic* in scenario $\mathcal{A}$, using evaluation results from previous works [10]. Figure 9b illustrates the average throughput of *picoquic* and *lsquic* in 2023 and 2025. Over this period, both QUIC implementations have shown substantial improvements regarding throughput. Specifically, *picoquic*'s throughput increased from 2.68 Gbit/s to 3.88 Gbit/s (a 44.8 % improvement), while *lsquic*'s throughput slightly rose from 4.06 Gbit/s to 4.81 Gbit/s (a 18.5 % increase).

Take away: While still not matching TCP in terms of throughput, the evaluated QUIC implementations continue to improve. Average throughput has increased significantly in some cases, and the handling of packet loss, reordering, and errors has also improved considerably compared to previous studies [9] [10].

VII. Summary & Discussion

Our QUIC throughput evaluations reveal several opportunities for enhancing performance across protocol layers interacting with its implementations. Notably, we observed substantial throughput variability between the tested implementations, depending on the use case and scenario. Cross-performance tests also showed striking asymmetries, with outcomes significantly influenced by the sender–receiver implementation pairings. In addition, the overhead introduced by HTTP/3 can lead to considerable throughput degradation compared to HTTP/0.9 transfers, but again with asymmetries between the different implementation. These findings underscore the importance of controlling the application layer when measuring performance.

Furthermore, the tested QUIC implementations demonstrate significantly different throughput results between QUIC-only and QUIC+HTTP scenarios. Therefore, we emphasize the need for a clearer distinction between QUIC-only and QUIC+HTTP performance benchmarks, depending on the intended use case: evaluating pure transport-layer performance (QUIC-only) or assessing real-world web performance (QUIC+HTTP). In addition, these differences caused by the application layer must be taken into consideration in future tests regarding the performance of QUIC implementations. Integrating QUIC-only performance tests alongside existing QUIC+HTTP benchmarks within the widely used QIR would establish a more comprehensive evaluation framework, ensuring that both use cases are adequately covered.

Moreover, comparing the throughput performance of different traffic generators for the same QUIC implementations reveals significant variations. These differences stem not only from the implementations themselves but also from how effectively each traffic generator interacts with the respective QUIC library. Thus, adopting a standardized traffic generator—akin to iperf3 for TCP—would enhance the consistency and comparability of QUIC throughput performance benchmarks.

Enabling GSO/GRO offloading and larger packets significantly reduces CPU overhead, boosting throughput. Notably, our results across different packet sizes underscore the dominant overhead introduced by QUIC packet processing—surpassing that of UDP packet handling, which can already be partially mitigated through GSO/GRO offloading. Implementing QUIC-specific offloading could further reduce overhead and significantly enhance performance.

QUIC implementations have notably advanced, showing improved throughput compared to earlier studies, reduced degradation at higher RTTs, and better handling of packet loss and reordering, bringing performance closer to TCP levels. However, when compared to typical TCP performance results, the evaluated QUIC implementations still fall short in terms of sustained throughput rates. To bridge this gap, support for QUIC-specific offloading, Jumbo Frames, and proposed kernel-bypass techniques, such as XDP [30] or DPDK [31], could significantly improve performance. This remains relevant even where 10 Gbit/s single-flow connections are rare. As QUIC adoption grows, even minor efficiency gains can notably reduce CPU usage, power consumption, and operational costs, while improving user throughput.

References

[1] Microsoft. *SMB over QUIC*. [Online; Accessed: March 14, 2025]. URL: https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-over-quic.

[2] Konrad Wolsing et al. "A performance perspective on web optimized protocol stacks: TCP+TLS+HTTP/2 vs. QUIC". In: ANRW'19. 2019.

[3] Jan Rüth and othersr. "Perceiving QUIC: Do users notice or even care?" In: *CoNEXT*. 2019.

[4] Darius Saif et al. "An early benchmark of quality of experience between HTTP/2 and HTTP/3 using lighthouse". In: *IEEE ICC*. 2021.

[5] Benedikt Jaeger et al. "QUIC on the highway: evaluating performance on high-rate links". In: IFIP Networking '23. 2023.

[6] *msquic*. [Online; Accessed: February 21, 2025]. URL: https://microsoft.github.io/msquic/.

[7] Nick Banks. *QUIC Performance*. Internet-Draft. 2020.

[8] Nikita Tyunyayev et al. "A high-speed QUIC implementation". In: CoNEXT-SW'22. 2022.

[9] Xiangrui Yang et al. "Making QUIC Quicker With NIC Offload". In: EPIQ'20. 2020.

[10] Michael König et al. "QUIC(k) Enough in the Long Run? Sustained Throughput Performance of QUIC Implementations". In: LCN'23. 2023.

[11] The Linux Kernel Documentation. *Segmentation Offloads*. https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html. [Online; Accessed: March 13, 2025].

[12] *Framework for TCP Throughput Testing*. RFC 6349. Aug. 2011. DOI: 10.17487/RFC6349.

[13] Kevin Corre. *Framework for QUIC Throughput Testing*. Internet-Draft. 2021.

[14] Mario Hock et al. "TCP at 100 Gbit/s – Tuning, Limitations, Congestion Control". In: *IEEE LCN*. 2019.

[15] Stephen Hemminger. "Network Emulation with NetEm". In: URL: https://api.semanticscholar.org/CorpusID:17786091.

[16] Luigi Rizzo et al. "Very high speed link emulation with TLEM". In: *IEEE LANMAN*. 2016.

[17] Sangtae Ha et al. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS operating systems review* (2008).

[18] Neal Cardwell et al. "BBR: Congestion-based congestion control: Measuring bottleneck bandwidth and round-trip propagation time". In: *Queue* 14.5 (2016).

[19] Marcel Kempf et al. "QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links". In: *Computer Communications* 223 (2024).

[20] *pidstat*. URL: https://sysstat.github.io/.

[21] *CPUnetLOG*. [Online; accessed 18. Feb. 2025]. URL: https://gitlab.kit.edu/kit/tm/telematics/congestion-control/logging/cpunetlog.

[22] *TCPlog*. [Online; accessed 18. Feb. 2025]. URL: https://gitlab.kit.edu/kit/tm/telematics/congestion-control/logging/tcplog.

[23] Marten Seemann and Jana Iyengar. "Automating QUIC Interoperability Testing". In: EPIQ '20. 2020.

[24] quicperf. *quicperf*. [Online; accessed 18. Feb. 2025]. URL: https://github.com/victorstewart/quicperf.

[25] *lsquic*. [Online; accessed 18. Feb. 2025]. URL: https://github.com/litespeedtech/lsquic.

[26] *picoquic*. [Online; accessed 18. Feb. 2025]. URL: https://github.com/private-octopus/picoquic.

[27] *ngtcp2*. [Online; accessed 18. Feb. 2025]. URL: https://github.com/ngtcp2/ngtcp2.

[28] *quiche*. [Online; accessed 18. Feb. 2025]. URL: https://github.com/cloudflare/quiche.

[29] *quic-go*. [Online; accessed 18. Feb. 2025]. URL: https://github.com/quic-go/quic-go.

[30] Toke Høiland-Jørgensen et al. "The express data path: Fast programmable packet processing in the operating system kernel". In: CoNEXT'18. 2018.

[31] Linux Foundation. *Data Plane Development Kit (DPDK)*. [Online; Accessed: March 13, 2025]. 2015. URL: http://www.dpdk.org.