Computer Communications 223 (2024) 90–100

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links

Marcel Kempf*, Benedikt Jaeger, Johannes Zirngibl, Kevin Ploch, Georg Carle

Chair of Network Architectures and Services, Technical University of Munich, Germany

ARTICLE INFO

Keywords:
QUIC
High-rate links
Performance evaluation
Transport protocols

ABSTRACT

QUIC is a new protocol standardized in 2021 designed to improve on the widely used TCP / TLS stack. The main goal is to speed up web traffic via HTTP, but it is also used in other areas like tunneling. Based on UDP, it offers features like reliable in-order delivery, flow and congestion control, stream-based multiplexing, and always-on encryption using TLS 1.3.

Unlike TCP, QUIC integrates these capabilities in user space, relying on kernel interaction solely for UDP. Operating in user space allows more flexibility but sacrifices some kernel-level efficiency and optimization that TCP benefits from. Various QUIC implementations exist, each distinct in programming language, architecture, and design.

QUIC is already widely deployed on the Internet and has been evaluated, focussing on low latency, interoperability, and standard compliance. However, benchmarks on high-speed network links are still scarce.

This paper presents an extension to the QUIC Interop Runner, a framework for testing the interoperability of QUIC implementations. Our contribution enables reproducible QUIC benchmarks on dedicated hardware and high-speed links. We provide results on 10G links, including multiple implementations, evaluate how OS features like buffer sizes and NIC offloading impact QUIC performance, and show which data rates can be achieved with QUIC compared to TCP. Moreover, we analyze different CPUs and CPU architectures influence reproducible and comparable performance measurements. Furthermore, our framework can be applied to evaluate the effects of future improvements to the protocol or the OS.

Our results show that QUIC performance varies widely between client and server implementations from around 50 Mbit/s to over 6000 Mbit/s. We show that the OS generally sets the default buffer size too small. Based on our findings, the buffer size should be increased by at least an order of magnitude. Our profiling analysis identifies Packet I/O as the most expensive task for QUIC implementations. Furthermore, QUIC benefits less from AES NI hardware acceleration while both features improve the goodput of TCP to around 8000 Mbit/s. The lack of support for NIC offloading from QUIC implementations results in missed opportunities for performance improvement.

The assessment of CPUs from different vendors and generations revealed significant performance variations. We employed core pinning to examine if the performance of QUIC implementations is affected by the allocation to specific CPU cores. The results indicated an increased goodput of up to 20% when running on a specifically chosen core compared to a randomly assigned core. This outcome highlights the impact of CPU core selection on the performance of QUIC implementations but also for reproducible measurements.

1. Introduction

QUIC is a general-purpose protocol that combines transport layer functionality, encryption through TLS 1.3, and features from the application layer. Proposed and initially deployed by Google [1,2], it was finally standardized by the Internet Engineering Task Force (IETF) in 2021 [3] after more than five years of discussion. Like TCP, it is connection-oriented, reliable, and provides flow and congestion control in its initial design. An extension for unreliable data transmission was added in RFC9221 [4].

One goal of QUIC is to improve web communication with HTTPS, which is currently using TCP / TLS as underlying protocols. It achieves

* Corresponding author.
E-mail addresses: kempfm@net.in.tum.de (M. Kempf), jaeger@net.in.tum.de (B. Jaeger), zirngibl@net.in.tum.de (J. Zirngibl), plochk@net.in.tum.de (K. Ploch), carle@net.in.tum.de (G. Carle).

https://doi.org/10.1016/j.comcom.2024.04.038
Received 31 December 2023; Received in revised form 22 April 2024; Accepted 30 April 2024
Available online 11 May 2024

this by accelerating connection build-up with faster handshakes, allowing only ciphers considered secure, and fixing the head-of-line blocking problem with HTTP/2. The transport layer handshake is directly combined with a TLS handshake allowing 0- and 1-RTT connection establishment.

Another goal of QUIC is to prevent network ossification. Software and hardware components are optimized for and sometimes limited to specific protocol stacks. For example, the Stream Control Transmission Protocol (SCTP) [5] targets several of the same optimizations as QUIC and was already proposed in 2000. However, it failed due to lacking support on endpoints and middleboxes. Besides TCP and UDP, no other transport protocols are practically useable in the Internet due to middleboxes such as firewalls and Network Address Translation (NAT) blocking other protocols.

To comply with currently deployed network devices and mechanisms, QUIC relies on the established and widely supported transport protocol UDP. The usage of UDP allows the implementation of QUIC libraries in user space. Thus, QUIC can initially be deployed without requiring new infrastructure, and libraries can be easily implemented, updated, and shipped.

On the one hand, this resulted in a variety of implementations based on different programming languages and paradigms [6]. On the other hand, previous efforts to optimize the performance of existing protocols have to be applied to new libraries, kernel optimizations cannot be used to the same degree, and the negative impact of encryption on performance has to be considered. Additionally, the heterogeneity among the QUIC implementations requires a consistent measurement environment for a reproducible evaluation. The performance differences between QUIC and TCP / TLS become more evident when the implementations are pushed to their limits on high-speed networks.

In this work, we show the impact of these effects through a fine-grained analysis of different QUIC implementations. We extend the existing QIR, a framework for interoperability testing of QUIC implementations [7,8]. Using Docker containers, it primarily focuses on functional correctness. Thus, we extend it to allow for performance-oriented measurements on bare metal.

Our contributions in this paper are:

(i) We develop and publish a **measurement framework based on the existing QIR framework**. The measurements are run on real hardware, and more metrics are collected, allowing an in-depth analysis of performance bottlenecks. Besides our code, all configurations and the collected data are published along with this paper. With this, we want to foster reproducibility and allow other researchers and library maintainers to evaluate different QUIC libraries and test potential performance optimizations.

(ii) We perform a **performance evaluation of the most widely used QUIC implementations** in interoperation on a 10G link with the proposed framework. Tested implementations show a wide diversity in their configuration and behavior.

(iii) We **evaluate impacting factors on the performance** of QUIC-based data transmissions. We analyze the effect of, *e.g.*, different buffer sizes, cryptography, and offloading technologies. This shows how goodput can be increased beyond the default setting and reveals unused potential for performance improvements from QUIC implementations.

(iv) We **analyze the impact of different hardware and CPU architectures** on the performance of QUIC implementations. We show that the performance of QUIC implementations highly depends on the underlying hardware. Depending on the CPU architecture, core pinning ensures a more stable performance and reproducibility and can drastically change the performance of QUIC implementations.

We explain relevant background regarding QUIC in Section 2. In Section 3, we introduce the implemented measurement framework and used configurations. We present our findings and evaluations in Section 4. Section 5 contains an outline of related work. Finally, we discuss our findings and conclude in Section 6.

2. Background

This section introduces relevant background for various properties impacting QUIC performance, as shown in Section 4. We identify components relevant to the overall performance and highlight the main differences compared to TCP / TLS. They include the always-on encryption in QUIC, the different acknowledgment (ACK) handling, the involved buffers in the network stack, and offloading functionalities supported by the NIC.

2.1. Encryption

QUIC relies on TLS version 1.3, which reduces available cipher suites to only four compared to previous TLS versions. Only ciphers supporting authenticated encryption with additional data (AEAD) are allowed by the RFC [9]. AEAD encrypts the QUIC packet payload while authenticating both the payload and the unencrypted header. Supported ciphers either rely on AES, a block cipher with hardware acceleration on most modern CPUs, or ChaCha20, a stream cipher that is more efficient than AES without hardware acceleration.[1]

Considering TLS in combination with TCP, data is encrypted into so-called TLS records, which are, in general, larger than individual TCP segments spanning over multiple packets. TCP handles packetization and the reliable, in-order transfer of the record before it is reassembled and decrypted at the receiver. With QUIC, each packet must be sent and encrypted individually. When receiving a packet, it is first decrypted before data streams can be reordered. Loss detection and retransmissions are done on packet- and not stream-level using the packet number similar to TCP's sequence number.

Additionally, QUIC adds another layer of protection to the header data, called header protection. Fields of the QUIC header, like the packet number, are encrypted again after packet protection has been applied, leading to twice as many encryption and decryption operations per packet.

2.2. Acknowledgments

Since QUIC provides reliability and stream orientation, it requires a similar ACK process as TCP. QUIC encapsulates QUIC packets into UDP datagrams while the packets carry the actual payload as QUIC frames. Different frame types exist, such as *stream*, *ACK*, *padding*, or *crypto*.

An *ACK* frame contains so-called ACK ranges, acknowledging multiple packets and ranges simultaneously, potentially covering multiple missing packets. All received packets are acknowledged. However, *ACK* frames are only sent after receiving an ACK-eliciting packet. A QUIC packet is called ACK-eliciting if it contains at least one frame other than *ACK*, *padding*, or *connection close*.

Other than TCP, QUIC sends ACKs encrypted, inducing additional cryptographic workload both on the sending and receiving side. Therefore, QUIC must determine how frequently ACKs are sent. The *maximum ACK delay* transport parameter defines the time the receiver can wait before sending an *ACK* frame [3]. Determining the acknowledgment frequency is a trade-off and may affect the protocol's performance. Fewer ACKs can lead to blocked connections due to retransmissions and flow control, while more put additional load on the endpoints.

Furthermore, sending and receiving ACKs with QUIC is more expensive than with TCP. With TCP, ACKs do not take any additional space since they are part of the TCP header and thus can be piggybacked on sent data. The kernel evaluates them before any cryptography is performed. We show the impact of acknowledgment processing and sending in Section 4.

[1] https://datatracker.ietf.org/doc/html/rfc8439#appendix-B.

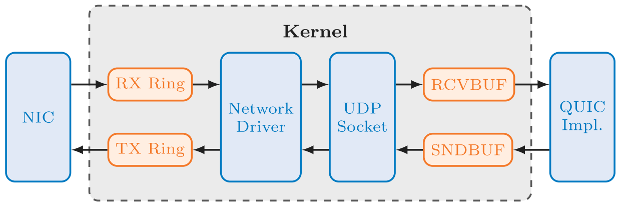


Fig. 1. Simplified overview of available system buffers relevant for a QUIC implementation.

2.3. Buffers

Different buffers of the system can impact the performance of QUIC. Fig. 1 shows a simplified schema of buffers managed by the kernel, which are relevant for the measurement scenarios of this paper. The NIC stores received frames to the RX ring buffer using Direct Memory Access (DMA) without kernel interrupts. Conversely, it reads frames from the TX buffer and sends them out. The kernel reads and writes to those ring buffers based on interrupts and parses or adds headers of layers 2 to 4, respectively. Afterward, the UDP socket writes the payload to the Receive Buffer (RCVBUF).

However, if the ring buffer or RCVBUF is full, packets are dropped, and loss occurs from the perspective of the QUIC implementation. The default size of the receive buffer in the used Linux kernels is 208 KiB. In contrast, if the Send Buffer (SNDBUF), which resides between the implementation and the socket, is packed, the QUIC implementation will be blocked until space is available. Both scenarios reduce the goodput. We show the impact of different buffer sizes in Section 4.1.

2.4. Offloading

The Linux kernel offers different NIC offloading techniques, namely TSO, GSO, and GRO. While the first only affects TCP, the others also apply to UDP and thus QUIC. The main idea of all offloading techniques is to combine multiple segments and thus reduce the overhead of processing packets. Following the ISO / OSI model, data sent by TCP is supposed to be split into chunks smaller or equal to the Maximum Segment Size (MSS) and then passed down to lower layers. Each layer adds its header and passes the data further. All of this is computed by the kernel. With hardware offloading, this segmentation task can be outsourced to the NIC. Offloading can only accelerate sending and receiving of packets, not the data transfer over the network [10].

So far, most offloading functions are optimized regarding TCP. QUIC profits less since packetization is done in user space. Utilizing offloading would require adjustments in the implementation to offload tasks like cryptography or segmentation [11]. In Section 4.5, we compare the impact of different offloading functions on QUIC and TCP / TLS.

3. Measurement framework

In this paper, we extend QIR initially proposed by Seemann and Iyengar [7]. It is designed to test different QUIC implementations for interoperability and compliance with the QUIC standard reporting results on a website [8]. Servers and clients from multiple implementations are tested against each other, performing various test cases, like handshake, 0-RTT, and session resumption. The main focus is on the implementations' functional correctness and less on performance. Even though all followed the same RFC drafts during specification, interoperability between different servers and clients was only present for some features. Implementations run inside Docker containers, and the network in between is simulated using *ns-3*.[2] Simple goodput measurements are available but only conducted on a 10 Mbit/s link. As of December 2023, nearly all tested libraries are close to the possible maximum goodput [8].

QIR orchestrates the measurements by configuring used client and server applications, creating required directories for certificates and files, and collecting log files and results afterward. In the sense of reproducibility, these features make QIR a powerful tool for QUIC (but not limited to) evaluations.

We extended QIR to enable high-speed network measurements on dedicated hardware servers. Fig. 2 shows an overview of our framework and its features. For this work, we use different physical servers for the client and server implementation to prevent them from impacting each other. We add additional configuration parameters, such that measurements can be specified with a single configuration file, and include version fingerprints both from the implementations and QIR to the output to foster reproducibility. Additionally, we extend the logging by including several tools which collect data from different components (*e.g.*, the NIC, CPU, and sockets).

We follow these three requirements in the implementation of our measurement framework:

Flexibility: Any QUIC implementation can be used as long as it follows the required interface regarding how client / server are started and variables are passed to them. We provide example implementations for the ones given in Section 3.4.

Portability: The framework can be deployed to any hardware infrastructure, as depicted in Fig. 2, requiring a link between the client and server nodes and `ssh` access from the management node.

Reproducibility: Experiments are specified in configuration files and results contain needed information on how they were generated, *e.g.*, the QIR and QUIC implementation version. Additionally, results contain a complete description of the used configuration, versions, and hardware.

Our code, results, and analysis scripts are available:
https://github.com/tumi8/quic-10g-paper

This includes our extension of QIR, all measurement configurations, and results shown in the paper, analysis scripts to parse the results, and Jupyter notebooks for visualization [12].

3.1. Workflow

We followed a similar workflow as QIR. First, our framework configures the used hardware nodes, especially the used network interfaces for the measurements. Each measurement executes four different scripts that the implementation or configuration can adjust: setup environment, pre-scripts, run-scripts, post-scripts as shown in Fig. 2. The setup script can be used to install local dependencies like Python environments. Pre- and post-scripts can be utilized to configure OS-level properties like the UDP buffer size and reset them after the measurement. Also, additional monitoring scripts can be started and stopped accordingly.

Relevant configuration parameters for the client / server implementation, such as IP address, port, and X.509 certificate files, are passed via environment variables. Then the server application is started, waiting for incoming connections. Subsequently, the client application conducts a QUIC handshake and requests a file from the server via HTTP/3. MsQuic is only a QUIC library and does not provide an HTTP/3 implementation. Therefore, we use HTTP/0.9 if at least one endpoint is running MsQuic. We argue that this insignificantly impacts our measurements, because we focus on single downloads of large files and no additional features of HTTP/3. Once the client terminates, the framework checks whether all files were transferred successfully and executes post-scripts to stop monitoring tools and to reset the environment to a clean state for subsequent measurements. Finally, the framework collects all logs.

[2] https://www.nsnam.org/.

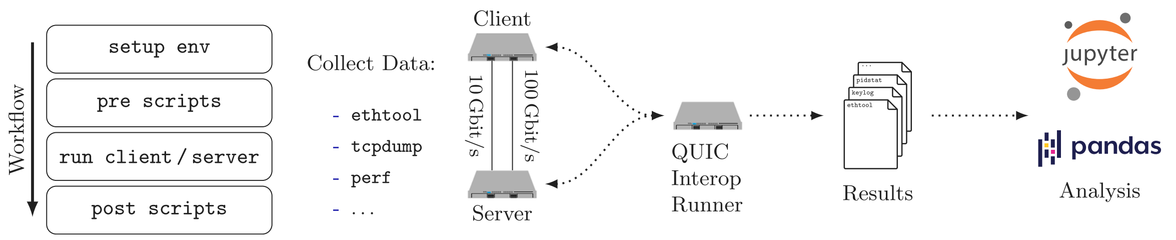


Fig. 2. Hardware architecture, measurement workflow, and analysis pipeline.

3.2. Hardware configuration

We decided to run the client and server applications on different hardware nodes to prevent interference and to fully include the kernel and NIC (other than with Docker). The QIR runs on another host and functions as a management node orchestrating the measurement. We use *pos* [13] to orchestrate the experiments, setting up the nodes and executing the measurement scripts in an automated way. This allows us to deploy the framework on different hardware infrastructures and to easily change the configuration while ensuring reproducibility. The management node can access the measurement nodes via `ssh`, while the others are connected via a 10 Gbit/s and a 100 Gbit/s link, as shown in Fig. 2. Different network interfaces and links are used for management and measurements. If not stated differently, all measurements were executed on *AMD EPYC 7543 32-Core* processors, 512 GB memory, and *Broadcom BCM57416 NICs*. As an operating system, we use *Debian Bullseye* on *5.10.0-8-amd64* for all measurements without additional configurations. We rely on live-boot images with RAM disks, drastically increasing I/O speed to focus on the network aspect.

3.3. Collected data and analysis

For each measurement, the framework computes the goodput as the size of the transmitted file divided by the time it takes to transmit it. Timestamps to calculate the transmission time are taken just before and after starting the client implementation. Thus, it includes the client startup, required handshakes and any final steps done after the download. To reduce this impact of transmission rate fluctuations (*e.g.*, caused by congestion control), a large file size (8 GB) is chosen to enforce a connection duration of several seconds.

The following monitoring tools are directly integrated into our framework to collect more data from within the implementations or from the hardware hosts. **tcpdump** is used to collect packet traces which can be decrypted with the exported session keys. Additionally, implementations can enable **qlog** (a schema for logging internal QUIC events [14]) and save results to a directory set up and collected by the framework. However, both result in extensive logging that heavily impacts performance and are only considered for debugging. **ifstat**, **ethtool**, **netstat**, and **pidstat** can be started to collect additional metrics from the NIC and CPU, such as the number of dropped packets or context switches. Finally, **perf** can be used for in-detail client and server application profiling. This allows to analyze how much CPU time is consumed for tasks like encryption, sending, receiving, and parsing packets. When enabled, the output of the used tools is exported along with the measurement results.

We provide a parsing script for all generated data that handles all different output formats of the used tools. By this, it is possible to analyze the change of goodput with different configurations and get a better understanding of why this is the case. It is possible to detect client and server-side bottlenecks as root causes for performance limitations. The variety of collected information allows to analyze how many packets are sent by the client, how high the CPU utilization of the server is, or how many bytes are sent overall. We consider this important since we observed multiple different QUIC components limiting the performance depending on the configuration. The analysis pipeline parses available result files and outputs the final results as Python *pandas* DataFrame.

3.4. Implementations

Since there is not one widely used implementation for QUIC (such as Linux for TCP), we evaluate multiple implementations written in different languages. The implemented framework currently includes eight QUIC libraries namely: aioquic [15], quic-go [16], NGINX [17], MsQuic by Microsoft [18], mvfst by Facebook [19], picoquic by Private Octopus [20], quiche by Cloudflare [21], and LSQUIC by LiteSpeed Tech [22]. For each implementation, we either used provided examples for the client and server or made minor adjustments to be compatible with QIR.

In this paper, we mainly focus on LSQUIC and quiche since they show high goodput rates, as shown in Section 4.3. They are implemented in Rust and C, respectively, and are already widely deployed [23]. Furthermore, both libraries rely on BoringSSL for TLS. Thus, cryptographic operations and TLS primitives are comparable, and we can focus on QUIC specifics in the following. We compare them to the remaining libraries in Section 4.3 to put their initial performance into perspective and further support our selection. We base our initial client and server on their example implementation and adapt only where necessary. MsQuic does also show high goodput rates, but as it does not support HTTP/3 and is not based on BoringSSL, it is only included in parts of the evaluation for better comparability. Additionally, we compare QUIC with a TCP / TLS stack consisting of a server using NGINX (version 1.18.0) and a client using `wget`.

4. Evaluation

We apply the measurement framework to evaluate the performance of different QUIC implementations. For every measurement, the client downloads an 8 GB file via HTTP/3. As MsQuic is purely a QUIC library and does not support HTTP/3, we use the Application-Layer Protocol Negotiation (ALPN) token `hq-interop` for all measurements where at least one endpoint runs MsQuic. Every measurement is repeated 50 times to assure repeatability, if not stated otherwise. The boxplots show the median as a horizontal line, the mean as icons such as ▲, and the quartiles Q_1 and Q_3. All implementations come with different available Congestion Control (CC) algorithms. For LSQUIC, we observed unintended behavior with *BBR*, resulting in several retransmissions and only between 30% to 70% of the goodput of *Cubic*. We assume this is related to a known issue with *BBR* in TCP [24]. To prevent significant impact from different algorithms, we set each implementation to *Cubic* if available or *Reno* otherwise.

4.1. Buffers

As explained in Section 2.3, different system buffers affect a QUIC connection. The essential buffers are the ring buffers in-between the NIC and network driver, and the send and receive buffers from the socket used by the library, as shown in Fig. 1.

To analyze the impact of these buffers, the measurement framework captures network driver statistics before and after each data transmission using **ethtool** and connection statistics using **netstat**. It provides the number of sent and received packets and dropped packets in each of the mentioned buffers.

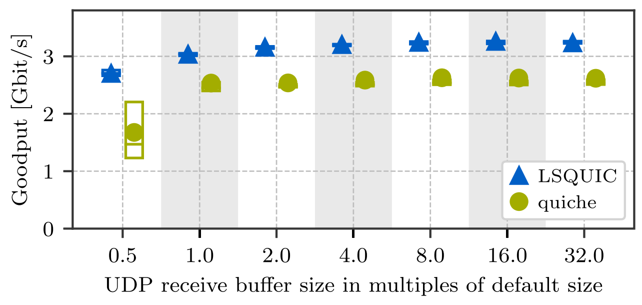


Fig. 3. QUIC goodput with different UDP receive buffer sizes. X-axis values are multiples of the default buffer size of 208 KiB.

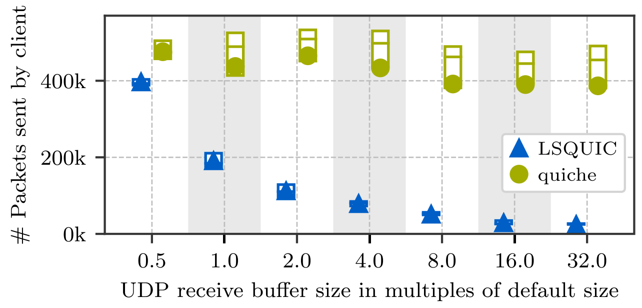


Fig. 4. Number of sent ACKs by the client. X-axis values are multiples of the default buffer size of 208 KiB.

A baseline measurement shows that the ring buffers drop no data. However, packets are dropped by the client receive buffer since both client implementations retrieve packets slower than they arrive. As a result, retransmissions are required by the server, and congestion control impacts the transmission. To analyze the impact on the goodput, we increase the default receive buffer size (208 KiB). The effect of this can be seen in Fig. 3. It shows the goodput of LSQUIC and quiche pairs for different buffer sizes as multiple of the default buffer on the x-axis. The goodput for both libraries improves with increasing buffer sizes and stabilizes after a 16-fold increase.

Using LSQUIC with the default buffer size results in 11.4 k dropped packets and a loss rate of 0.2%. With the largest tested buffer size, LSQUIC reaches a goodput of 3250 Mbit/s, an 7% increase compared to the default buffer size. Besides the reduced loss and retransmissions, the number of ACKs sent by the LSQUIC client is drastically reduced from 180 k to 46 k (26%). This development for all tested buffer sizes is shown in Fig. 4. The reduction of sent ACK packets also results in reduced CPU usage by the client while increasing the server CPU usage shifting the bottleneck further towards the sending of QUIC packets. In all scenarios, LSQUIC sends fewer ACKs than reported by Marx et al. [25].

Regarding quiche, only 7 k packets are dropped with the default buffer size, hence a 0.1% loss rate. Larger buffers decrease the loss by one order of magnitude and increase the goodput but only by 3% to 2530 Mbit/s. In contrast to LSQUIC, no difference regarding sent ACKs can be seen (see Fig. 4). Reducing the receive buffer further impacts both libraries, especially quiche. The goodput drops by 40% and a more significant deviation is visible.

It is important to note that the default and maximum receive buffer size is different for each operating system and architecture. During our research, we found that the values for the default and maximum receive buffer size vary from 208 KiB for current Debian versions up to 16 MiB for current Android versions. However, most of the evaluated combinations of different operating systems and architectures use a default buffer size of less than 1 MiB.

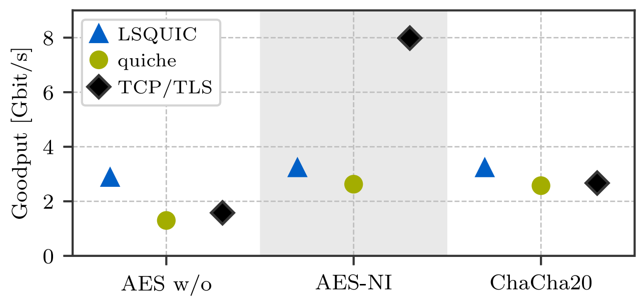


Fig. 5. Impact of different TLS ciphers on QUIC and TCP / TLS goodput. During the AES measurement without hardware acceleration, the implementations were forced to use AES by deactivating ChaCha20 in the respective TLS library to prevent the fallback.

Key take-away: *The UDP receive buffer size has a visible impact on QUIC-based data transmission, as it is often undersized by default. We recommend a general increase of the default buffer by at least an order of magnitude to support widespread deployment of QUIC.*

4.2. Crypto

As explained in Section 2.1, QUIC supports TLS using AES or ChaCha20. Generally, operations to encrypt and decrypt are computationally expensive but are widely optimized in hardware and software today. QUIC and TLS 1.3 only support AEAD algorithms [26], which can encrypt and sign data in one single pass. For AES, only three ciphers are available: `AEAD_AES_128_GCM`, `AEAD_AES_128_CCM`, and `AEAD_AES_256_GCM`. From them, only `GCM` ciphers are required or should be implemented [26], the `CCM` cipher is rarely used. Usually, the 128 bit `GCM` cipher is preferred [23]. For the following evaluation, we use the default cipher. ChaCha20 always uses the `Poly1305` authenticator to compute message authentication codes.

We evaluated the QUIC implementations in three different scenarios. Two of them use AES either without or with the AES New Instruction Set that offers full hardware acceleration [27] (which is the default in our test environment), and one scenario uses ChaCha20. For the following, we refer to the hardware-accelerated AES version as AES-NI. LSQUIC and quiche automatically fall back to ChaCha20 whenever hardware acceleration is unavailable. To evaluate AES without hardware acceleration, we patched the used `BoringSSL` library accordingly.

As seen in Fig. 5, ChaCha20 achieves the same throughput as AES-NI for both QUIC libraries. While removing hardware acceleration decreases the goodput with AES by 11% for LSQUIC and even by 51% for quiche. This difference between the implementations results from the quiche client sending more than 16 times as many ACKs than LSQUIC. While the server is the bottleneck in all measurements, more packets sent by the client add additional load to the server for receiving and decrypting packets (see Section 4.1). This decryption is more expensive without hardware acceleration. Thus, crypto has a higher impact on the overall goodput. Also, we observe that the client CPU utilization of quiche is 5% lower with ChaCha20 than with AES.

With TCP / TLS, the effect changes. While ChaCha20 reaches higher goodput rates than AES without hardware acceleration, AES-NI outperforms ChaCha20 with a three times higher goodput, almost reaching the link rate. Additionally, for TCP / TLS we observed that the client CPU bottlenecked the connection for AES-NI and ChaCha20, while the server was the bottleneck without hardware acceleration.

Key take-away: *While selecting an appropriate cipher suite drastically impacts TCP / TLS, QUIC reaches similar goodput for AES-NI and the ChaCha20-based cipher. The acceleration effect is further reduced due to smaller TLS records and encrypted ACKs.*

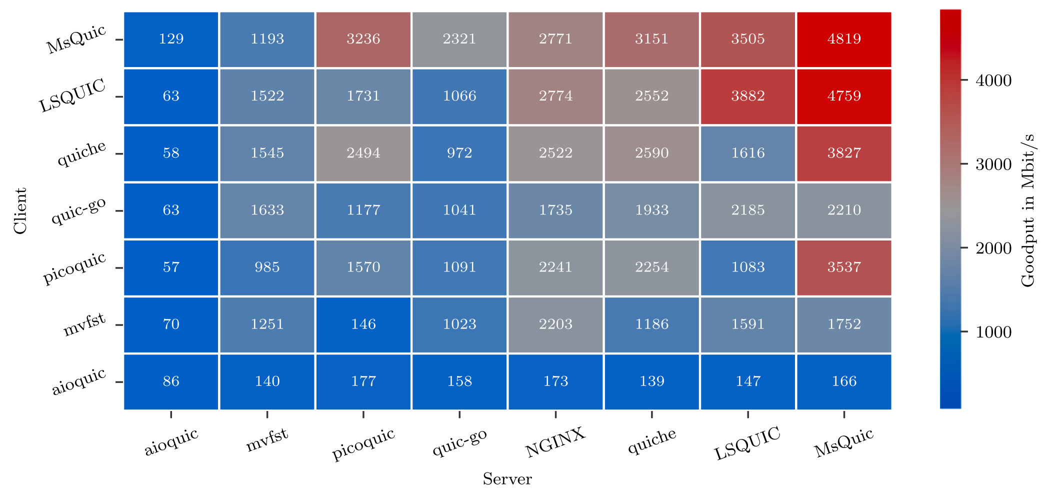


Fig. 6. Goodput results for different QUIC libraries tested against each other on a 10 Gbit/s link. Every combination was repeated 20 times. In comparison TCP / TLS reaches 8010 Mbit/s. The UDP receive buffer is increased by a factor of 32 on both endpoints to avoid packet loss as described in Section 4.1.

4.3. Implementation comparison

QIR is designed to test operability between different implementations. We evaluate the client and server of the implementations listed in Section 3.4. Fig. 6 shows the goodput for each client–server pair for all implementations. It clearly shows that the goodput widely differs (57 Mbit/s to 4819 Mbit/s) between the implementations.

The Python library aioquic shows the lowest performance, as expected since it is the only interpreted and not compiled implementation. It can be considered an implementation suitable for functional evaluation or a research implementation. When it comes to goodput, it can be neglected. The three best-performing implementations are LSQUIC, quiche, and MsQuic. Regarding picoquic, we are not able to reach similar goodput as reported by Tyunyayev et al. [28] and were not able to easily reproduce the required optimizations. Therefore, we stick with LSQUIC-LSQUIC and quiche-quiche pairs for the following evaluation since they achieved the highest goodput and both support HTTP/3. We also use MsQuic-MsQuic pairs for the following measurements to obtain a more detailed analysis.

Like interoperability of QUIC features, goodput performance widely varies among client–server pairs. We conjecture that different operation modes, QUIC parameters, and efficiency of the used components result in these fluctuations. The LSQUIC server achieves the highest rate with the LSQUIC client while being less performant with clients of other implementations. Besides MsQuic, the quiche and NGINX servers achieves decent goodput with most clients. The picoquic-mvfst measurements achieve peculiar low rates of only 146 Mbit/s.

Looking at the two combinations of LSQUIC and quiche, the goodput is more than 1.5 times as high with a quiche server and LSQUIC client compared to the other way around. After analyzing at the collected metrics, we found that the performance of heterogeneous server-client combinations is usually not CPU-limited. With LSQUIC-LSQUIC and quiche-quiche pairs, the performance in our measurements is constantly CPU limited, which is shown by a CPU utilization of the server of more than 99% in average during the connection. The combination of LSQUIC as server and quiche as client only reaches a server CPU utilization of 70% in average with the client CPU also not reaching more than 75% CPU utilization. The number of packets sent by the client, which corresponds to the number of acknowledgments in our measurements, is also significantly lower in this combination, staying below 10 k. In the measurements with homogeneous pairs of LSQUIC and quiche, we observed the LSQUIC client sending around 30 k packets and the quiche client even sending around 450 k packets. All observations suggest that the flow control mechanisms of the libraries are not well aligned and lack performance optimization in interoperation with other QUIC libraries.

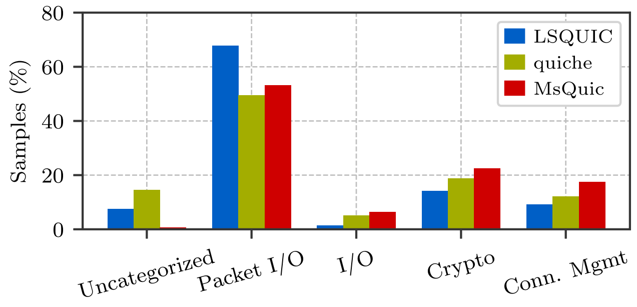


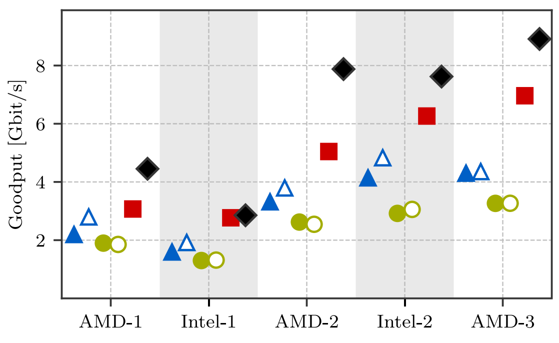
Fig. 7. Distribution of server perf samples across different categories. The results are shown in relation to the total number of samples collected by perf for the respective QUIC server application (excluding idle states).

Key take-away: *The goodput of different QUIC libraries varies drastically, not only between libraries in general but also between clients and servers. LSQUIC, quiche, and MsQuic perform best in this scenario. Only in the case of LSQUIC as server and quiche as client the goodput drops below 2.5 Gbit/s.*

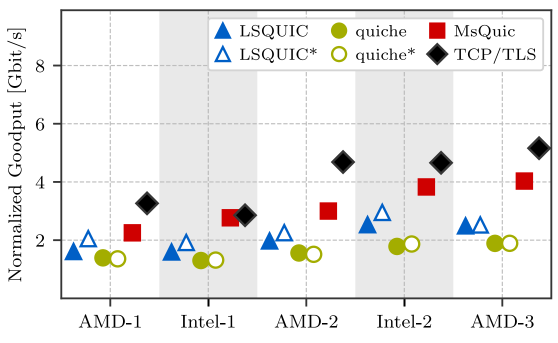
4.4. Performance profiling

We use **perf** to analyze LSQUIC, quiche, and MsQuic further and evaluate the CPU time consumption for each component. During each measurement, perf collects samples for the complete system. We categorize the samples for both implementations based on their function names and a comparison to the source code. Fig. 7 shows the results for the respective server implementations. *Packet I/O* covers sending and receiving messages, especially the interplay with the UDP socket and kernel functionality (sendmsg and recvmsg). *I/O* covers reading and writing the transmitted file by server and client. *Crypto* describes all TLS-related en- / decryption tasks. *Connection Management* covers packet and acknowledgment processing and managing connection states and streams. If no function name can be collected by **perf** or if we cannot clearly map it to a category, we use *Uncategorized*.

We manually created this mapping by assigning each function occurring in the perf dump to a category. While we could not assign all functions, a clear trend is visible. The call stacks of the samples recorded with quiche do often not contain enough information to map them to a specific category, especially when octets are handled.



(a) Absolute goodput values



(b) Goodput normalized to the minimum CPU frequency of 2.2 GHz

Fig. 8. Goodput as measured on different hardware architectures listed in Table 1. LSQUIC* and quiche* are built with compile flags to optimize for the used architecture.

Although a clear mapping is not possible in these cases, the use of these functions in the various modules of the source code suggests that a large share of the uncategorized samples belongs to *Packet I/O*. The strict inclusion of TLS is often criticized due to the expectation of a high overhead [29]. However, *Packet I/O* takes up a majority of resources for all three libraries. Passing packets from the libraries in user space to the kernel, the NIC, and vice versa currently impacts the performance most.

Interestingly, we see differences in performance in the used programming language or architecture and between the three client and server combinations of different implementations. Resulting in the highest difference, the goodput with a quiche server and LSQUIC client is more than twice as high as vice versa. This also shows, that even when the same protocol is implemented in languages of comparable efficiency (C, Rust), the overall behavior does differ. During these measurements, both the client and server are only at around 70% CPU usage, no loss is visible, and no additional retransmission occurs. In this case, the bottleneck seems to be the interaction of the flow control mechanisms of both libraries in this client / server constellation. We also noticed that the MsQuic pair is client limited, as the server only reaches around 80% CPU usage in our measurements, while the client is around 100%. Both LSQUIC and quiche are server limited in our measurements with a server CPU usage above 95%. Besides general interoperability tests, our measurement framework can be used in the future to identify these scenarios and to improve QUIC libraries, their flow and congestion control mechanisms, and their interaction.

Key take-away: *Our findings show that the most expensive task for QUIC is Packet I/O. While the cost of crypto operations is visible, it is not the main bottleneck here. Overall, our results comply with related work [11], and we share our mappings so that they can be refined and extended in the future.*

4.5. Segmentation offloading

To reduce the impact of packet I/O, a QUIC library can combine multiple packets and rely on offloading. Similarly, an implementation can perform batching by using the *sendmmsg*[3] and *recvmmsg*[4] system calls. This reduces the amount of system calls as multiple packets are moved to/from the kernel space within a single buffer.

By default, Generic Segmentation Offload (GSO), Generic Receive Offload (GRO), and TCP Segmentation Offload (TSO) are activated in Linux. We analyze which offload impacts QUIC and TCP by incrementally activating different offloading techniques. For this measurement, we use a slightly modified version of the MsQuic library. The Linux datapath of MsQuic did not recognize GRO support in our setup, which is why we enforced GRO usage. This results in a slightly increased

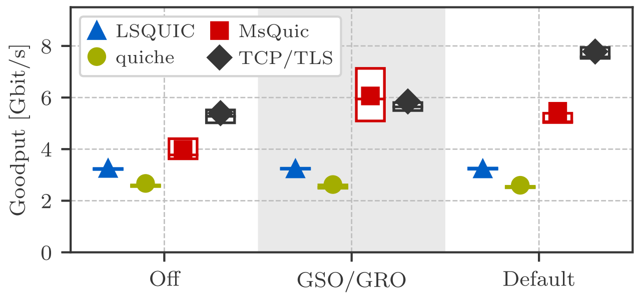


Fig. 9. Impact of hardware offloading on QUIC and TCP / TLS goodput. By default, GSO, GRO, and TSO are activated.

performance with MsQuic compared to the previous measurements. Fig. 9 indicates that all the offloading techniques hardly influence the goodput of LSQUIC and quiche, while MsQuic and TCP largely profit when they are available. MsQuic achieves between 35% and 50% higher goodput with GSO and GRO enabled. The achieved goodput of LSQUIC and quiche does not change when GSO / GRO is enabled. However, the client CPU utilization increases from 82% to 92% for LSQUIC and from 77% to 84% for quiche. Turning off all offloads does not decrease goodput but decreases the client's CPU utilization and also power consumption for LSQUIC and quiche.

Analyzing the source code of all three QUIC implementations has revealed that only MsQuic supports offloading and therefore benefits from it.

While LSQUIC implements functionality to support *sendmmsg* and *recvmmsg*, we were not able to use it as of December 2023. Data transmission with the functionality activated randomly terminated with exceptions. MsQuic also supports *sendmmsg* and *recvmmsg*, which were used in the measurements where offloading was deactivated. We expect improved support for those features also by other implementations and suggest reevaluating libraries with our framework in the future.

Key take-away: *Two of the three tested QUIC implementations do not profit from any segmentation offloading techniques as of December 2023. Compared with TCP, there is much room for improvement to apply the same benefits to UDP and QUIC. Since QUIC encrypts every packet individually, including parts of the header, techniques similar to TSO cannot be applied, which would require that headers can be generated in the offloading function. However, with adjustments to the protocol and the offloading functions, speedups can be achieved for segmentation and crypto offloading [11].*

4.6. Hardware

Even though an implementation in user space comes with more flexibility, tasks that are done by the kernel for TCP become more expensive with QUIC. More CPU cycles are required per packet.

[3] https://man7.org/linux/man-pages/man2/sendmmsg.2.html.
[4] https://man7.org/linux/man-pages/man2/recvmmsg.2.html.

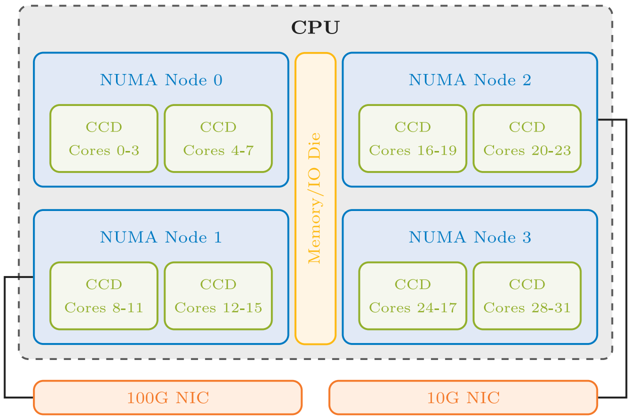


Fig. 10. Simplified overview of the AMD EPYC 7543 CPU architecture including the NICs from our setup. The available PCIe lanes are split to the four NUMA nodes, resulting in the two NICs being connected to different NUMA nodes.

Table 1
Different CPUs used for the measurements. The default for measurements was AMD-2 if not noted otherwise.

	CPU	Year	max GHz
AMD-1	AMD EPYC 7551P	2017	3.0
AMD-2	AMD EPYC 7543	2021	3.7
AMD-3	AMD EPYC 9354	2022	3.8
Intel-1	Intel Xeon CPU D-1518	2015	2.2
Intel-2	Intel Xeon Gold 6312U	2021	3.6

We repeated the final goodput measurements with increased buffers on five host pairs with different generations of AMD and Intel CPUs listed in Table 1. Client and server hosts are equipped with the same CPU for each measurement.

Additionally, we optimized LSQUIC and quiche further by adding compile flags to optimize for the used architecture. This results in the usage of additional instructions and optimizations for the respective CPU. The optimized implementations are referred to as LSQUIC* and quiche*.

The results in Fig. 8(a) show that quantizing QUIC throughput highly depends on the used CPU and architecture. Both QUIC and TCP profit from newer CPUs with more modern instruction sets, higher clock frequencies, and improved manufacturing technology. Compile flags improved the LSQUIC goodput by 11% to 20%, while they hardly affected quiche. Also, QUIC and TCP perform differently among the different architectures. For example, when comparing the two CPUs released in 2021, AMD-2 and Intel-2, QUIC performs between 11% and 27% better on the Intel chip with MsQuic reaching more than 6.2 Gbit/s. On the other side, TCP goodput decreases by 3% compared to the CPU in the AMD-2 host pair. This shows that QUIC (in the user space) and TCP (in the kernel) profit differently from CPU architectures and instruction sets. To further quantify the impact of improved microarchitectures, we normalized the goodput to the lowest maximum CPU frequency of 2.2 GHz in Fig. 8(b), as Raumer et al. [30] showed that the cost for packet processing scales almost linearly with the CPU frequency. Especially for the Intel CPUs, the newer Intel-2 CPU implementing the Sunny Cove microarchitecture and 10 nm process nodes shows a significant improvement in goodput compared to the older Intel-1 CPU, which relies on the Broadwell microarchitecture and 14 nm process nodes. Intel claims a 18% increase in instruction per cycle with the upgrade from their 14 nm to the newer 10 nm technology [31].

Key take-away: *The used hardware is highly relevant for evaluating QUIC performance. While not feasible for all research groups, we suggest attempting an evaluation of potential improvements to QUIC libraries on different CPU generations and frequencies in the future to better quantify their impact.*

4.7. CPU architecture

The results in the previous section show that the used CPU is highly relevant for the performance of QUIC. As different CPUs come with different numbers of cores and architectures, we analyze the impact of running QUIC on different cores. We use the **taskset** command to pin the client and server processes to different cores. A comparison to TCP is not possible as it is running in the kernel and therefore it is not possible to pin it to specific cores. The measurements are again performed on the default setup presented in Section 3.2. The AMD EPYC 7543 CPUs comes with 32 cores and 64 threads, using the Zen 3 architecture [32]. Detailed information about the CPU topology are retrieved with the tools provided by the *hwloc*[5] package. On the used CPU, cores are grouped into eight Core Chiplet Dies (CCDs), each consisting of four cores sharing a 32 MiB L3 cache. The CPU supports 1, 2, or 4 NUMA nodes. This can be set in the BIOS with 1 NUMA node per socket being the default setting on our system.

In an initial baseline measurement, we compare the default, one NUMA node per socket, to a configuration of four NUMA nodes per socket. We observed a 4% higher mean goodput over all measurements for MsQuic and a 7% increase for quiche. For LSQUIC, no differences are visible. We assume that this is caused by the OS not striping the memory across the NUMA nodes. As the setting with one NUMA node per socket does not only show worse performance but also adds a transparent layer and does therefore not allow for a performance comparison between different cores, we conducted our measurements with four NUMA nodes per socket.

We performed further measurements by pinning the server and client processes to cores 0, 8, 16, and 24 to cover all NUMA nodes. The results are shown in Figs. 11(a) to 11(c). Each colored box represents the mean of 20 repetitions with the client and server pinned to the respective core. While the mean goodput of those 20 repetitions is displayed in the first line of each box, the second line shows the deviation from the mean goodput over all 16 measurements.

For LSQUIC, we observe a significantly higher goodput when pinning the server to core 16. The CPU architecture of the used setup is shown in Fig. 10. It can be seen that the 10 Gbit/s NIC, which is the one used for the measurements, is connected NUMA node 2, explaining the higher goodput when pinning to core 16. The highest mean goodput of 3855 Mbit/s is achieved when pinning the client also to core 16. A quick look on the captured CPU utilization shows that the mean CPU utilization of the server is at 98% while the client is at 90%. This indicates that LSQUIC is mostly server-limited in our measurements and explains less impact of the core pinning on the client. MsQuic and quiche also achieve their highest goodput when pinning both endpoints to core 16. It can also be seen that with quiche, pinning the server to the right core but not the client will result in a lower goodput than not performing any core pinning at all. We suspect that the increased sending rate of the server leads to an overload of the client or side effects with ACKs.

The servers used in the measurement environment are additionally equipped with 100 Gbit/s NICs. While no QUIC implementation reaches the line rate of 10 Gbit/s, we repeated the measurements with the 100 Gbit/s NICs and the same setup because they are connected to NUMA node 1, as shown in Fig. 10.

At first sight, the results in Figs. 11(d) to 11(f) show a similar behavior to the measurements with the 10 Gbit/s NICs but with a different core. For LSQUIC, the differences are even more significant than in the measurements with 10 Gbit/s NICs. The mean goodput when pinning the server to core 8 is around 22% higher than the mean goodput over all measurements, which corresponds to the performance when no core pinning is performed.

[5] https://www.open-mpi.org/projects/hwloc/.

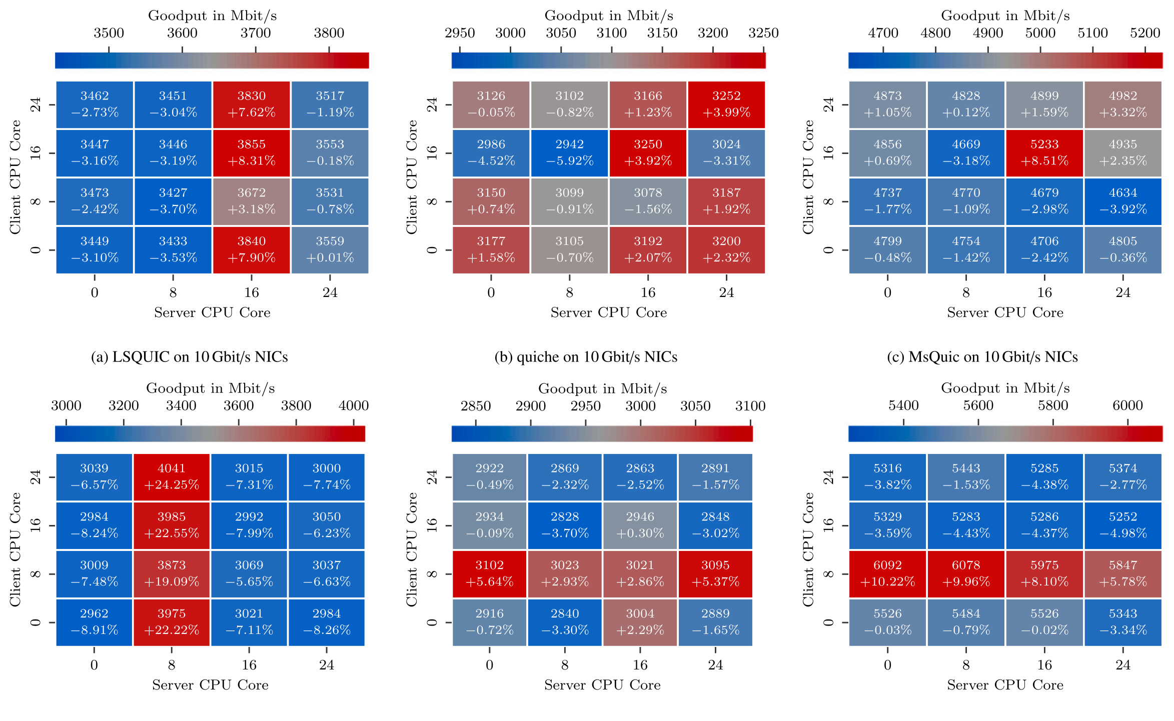


Fig. 11. Impact of core pinning on QUIC goodput. The second line shows the difference to the mean goodput over all 16 measurements. Every measurement was repeated 20 times. Measurements were performed with 10 Gbit/s NICs or 100 Gbit/s NICs and 4 NUMA nodes per socket.

For all three implementations, a column or row pattern is clearly visible. As discussed before, this indicates that LSQUIC is CPU limited on the server side while quiche and MsQuic are CPU limited on the client side in these measurements. In these measurements, core pinning on the endpoint that is not the bottleneck does not seem to have a significant impact.

Key take-away: *In order to conduct reproducible measurements, it is not sufficient to consider the hardware used. Depending on the CPU architecture, the performance of QUIC can vary significantly across different cores. NUMA can have a huge impact on the performance and should be considered when evaluating QUIC. Pinning QUIC to a specific core can lead to better or worse performance but causes more stable results.*

5. Related work

The interoperability of different QUIC implementations has been tested by research throughout the protocol specification. Seemann and Iyengar [7], Piraux et al. [33], and Marx et al. [25] developed different test scenarios and analyzed a variety of QUIC implementations. Furthermore, Rüth et al. [34], Piraux et al. [33], and Zirngibl et al. [23] have already shown a wide adoption of QUIC throughout the protocol specification and shortly before the release of RFC9000 [3]. They show that multiple large corporations are involved in the deployment of QUIC, various implementations are used, and different configurations can be seen (*e.g.*, transport parameters). In 2023, Zirngibl et al. [35] have further shown that a variety of QUIC implementations are actually used and deployed as servers on the Internet. They developed an approach to identify QUIC server libraries based on transport parameter orders and error messages, conducted Internet-wide measurements and identified at least one deployment for all libraries evaluated in this work. However, these studies mainly focused on functionality analyses, interoperability, and widespread deployments but did not focus on the performance of libraries.

Different related works analyzed the performance of QUIC implementations [11,28,36–41]. However, they either analyzed QUIC in early draft stages, *e.g.*, Megyesi et al. [38] in [38], or mainly focused on scenarios covering small web object downloads across multiple streams, *e.g.*, by Sander et al. [37] and Wolsing et al. [40]. Similar to our work, Yang et al. [11] orchestrated QUIC implementations to download a file. With a single stream, they reached a throughput between 325 Mbit/s and even 4121 Mbit/s for Quant. However, they omit HTTP/3 and only download a file of size 50 MB. Therefore, the impact of the handshake and cryptographic setup is higher, and other effects might be missed.

Zhang et al. [42] analyzed the performance of QUIC over high-speed networks. In contrast to our work, they used common browsers as clients. They identify receiver-side processing as one major issue and also suggest offloading and batching techniques like GRO and *recvmmsg*.

Endres et al. [43] adapted the QIR to evaluate QUIC implementations similar to the starting point of our framework. However, they still relied on the virtualization using docker and network emulation using ns-3 and focused on a different scenario, namely satellite links.

Tyunyayev et al. [28] combined picoquic with the Data Plane Development Kit (DPDK) to bypass the kernel. They compared their implementation to other QUIC stacks and increased the throughput by a factor of three. They argue that the speedup is primarily due to reduced I/O but do not investigate other factors in more detail.

Similar to this work, König et al. [44] compared the performance of different QUIC implementations. They found that QUIC cannot reach the performance of TCP and that drastic differences between implementations are visible. They showed a high CPU usage for all implementations and imply performance impacts due to CPU scheduling. In comparison, our work provides further insights into these effects and a detailed evaluation based on the CPU architecture. Furthermore, they focused on dedicated traffic generators based on selected libraries but did not use HTTP/3 and did not evaluate different client and server pairs.

6. Conclusion

In this work, we analyzed the performance of different QUIC implementations on high-rate links and shed light on various influence factors. We systematically created a measurement framework based on the idea of the QUIC Interop Runner. It allows automating QUIC goodput measurements between two dedicated servers. It can use different QUIC implementations, automate the server configuration, collect various statistics, *e.g.*, from the network device and CPU statistics, and provide means to collect, transform, and evaluate results.

We applied the presented framework to evaluate the goodput of mainly LSQUIC, quiche, ans MsQuic on 10 Gbit/s links and analyzed what limits the performance. A key finding in this work is that the UDP receive buffer is too small by default, which leads to packets getting dropped on the receiver side. This results in retransmissions and a reduced goodput. We show that increasing the buffer by at least an order of magnitude is necessary to reduce buffer limits in high link rate scenarios. We observed several differences in the behavior of LSQUIC and quiche, such as differing default parameters, *e.g.*, the UDP packet size or a diverse approach regarding the acknowledgment sending rate. When comparing different TLS 1.3 ciphers, QUIC almost reaches the same goodput with ChaCha20 as with the hardware-accelerated AES ciphers, which behaves differently with TCP. We could not measure any performance increase with the support of segmentation offloading features of the operating system with LSQUIC and quiche. However, MsQuic supports different offloading techniques as well as batching, leading to a significant performance increase if available and enabled. Finally, we show that evaluating QUIC highly depends on the used CPU and even the used core. By applying various optimizations, we increased the goodput of LSQUIC by more than 25% and achieved up to 5 Gbit/s on Intel CPUs. By pinning the QUIC server and client to a specific core, we could increase the goodput by up to 20% in the measurements with 100 Gbit/s NICs. Another advantage of core pinning for reproducible measurements is the increased stability of the results.

Even though QUIC has many similarities to TCP and the specification took several years, our work shows that many details already analyzed and optimized for TCP are still limiting QUIC. Furthermore, the variety of implementations complicates a universal evaluation and yields further challenges to improve performance in the interplay of libraries, *e.g.*, the drastically reduced goodput using an LSQUIC server and quiche client, as shown in Section 4.3.

To allow for an informed and detailed evaluation of QUIC implementations in the future, we publish the framework code, the analysis scripts, and the results presented in the paper [12]. The measurement framework can be applied to evaluate future improvements in QUIC implementations or operating systems.

CRediT authorship contribution statement

Marcel Kempf: Writing – review & editing, Writing – original draft, Visualization, Resources, Project administration, Methodology. **Benedikt Jaeger:** Writing – review & editing, Writing – original draft, Project administration, Methodology. **Johannes Zirngibl:** Writing – review & editing, Supervision, Validation, Methodology. **Kevin Ploch:** Resources, Visualization . **Georg Carle:** Project administration, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The European Union's Horizon 2020 research and innovation programme funded this work under grant agreements No 101008468 and 101079774. Additionally, we received funding by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. This work is partially funded by Germany Federal Ministry of Education and Research (BMBF) under the projects 6G-life (16KISK001K) and 6G-ANNA (16KISK107) and the German Research Foundation under the project HyperNIC (CA595/13-1). The paper at hand is an extended version of our paper presented at the IFIP Networking 2023 conference [45]. This paper introduces additional QUIC implementations to the ones presented in the conference paper. Newer CPUs and 100 Gbit/s NICs are included in the evaluation. Furthermore, we extended the evaluation by additional scenarios and measurements, especially focusing on the impact of different CPU architectures. Finally, the related work section was extended and updated with recent publications.

References

[1] J. Roskind, Experimenting with QUIC, 2013, URL: https://blog.chromium.org/2013/06/experimenting-with-quic.html, (Accessed 10 February 2023).
[2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, Z. Shi, The QUIC transport protocol: Design and internet-scale deployment, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, Association for Computing Machinery, New York, USA, 2017, pp. 183–196.
[3] J. Iyengar, M. Thomson, QUIC: A UDP-Based Multiplexed and Secure Transport, RFC 9000, RFC Editor, 2021, http://dx.doi.org/10.17487/RFC9000, URL: https://rfc-editor.org/rfc/rfc9000.txt.
[4] T. Pauly, E. Kinnear, D. Schinazi, An Unreliable Datagram Extension to QUIC, RFC 9221, RFC Editor, 2022, http://dx.doi.org/10.17487/RFC9221, URL: https://www.rfc-editor.org/info/rfc9221.
[5] R. Stewart, M. Tüxen, K. Nielsen, Stream Control Transmission Protocol, RFC 9260, RFC Editor, 2022.
[6] IETF QUIC Working Group, Implementations, 2023, URL: https://github.com/quicwg/base-drafts/wiki/Implementations, (Accessed 10 February 2023).
[7] M. Seemann, J. Iyengar, Automating QUIC interoperability testing, in: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, EPIQ '20, Association for Computing Machinery, 2020, pp. 8–13.
[8] M. Seemann, J. Iyengar, QUIC interop runner, 2020, URL: https://interop.seemann.io/, (Accessed 10 February 2023).
[9] M. Thomson, S. Turner, Using TLS to Secure QUIC, RFC 9001, RFC Editor, 2021, http://dx.doi.org/10.17487/RFC9001, URL: https://rfc-editor.org/rfc/rfc9001.txt.
[10] The kernel development community, Segmentation offloads, 2023, URL: https://docs.kernel.org/networking/segmentation-offloads.html, (Accessed 10 February 2023).
[11] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, G. Antichi, Making QUIC quicker with NIC offload, in: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, EPIQ '20, 2020, pp. 21–27, URL: https://doi.org/10.1145/3405796.3405827.
[12] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, G. Carle, Code and data publication, 2023, URL: https://github.com/tumi8/quic-10g-paper.
[13] S. Gallenmüller, D. Scholz, H. Stubbe, G. Carle, The pos framework: A methodology and toolchain for reproducible network experiments, in: The 17th International Conference on Emerging Networking EXperiments and Technologies, CoNEXT '21, Munich, Germany (Virtual Event), 2021, http://dx.doi.org/10.1145/3485983.3494841.
[14] R. Marx, L. Niccolini, M. Seemann, L. Pardue, Main Logging Schema for Qlog, Internet-Draft, Internet Engineering Task Force, 2023, URL: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/07/.
[15] aiortc, aioquic, 2023, URL: https://github.com/aiortc/aioquic, (Accessed 26 October 2023).
[16] L. Clemente, M. Seemann, quic-go, 2023, URL: https://github.com/quic-go/quic-go, (Accessed 26 October 2023).
[17] nginx, nginx, 2023, URL: https://hg.nginx.org/nginx, (Accessed 26 October 2023).
[18] Microsoft, msquic, 2023, URL: https://github.com/microsoft/msquic, (Accessed 26 October 2023).
[19] Facebook, mvfst, 2023, URL: https://github.com/facebook/mvfst, (Accessed 26 October 2023).
[20] Private Octopus, picoquic, 2023, URL: https://github.com/private-octopus/picoquic, (Accessed 26 October 2023).
[21] Cloudflare, quiche, 2023, URL: https://github.com/cloudflare/quiche, (Accessed 26 October 2023).
[22] LiteSpeed Tech, lsquic, 2023, URL: https://github.com/litespeedtech/lsquic, (Accessed 26 October 2023).
[23] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, G. Carle, It's over 9000: Analyzing early QUIC deployments with the standardization on the horizon, in: Proc. ACM Int. Measurement Conference, IMC, 2021.

[24] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, G. Carle, Towards a deeper understanding of TCP BBR congestion control, in: 2018 IFIP Networking Conference (IFIP Networking) and Workshops, IEEE, 2018, pp. 1–9.

[25] R. Marx, J. Herbots, W. Lamotte, P. Quax, Same standards, different decisions: A study of QUIC and HTTP/3 implementation diversity, in: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, EPIQ '20, Association for Computing Machinery, New York, USA, 2020, pp. 14–20.

[26] E. Rescorla, The Transport Layer Security (TLS) Protocol Version 1.3, RFC 8446, RFC Editor, 2018, http://dx.doi.org/10.17487/RFC8446, URL: https://www.rfc-editor.org/info/rfc8446.

[27] S. Gueron, Intel® advanced encryption standard (AES) new instructions set, 2010, URL: https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf, (Accessed 10 February 2023).

[28] N. Tyunyayev, M. Piraux, O. Bonaventure, T. Barbette, A high-speed QUIC implementation, in: Proceedings of the 3rd International CoNEXT Student Workshop, in: CoNEXT-SW '22, 2022, pp. 20–22.

[29] QUIC IETF mailinglist, 2020, URL: https://mailarchive.ietf.org/arch/msg/quic/SBetxLwCq5I7un2tkzFb7tXhJMU/, (Accessed 10 February 2023).

[30] D. Raumer, F. Wohlfart, D. Scholz, G. Carle, Performance exploration of software-based packet processing systems, in: Proceedings of Leistungs-, Zuverlässigkeits- und Verlässlichkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen, 6. GI/ITG-Workshop MMBnet 2015, Hamburg, Germany, 2015.

[31] D. Schor, Intel Sunny Cove core to deliver a major improvement in single-thread performance, 2019, URL: https://fuse.wikichip.org/news/2371/intel-sunny-cove-core-to-deliver-a-major-improvement-in-single-thread-performance-bigger-improvements-to-follow/, (Accessed 19 April 2024).

[32] C. Karamatas, AMD EPYC 7003 series microarchitecture overview, 2022, URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/white-papers/overview-amd-epyc7003-series-processors-microarchitecture.pdf, (Accessed 29 December 2023).

[33] M. Piraux, Q. De Coninck, O. Bonaventure, Observing the evolution of QUIC implementations, in: Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC, EPIQ '18, Association for Computing Machinery, New York, USA, 2018, pp. 8–14.

[34] J. Rüth, I. Poese, C. Dietzel, O. Hohlfeld, A first look at QUIC in the wild, in: Proc. Passive and Active Measurement, PAM, Springer International Publishing, 2018.

[35] J. Zirngibl, F. Gebauer, P. Sattler, M. Sosnowski, G. Carle, QUIC hunter: Finding QUIC deployments and identifying server libraries across the internet, in: Passive and Active Measurement, Springer Nature Switzerland, 2024, pp. 273–290.

[36] A. Yu, T.A. Benson, Dissecting performance of production QUIC, in: Proceedings of the Web Conference 2021, WWW '21, Association for Computing Machinery, New York, USA, 2021, pp. 1157–1168, http://dx.doi.org/10.1145/3442381.3450103.

[37] C. Sander, I. Kunze, K. Wehrle, Analyzing the influence of resource prioritization on HTTP/3 HOL blocking and performance, in: Proc. Network Traffic Measurement and Analysis Conference, TMA, 2022.

[38] P. Megyesi, Z. Krämer, S. Molnár, How quick is QUIC? in: Proc. IEEE ICC, 2016, pp. 1–6, http://dx.doi.org/10.1109/ICC.2016.7510788.

[39] T. Shreedhar, R. Panda, S. Podanev, V. Bajpai, Evaluating QUIC performance over web, cloud storage, and video workloads, IEEE Trans. Netw. Serv. Manag. 19 (2) (2022) 1366–1381.

[40] K. Wolsing, J. Rüth, K. Wehrle, O. Hohlfeld, A performance perspective on web optimized protocol stacks: TCP+TLS+HTTP/2 vs. QUIC, in: Proceedings of the Applied Networking Research Workshop, ANRW '19, Association for Computing Machinery, New York, USA, 2019, pp. 1–7.

[41] S. Bauer, P. Sattler, J. Zirngibl, C. Schwarzenberg, G. Carle, Evaluating the benefits: Quantifying the effects of TCP options, QUIC, and CDNs on throughput, in: Proceedings of the Applied Networking Research Workshop, 2023, http://dx.doi.org/10.1145/3606464.3606474.

[42] X. Zhang, S. Jin, Y. He, A. Hassan, Z.M. Mao, F. Qian, Z.-L. Zhang, QUIC is not quick enough over fast internet, 2023, http://dx.doi.org/10.48550/arXiv.2310.09423, URL: https://arxiv.org/abs/2310.09423.

[43] S. Endres, J. Deutschmann, K.-S. Hielscher, R. German, Performance of QUIC implementations over geostationary satellite links, 2022, http://dx.doi.org/10.48550/ARXIV.2202.08228, URL: https://arxiv.org/abs/2202.08228.

[44] M. König, O.P. Waldhorst, M. Zitterbart, QUIC(k) enough in the long run? Sustained throughput performance of QUIC implementations, in: 2023 IEEE 48th Conference on Local Computer Networks, LCN, 2023, http://dx.doi.org/10.1109/LCN58197.2023.10223395.

[45] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, G. Carle, QUIC on the highway: Evaluating performance on high-rate links, in: International Federation for Information Processing (IFIP) Networking 2023 Conference, IFIP Networking 2023, Barcelona, Spain, 2023, http://dx.doi.org/10.23919/IFIPNetworking57963.2023.10186365.