POSTER: Better QUIC implementations with Nesquic

Laurin Brandner
ETH Zürich

Kevin Marti
ETH Zürich

Bas Niekel
ETH Zürich

Ayush Mishra
ETH Zürich

Gianni Antichi
Politecnico di Milano

Laurent Vanbever
ETH Zürich

ABSTRACT

QUIC represents a re-design of the transport stack in response to the requirements of modern applications. While it provides a myriad of benefits, studies have shown that its performance penalties in several contexts keep it from being more widely adopted. While these studies provide a good overview of when QUIC can lack performance compared to TCP, they do little to specifically pinpoint *where* in the QUIC stack these slowdowns come from and how they can be fixed. To address this gap, we present `Nesquic`, a testing infrastructure for QUIC stacks. It collects library-internal metrics of each stack component without changing the library's source code. Our preliminary findings show that not only can `Nesquic` help developers pinpoint which components of their QUIC stacks are less performant, but it also gives them actionable insights into fixing these issues.

CCS CONCEPTS

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

KEYWORDS

QUIC, HTTP/3, Transport Protocol, Performance, eBPF

ACM Reference Format:
Laurin Brandner, Kevin Marti, Bas Niekel, Ayush Mishra, Gianni Antichi, and Laurent Vanbever. 2025. POSTER: Better QUIC implementations with Nesquic. In *ACM SIGCOMM 2025 Posters and Demos (SIGCOMM Posters and Demos '25), September 8–11, 2025, Coimbra, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3744969.3748410

1 INTRODUCTION

QUIC is a novel transport protocol designed from the ground up to address many of the archaic shortcomings of TCP. Not only does it support 0-RTT handshakes, multi-streaming, and baked-in encryption – functionalities that align with modern apps' requirements – but it is also implemented over UDP in the user space to enable rapid deployment and continued evolution.

QUIC is currently deployed by 8.5% of websites on the Internet [11], including major players like Google [4], Meta [7], and Cloudflare [2]. These deployments have demonstrated that, in practice, QUIC can improve performance, especially of short flows [14]. However, a performance penalty currently exists for this feature-rich transport protocol. Several studies have shown that QUIC's throughput can degrade for longer flows when compared to TLS over TCP [14, 15]. QUIC is also CPU-hungry, requiring twice as much processing power than TCP+TLS [12, 15]. Ultimately, QUIC needs to be more efficient for a broader range of applications to enjoy all its benefits.

Why is QUIC not quick? A large number of studies have identified cases where QUIC can be less performant [12, 14, 15], but their insights are rarely actionable. For example, they find that additional socket syscalls arising from running in user space make I/O a major source of overhead. However, none provide library-specific insights that help to improve performance within the given limitations imposed by the kernel. Instead, they propose architectural changes to the kernel and hardware that eliminate these limitations in the first place. For instance, [15] advocates for improved UDP GSO/GRO support and flexible pacing configurations for UDP packets. [13] proposes to offload cryptographic data and header protection to the NIC, only leaving control plane operations to the CPU.

Can we do better? The question arises - is this performance degradation a natural trade-off that the QUIC protocol needs to make in order to enjoy the flexibility of operating from user space – or can we do better? The many implementations of QUIC and their varying levels of performance indicate that we indeed can. QUIC stacks have been shown to make diverse design choices in flow control, multiplexing schedulers, or packetization strategies [5]. This shows that the QUIC standard is flexible enough to allow developers to make different trade-offs and, in the process, potentially leave some performance on the table.

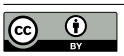
This work[1] aims to reveal these performance deficits more systematically while pinpointing these slowdowns to specific design choices of a QUIC stack's implementation.

2 DESIGN

Testing and benchmarking QUIC comes with several challenges. Most notably, QUIC stacks come in different shapes and sizes, with some of them [8, 9] being implemented as stand-alone transport stacks, while others are implemented as fully integrated HTTP/3 stacks [1, 10]. However, even within this dichotomy, differences exist. QUIC network stacks are composed of multiple components, like congestion control, cryptography, I/O, etc. Even though all components are needed for a functional network stack, approximately half of all open-source libraries on GitHub implement only a subset. Previous root cause analyses [13] largely ignore this and examine the libraries without fully optimized I/O.

Most measurement studies are also limited to measuring coarse metrics like throughput or delay because of this vast diversity. These metrics are implementation-agnostic, making them much more straightforward to measure using traditional packet trace-based tools. However, they often miss the complete picture. By their

[1]The source code is available at https://github.com/nsg-ethz/nesquic



This work is licensed under a Creative Commons Attribution 4.0 International License.
SIGCOMM Posters and Demos '25, September 8–11, 2025, Coimbra, Portugal
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2026-0/25/09.
https://doi.org/10.1145/3744969.3748410

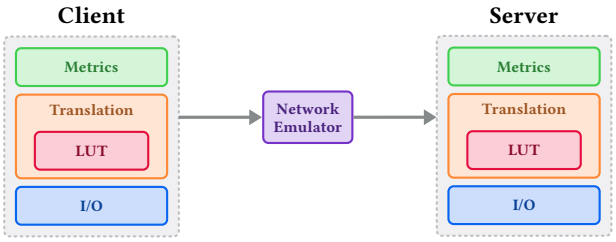


Figure 1: `Nesquic` can supplement missing components for a QUIC implementation before testing it.

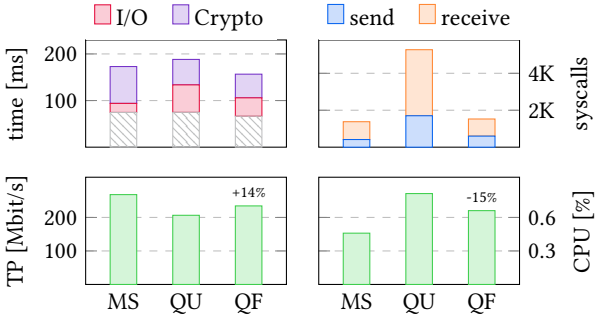


Figure 2: Fixing quinn (QF) reduces CPU utilization and improves throughput.

nature, they only measure the aggregate performance of all the components of a QUIC stack and are, therefore, not very useful in pinpointing the source of slowdowns. Our tool, `Nesquic`, addresses these challenges in the following ways:

A dedicated I/O Layer. Since half of all stacks lack efficient I/O, `Nesquic` implements its own high-performance I/O layer that can be plugged into stacks that lack one. Reusing this implementation makes comparisons more accurate. However, designing it to be efficient *and* generic proves to be challenging. To address this, `Nesquic` implements two layers that separate reusable optimizations from components specific to each library-under-test (LUT). The I/O layer manages UDP sockets and exposes a generic interface for read and write operations. The translation layer interfaces with the I/O layer and implements additional components, such as state management, that the LUT might also lack and cannot be reused without overhead.

An eBPF-based Metric Layer. The metric layer utilizes eBPF [3] to measure component-wise performance. It attaches uprobes and kprobes to performance-critical functions to benchmark the runtime of cryptography and I/O, respectively. To this end, only the names of the measured functions are needed, leaving the LUT unmodified. The measurements can be exported in a logging format like qlog [6] to complement the internal states of the network stack with detailed runtime metrics. In the future, we plan to measure more components, like packetization and congestion control. Moreover, we intend to enrich the collected metrics with runtime arguments to enable even finer-grained analyses.

Figure 1 provides an overview of how a QUIC implementation can be tested using `Nesquic`.

3 PRELIMINARY EVALUATION

As a proof of concept, we use `Nesquic` to evaluate quinn [9] and msquic [8]. Both libraries are I/O-aware and are implemented in Rust and C++, respectively. We employ a simple test case where the client opens a single connection and downloads a 25 MB file over an unrestricted link. Our measurements show that quinn only achieves 206 Mbps on average, while msquic achieves 30% higher throughput at 268 Mbps. Moreover, quinn's average CPU utilization (81%) is much higher than msquic (46%). We see this performance deficit between the two implementations consistently across 50 trials within 2% of variation.

Digging deeper with Nesquic. We use `Nesquic` to dissect the runtime performance of both libraries and find that quinn packets spend significantly more time in the I/O component compared to msquic. While msquic takes on average 19 ms to receive and send new packets, quinn takes close to 60 ms. Previous work has shown that I/O syscalls can degrade performance [13], motivating us to count the send and receive syscalls invoked throughout the experiment. While quinn performs more than 5277 I/O operations, msquic performs only 1377. This is a clear indication that quinn's I/O component is sub-optimal and can be improved.

Reviewing quinn's I/O implementation reveals that it writes at most 16 KB of data per syscall, while msquic writes up to 64 KB per syscall, the maximum for UDP sockets on Linux. As a result, quinn invokes roughly 4× more syscalls, which leads to higher CPU utilization and I/O overhead due to the excessive context switching. To validate these findings, we modify quinn to write up to 64 KB per syscall as well. This improves quinn's throughput by 14% and reduces CPU usage by 15%. Even with this fix in place, quinn does not achieve the same throughput as msquic. Quinn still performs 45% more send operations than msquic. This is likely due to more optimized packetization in msquic. Additionally, each send operation in quinn is around 50% slower than in msquic. Further analysis with a finer-grained metric layer is needed to understand the root cause of this discrepancy. These results for msquic (MS), quinn (QU), and fixed quinn (QF) are summarized in Figure 2.

Finally, we also measure the overhead of `Nesquic`. A lightweight metric layer is crucial to guarantee the accuracy of the measurements. We find that collecting CPU and I/O metrics has a negligible effect on the LUT. However, measuring the crypto runtime reduces throughput by 37%. CPU and I/O are monitored with lightweight bash scripts and kprobes. Crypto, on the other hand, is monitored with uprobes, which are more expensive due to context switching.

4 CONCLUSION AND FUTURE WORK

We show that `Nesquic` is a rigorous benchmarking tool that can help developers make their QUIC stacks more performant. Future efforts will focus on extending support to all major QUIC libraries and developing a more comprehensive test suite. A key challenge is the overhead of monitoring user space components like cryptography. To alleviate it, we propose to use bpftime [16]. It implements efficient uprobes with a custom eBPF runtime in user space, eliminating the need for additional context switching. We have released `Nesquic` as an open-source project to promote reproducibility and aid the development of more performant QUIC stacks.

REFERENCES

[1] Cloudflare. 2018. quiche. https://github.com/cloudflare/quiche.

[2] Cloudflare. 2025. Cloudflare Radar - Adoption & Usage. https://radar.cloudflare.com/adoption-and-usage.

[3] eBPF community. 2025. eBPF. https://ebpf.io.

[4] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, 183–196. https://doi.org/10.1145/3098822.3098842

[5] Robin Marx, Joris Herbots, Wim Lamotte, and Peter Quax. 2020. Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (SIGCOMM '20)*. ACM. https://doi.org/10.1145/3405796.3405828

[6] Robin Marx, Wim Lamotte, Jonas Reynders, Kevin Pittevils, and Peter Quax. 2018. Towards QUIC debuggability. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (CoNEXT '18)*. ACM, 1–7. https://doi.org/10.1145/3284850.3284851

[7] Matt Joras and Yang Chi. 2020. How Facebook is bringing QUIC to billions. https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/.

[8] Microsoft. 2019. msquic. https://github.com/microsoft/msquic.

[9] Dirkjan Ochtman and Benjamin Saunders. 2018. quinn. https://github.com/quinn-rs/quinn.

[10] quic-go authors & Google. 2017. quicgo. https://github.com/quic-go/quic-go.

[11] Robin Marx. 2025. Usage statistics of QUIC for websites. https://w3techs.com/technologies/details/ce-quic.

[12] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2022. Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. *IEEE Transactions on Network and Service Management* 19, 2 (June 2022), 1366–1381. https://doi.org/10.1109/tnsm.2021.3134562

[13] Xiangrui Yang, Lars Eggert, Jörg Ott, Steve Uhlig, Zhigang Sun, and Gianni Antichi. 2020. Making QUIC Quicker With NIC Offload. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC (SIGCOMM '20)*. ACM. https://doi.org/10.1145/3405796.3405827

[14] Alexander Yu and Theophilus A. Benson. 2021. Dissecting Performance of Production QUIC. In *Proceedings of the Web Conference 2021 (WWW '21)*. ACM, 1157–1168. https://doi.org/10.1145/3442381.3450103

[15] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. 2023. QUIC is not Quick Enough over Fast Internet. (2023). https://doi.org/10.48550/ARXIV.2310.09423

[16] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, XiaoZheng Lai, and Andrew Quinn. 2023. bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions. arXiv:2311.07923 [cs.OS]