Poster: Enhancing the performance of a single connection using Multipath QUIC

Vany V. Ingenzi*
vany.ingenzi@uclouvain.be
UCLouvain & WELRI
Louvain-La-Neuve, Belgium

Tom Barbette
tom.barbette@uclouvain.be
UCLouvain ICTEAM
Louvain-La-Neuve, Belgium

Olivier Bonaventure*
olivier.bonaventure@uclouvain.be
UCLouvain & WELRI
Louvain-La-Neuve, Belgium

***Abstract*—The QUIC protocol, designed to reduce latency and improve internet security, faces goodput performance challenges in high-speed networks, particularly with single-thread implementations. This poster extends an existing userspace Multipath QUIC (MPQUIC) implementation to enhance the goodput of a single connection by pinning different network path logics to different cores. Our solution, *mcMPQUIC*, achieves a goodput of up to 20 Gbps with ten paths/cores, surpassing the baseline MPQUIC performance by more than five times.**

***Index Terms*—QUIC, Multipath QUIC, Goodput performance**

I. Introduction

Despite widespread deployment of QUIC [1] since its standardization, most implementations are single-threaded and poorly scale with multicore hosts [2]–[4]. MPQUIC [5] is an extension of QUIC that allows a single connection to use multiple network paths simultaneously. A network path is defined by a 4-tuple, which includes for both the client and server their addresses and ports.

Multithreaded QUIC implementations exist, such as `MsQuic` [6], and `Quinn` [7]. These implementations mainly focus on parallelizing the packet I/O. Parallelizing the QUIC state machine for a single connection is complicated due to the large number of dependent control states and how often they change. Previous works on benchmarking QUIC implementations [2]–[4] show that `piqoquic-dpdk` [4], `MsQuic` [6], and `Quinn` [7] achieve a maximum goodput of 15 Gbps, 10 Gbps, and 8.22 Gbps, respectively.

This poster proposes an MPQUIC [5] based architecture aiming to improve the parallelization of a single QUIC connection. Our solution, *mcMPQUIC*[1], employs MPQUIC to establish multiple network paths all within a single connection. The number of paths used for a given connection scales in accordance with the number of cores on the host. This is because we assign the control loop logic for each network path to a distinct core. A stream is attached to a network path, and the data of that stream can only be forwarded on the particular path. *mcMPQUIC* is a multithreaded extension of Cloudflare's *Quiche* [8] and its multipath extension [9], initially single-threaded. Additionally, we provide an evaluation framework[2] which extends the QUIC-interop runner [2], enabling the evaluation of MPQUIC implementations and the reproduction of our results.

*This publication is supported by the Walloon Region as part of the FRFS-WEL-T strategic axis.

[1]https://github.com/vanyingenzi/quiche/tree/decoupled_multicore

[2]https://github.com/vanyingenzi/master_thesis_utilities

979-8-3503-5171-2/24/$31.00 ©2024 IEEE

II. Architecture

The high-level goal of *mcMPQUIC* is to make the application data transfer as concurrent as possible and use multiple cores to scale the transfer. As depicted in Figure 1, *mcMPQUIC* architecture assigns streams to network paths, then creates a *path thread* that handles the logic of a network path and its assigned streams. We pin each *path thread* on one CPU core and scale *path threads* depending on the number of CPU cores on the host. In this poster, we leave it up to the application to distribute data streams on different paths. Our architecture was inspired by Netchannel [10]. However, while Netchannel aggregates multiple TCP connections, it introduces overhead and requires kernel modifications.

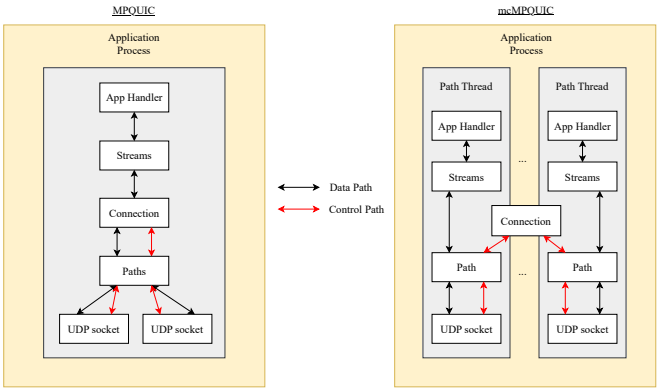

Fig. 1. The *mcMPQUIC* architecture compared to the single-threaded MPQUIC architecture. The data path represents the path of the application data across the architecture and the control path is the path for control information of the transport protocol (i.e control frames).

A. Synchronization of Path Threads

Although *mcMPQUIC* enables independent data transfer across network paths, synchronization is still required for control frames (e.g., connection close, new connection ID). This synchronization is managed through event queues associated with each network path. When a network path receives a control frame, it signals other local paths by placing an event in their event queues. These event queues are stored within the Connection data structure (Figure 1). We initially evaluated an architecture where multiple path threads shared a data

structure containing control information. However, this design performed poorly due to the overhead of mutual exclusion, which prevented efficient scaling. Our analysis showed that with two path threads, the threads spent, on average, twice as much time waiting for access to the shared data structure as they did receiving packets via system calls.

B. Modified MP(QUIC) Semantics

1) Streams: Even though data can be sent on multiple streams concurrently, streams need to synchronize connection-wide to not overwhelm the resources at the receiving hosts. This mechanism is called *flow control*, and QUIC [1] defines two operational points for flow control: connection-level flow control and stream-level flow control. Our design disrupts with the RFC, by doing a path-level flow control instead of connection-level flow control to maximize parallelism. Path-level flow control means the `MAX_DATA` frames will handle the offset and manage resources on a single path. Therefore, this minimizes the synchronization needed during data transfer.

2) Acknowledgments: MPQUIC [5] allows the application to send acknowledgments on another path than the one it is acknowledging the packets from. Cross-path acknowledgments require synchronization at each transmission of an acknowledgment in order to check if a packet has been acknowledged on another network path. Hence, an *mcMPQUIC* endpoint does not send acknowledgments on another path than the one we are sending data on to eliminate this associated overhead.

III. Evaluation

For the experiments, we solely use machines from Cloudlab [11] for ease of *repeatability* [12]. Two 25GbE links connect the client and the server. The client and server are AMD EPYC Rome machines (Cloudlab reference: c6525-25g) running Ubuntu 22.04.2 LTS as the operating system with Linux kernel 5.15. The hosts are connected through two different networks using the dual port technology of the NIC. In order to test the scalability of our solution with a higher number of paths, we equally distribute the number of paths across the two networks. For each number of path/core, we repeat the experiment 10 times, each lasting 20 seconds. The payload is generated by memory-to-memory transfer to remove any impact of Disk I/O.

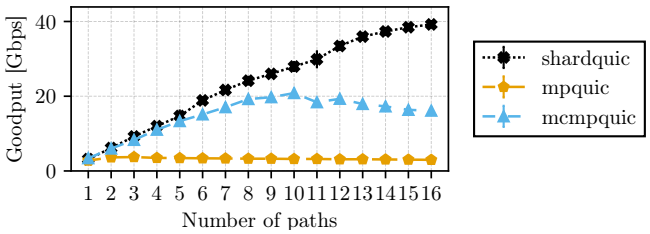

Fig. 2. Goodput as we increase the number of paths.

A. Scalability

From Figure 2, we observe that single-threaded MPQUIC struggles to leverage bandwidth aggregation effectively, flattening at a goodput of 3.71 Gbps as the number of paths increases. In contrast, *mcMPQUIC* demonstrates a remarkable improvement, achieving a maximum average goodput of 20.83 Gbps; far surpassing the 3.71 Gbps of the baseline MPQUIC. This substantial improvement is due to *mcMPQUIC*'s ability to distribute data transfer across multiple CPU cores, unlike the single-threaded MPQUIC. As shown, *mcMPQUIC* scales up to eight paths linearly, reaching a knee capacity of 19 Gbps and achieves a usable goodput with ten paths. However, performance relatively declines as we go beyond ten network paths.

To have an upper bound to compare *mcMPQUIC* to, we use parallel independent connections that share no state among each other, we call it *shardQUIC*. As seen on Figure 2 the widening performance gap between *shardQUIC* and *mcMPQUIC* is due to the synchronization overhead across network paths. Unlike *shardQUIC*, which uses independent QUIC connections, *mcMPQUIC* uses a single connection which requires synchronization across its paths. Sharding can be a viable solution in some use cases (e.g., file transfer), but when data is processed on-the-fly, sharded connections are not as transparent to the application as a single connection.

B. mcMPQUIC Bottleneck

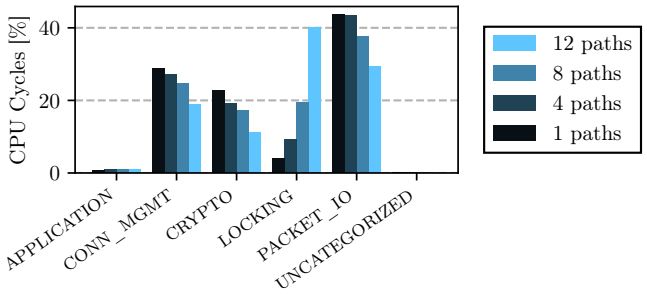

Fig. 3. CPU profile for an *mcMPQUIC* client as the number of paths increase.

To understand the performance limit as paths increase (Figure 2), we conducted a `perf` profile analysis, shown in Figure 3. Examining the *mcMPQUIC* profile in Figure 3, we observe that *Packet I/O* consumes the majority of CPU cycles on the client side, up to eight paths, which is consistent with Kempf et al. [2]. Given that the path threads control loop is nearly symmetric, we expect the later profile to remain unchanged as the number of paths increases for *mcMPQUIC*. However, *Locking* tasks take more CPU cycles at the client when we increase the number of paths/cores. This increase is due to locking on data structures to check for event queues, as stated in Section II.

IV. Future Work

Our solution transfers data five times faster than the baseline MPQUIC. It is ideal for goodput-sensitive applications with many concurrent data flows. Despite better scalability than the baseline, a synchronization bottleneck emerges as the number of paths increases. Furthermore, per-path flow control limits the ability to distribute a single flow across multiple streams on different paths. Additionally, the implementation requires the application to be multithreaded, adding complexity. Future efforts will address these challenges and optimize synchronization overhead through improved data structures. Lastly, as MPQUIC undergoes IETF standardization, we will ensure compatibility with future draft versions.

REFERENCES

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[2] M. Kempf, B. Jaeger, J. Zirngibl, K. Ploch, and G. Carle, "Quic on the fast lane: Extending performance evaluations on high-rate links," *Computer Communications*, vol. 223, pp. 90–100, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014036642400166X

[3] M. König, O. P. Waldhorst, and 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)*. IEEE, 2023, pp. 1–4.

[4] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette, "A high-speed quic implementation," in *Proceedings of the 3rd International CoNEXT Student Workshop*, 2022, pp. 20–22.

[5] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Multipath Extension for QUIC," Internet Engineering Task Force, Internet-Draft draft-ietf-quic-multipath-06, Oct. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/06/

[6] M. Bünstorf and B. Jaeger, "Msquic–a high-speed quic implementation," *Innovative Internet Technologies and Mobile Communications (IITM)*, 2023.

[7] "Quinn," https://github.com/quinn-rs/quinn, 2024. [Online]. Available: https://github.com/quinn-rs/quinn

[8] Cloudflare, "Quiche," https://github.com/cloudflare/quiche, 2024. [Online]. Available: https://github.com/cloudflare/quiche

[9] Q. D. Cloudflare, "Quiche," https://github.com/qdeconinck/quiche, 2024. [Online]. Available: https://github.com/qdeconinck/quiche

[10] Q. Cai, M. Vuppalapati, J. Hwang, C. Kozyrakis, and R. Agarwal, "Towards μs tail latency and terabit ethernet: disaggregating the host network stack," in *Proceedings of the ACM SIGCOMM 2022 Conference*, 2022, pp. 767–779.

[11] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Jul. 2019, pp. 1–14. [Online]. Available: https://www.flux.utah.edu/paper/duplyakin-atc19

[12] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. Shasha, "The repeatability experiment of sigmod 2008," *ACM SIGMOD Record*, vol. 37, no. 1, pp. 39–45, 2008.