UCLouvain

École polytechnique de Louvain

Enhancing the performance of a single connection using Multi-Path QUIC

Author: **Vany Valentin Ingenzi**
Supervisors: **Tom Barbette, Olivier Bonaventure**
Readers: **Aurélien Buchet, Nikita Tyunyayev**
Academic year 2023–2024
Master [120] in Computer Science

Contents

Acknowledgements

First and foremost, I'd like to express my profound gratitude to my family for the opportunities they have given me throughout my life. I would not be writing this thesis without their unwavering support and the sacrifices they made to ensure I received a good education. Their love and belief in me have been my constant motivation.

Secondly, I'd like to acknowledge my supervisors, Prof. Barbette and Prof. Bonaventure, for proposing this thesis topic and for the time they dedicated week after week to guide me. Additionally, I want to express my sincere gratitude to their assistants Maxime Piraux, and Nikita Tunyayev, for their valuable insights and support during our weekly meetings. I would also like to thank Aurélien Buchet for agreeing to join the Jury.

I want to thank my friends who have continuously encouraged me to pursue my academic aspirations and fostered my passion for applied research.

Finally, I'd like to express my deep gratitude to the open-source community for their significant contribution to this thesis. The tools they have provided have been instrumental in my research. I would also like to thank the maintainers of CloudLab for their work, as all the experiments were conducted on their servers.

Grammarly[1] and *ChatGPT*[2] were used to assist in the writing process. These tools were not used to generate the content itself but rather to help me better express my ideas and ensure the clarity of my sentences.

[1] `https://www.grammarly.com/`

[2] `https://www.openai.com/chatgpt`

Abstract

The QUIC protocol, designed to reduce latency and improve internet security, faces performance challenges in high-speed networks, particularly with single-core implementations. This thesis enhances the performance of a single connection using Multipath QUIC (MPQUIC). We design, implement, and evaluate *mcMPQUIC*, a multicore MPQUIC implementation based on the Quiche library. Our implementation aims to leverage multicore hosts by pinning a path to a core. As a result, it achieves a throughput of up to 21 Gbps with ten paths, surpassing the baseline MPQUIC performance by more than five times. In this thesis, we first profile an existing MPQUIC implementation to identify bottlenecks and multiplexing opportunities. We then implement a multithreaded version and evaluate its scalability, demonstrating a 72% throughput increase with two paths and up to 5x with ten paths compared with the baseline. Lastly, we propose a simple framework for reproducing our results and comparing them to other MPQUIC implementations.

Chapter 1

Introduction

QUIC [1] is becoming more deployed since the recently standardized protocol aims to reduce latency and improve internet security and confidentiality. However, QUIC implementations reside in userspace to cope with the rapid changes and deployment of the protocol. For an I/O intensive application running in userspace, the process endures huge stalls due to the packet I/O system calls. This effect is even more highlighted on highspeed networks where QUIC implementations struggle to achieve a goodput of more than 8Gbps with a single core [2–6].

In addition, rapid growth in the high bandwidth networks has pushed the bottleneck of networking systems from the network to the host [7, 8]. However, the Linux kernel has a significant overhead due to the processing of a packet before being delivered to the application [7]. The primary contributors to Linux stack overhead in packet processing are data copy overhead, cache inefficiencies, and the limited collaboration of the CPU scheduler and network stack. This inefficiency has led to recent trends in highspeed networking of bypassing the kernel to handle traffic [8, 9]. These kernel-bypassing network stacks primarily improve performance by being better optimized for multicore systems than the kernel's network stack. Additionally, they reduce CPU latency by directly interacting with the network interface card. Kernel bypassing has been a promising way to increase the speed of QUIC connections [3]. However, it increases the complexity of the application. One could also rely on connection sharding, whereas they divide the workload of the applicational level protocol and have concurrent connections to deliver the workloads independently. The downside of the latter solution is that it is not transparent to the application-level protocol. One of the most performing QUIC implementations without Kernel Bypassing is *MsQuic* from Microsoft [5]. *MsQuic* enhances performance by multithreading the QUIC connection, utilizing one thread for Packet I/O and another for QUIC connection management.

A multipath transport protocol is a transport that is capable of transmitting data transparently between endpoints using two different network paths simultaneously (i.e., WiFi and 5G/Ethernet). Multipath QUIC [10] is an ongoing effort to bring multipath capabilities to QUIC. The extended protocol was created to better utilize available resources at the host. The advantage of MPQUIC over QUIC is bandwidth aggregation and reliability in case of link failure.

In this thesis, we aim to leverage multicore systems and multipath capabilities to enhance the performance of a single MPQUIC connection. We start by profiling an existing MPQUIC implementation to understand multiplexing opportunities. Then, we implement the multicore version, *mcMPQUIC*, and evaluate its scalability, demonstrating

up to a 72% increase with two paths and up to a 5x throughput gain with ten paths.

The thesis code used in this work is featured in two repositories. The open-source implementation of *mcMPQUIC* can be found at:

`https://github.com/vanyingenzi/quiche/tree/decoupled_multicore`

The experiment framework used to orchestrate experiments and create the plots can be found at:

`https://github.com/vanyingenzi/master_thesis_utilities`

This thesis is composed of five chapters:

Chapter 2 starts by providing the necessary background about the QUIC and MPQUIC protocols. Then, the chapter discusses QUIC implementations and their performance evaluation on high-speed links, as conducted by the research community. Further exploration includes techniques at the application level to exploit the multihoming capabilities of hosts. Lastly, we compare recently proposed network stacks with the Linux network stack.

Chapter 3 presents our methodology, design, and architecture of mcMPQUIC. We start by profiling the existing MPQUIC implementation, discussing its limitations, and exploring the MPQUIC semantics to parallelize the processing of our solution. The chapter then delves into the details of our multicore implementation.

Chapter 4 presents our experimental framework inspired by existing frameworks. We discuss and provide examples of configuration files for researchers to compare our mcMPQUIC implementation with theirs.

Chapter 5 evaluates mcMPQUIC compared to the baseline MPQUIC implementation. We also apply several kernel-related optimizations to evaluate their impact on our solution.

Chapter 6 concludes this thesis by summarizing the work done. Additionally, the chapter proposes further directions for future work.

Chapter 2

State of the art

This chapter discusses the QUIC protocol semantics, the multipath extension protocol MPQUIC ietf-draft-6 [10]. Then, we briefly talk about the implementations of the QUIC that are related to our work. Additionally, we look into application-level protocols that do connection sharding on multihomed hosts. Finally, briefly look into the architecture design of highspeed multicore network stacks.

2.1 QUIC

QUIC [1] is a secure connection-oriented transport protocol standardized in May 2021 within the IETF. The protocol aims at reducing latency and improving security over the internet. In addition, the protocol offers connection migration and faster handshake compared to the widely deployed alternative TCP.

2.1.1 Stream

A stream in QUIC is a lightweight ordered byte-stream abstraction given to the application. Unlike TCP, where all data is sent as one continuous stream, QUIC allows multiplexing streams to prevent a problem known as *Head-of-line blocking (HoL)*. This problem occurs in TCP as the loss of a packet blocks the transmission of all succeeding data, as the entire connection is processed as a single-ordered byte stream. However, with QUIC, each stream operates independently, so packet loss only affects the data within the concerned streams, allowing other streams to continue transmitting data uninterrupted. By introducing stream multiplexing at the transport level and providing stream abstractions to applications, QUIC effectively eliminates the *Head-of-line blocking* which can hurt application's performance. An example of this problem can be seen in figure 2.1.

Streams are either unidirectional or bidirectional, depicting the direction in which data flows from the initiator's point of view to the peer. Streams are created by either endpoint (i.e., server or client) by simply sending data using their *Stream ID*. A QUIC connection can have multiple streams transmit data concurrently.

Within a connection, streams are identified by a numerical value referred to as the *Stream ID*. The *Stream ID* is a unique 62-bit integer. Using a stream with a *Stream ID* that is out-of-order leads to all the streams with a lower *Stream ID* and the same type being opened. An endpoint uses the *Stream ID* and an *offset* field in `STREAM` frames to place data in order. As a consequence of offering an ordered byte-stream abstraction, an endpoint must buffer any data received out of order up to the flow control limit.

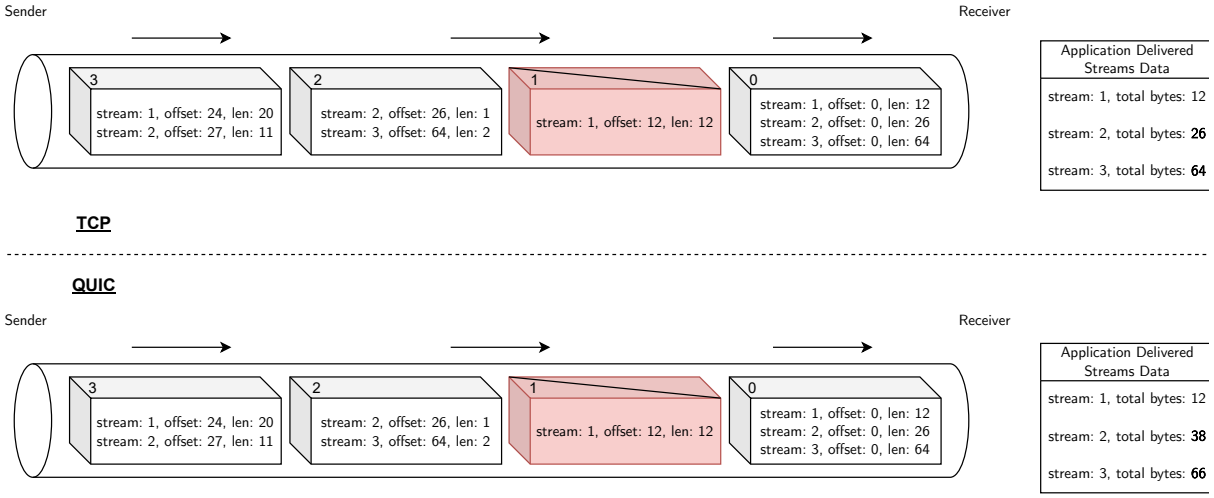


Figure 2.1: *Head of Line* blocking problem and the impact on a QUIC connection compared to a TCP connection. In the figure, the second packet sent is lost.

2.1.2 Flow Control

The data being transferred on a stream is regulated through the *flow control* mechanism. A receiver needs to limit the sender so that the sender cannot overwhelm the resources available to the receiver. Flow control operates at two levels:

- **Stream flow control**. This is done per stream, preventing a single stream from consuming all available resources at the client that must be shared among concurrent streams (i.e., buffers).
- **Connection flow control**. This limits the data sent in `STREAM` frames on all streams, preventing senders from exceeding the total receiving buffer limit for the entire connection.

The mechanism uses a limit-based scheme relying on an *offset*. For stream flow control, the receiver sends a `MAX_STREAM_DATA` frame indicating the *Stream ID* of the concerned stream and an *offset* that represents the maximum number of bytes allowed to be sent on that stream. For connection flow control, the receiver sends a `MAX_DATA` frame to the sender, specifying the total maximum offset for the entire connection, limiting the aggregate data sent across all streams. The initial values of the max data to send per stream and connection-wide are exchanged in the *transport parameters*, which will be discussed later.

2.1.3 Encryption

TLS interaction

QUIC uses TLS (Transport Layer Security), specifically TLS 1.3 [11], for security [12]. To establish connections, QUIC relies on the TLS 1.3 cryptographic handshake. TLS

offers both authentication of the server and ensures the integrity, confidentiality, and authenticity of the data exchanged between endpoints. Unlike TLS on top of TCP, which interacts in a layered architecture, QUIC and TLS cooperate to provide security and performance. The RFC [12] describes two main interactions:

- The reliable stream abstraction offered by QUIC to TLS for TLS's data transmission. QUIC transports TLS handshake data within dedicated `CRYPTO` frames. These frames are similar to `STREAM` frames but are not flow-controlled and do not carry a stream identifier.
- The updates to the QUIC state from TLS regarding the connection security (i.e., encryption level keys, handshake state).

Packet Protection

QUIC protects packets with keys derived from the TLS handshake using an AEAD algorithm negotiated by TLS [12]. TLS uses different encryption levels :

- Initial Keys: Used for `Initial` packets.
- Early data (0-RTT) keys: Used to transmit application data in a *0-RTT resumption*.
- Handshake keys: These are keys used for `Handshake` packet.
- 1-RTT keys: Used for application data once they are derived from the handshake.

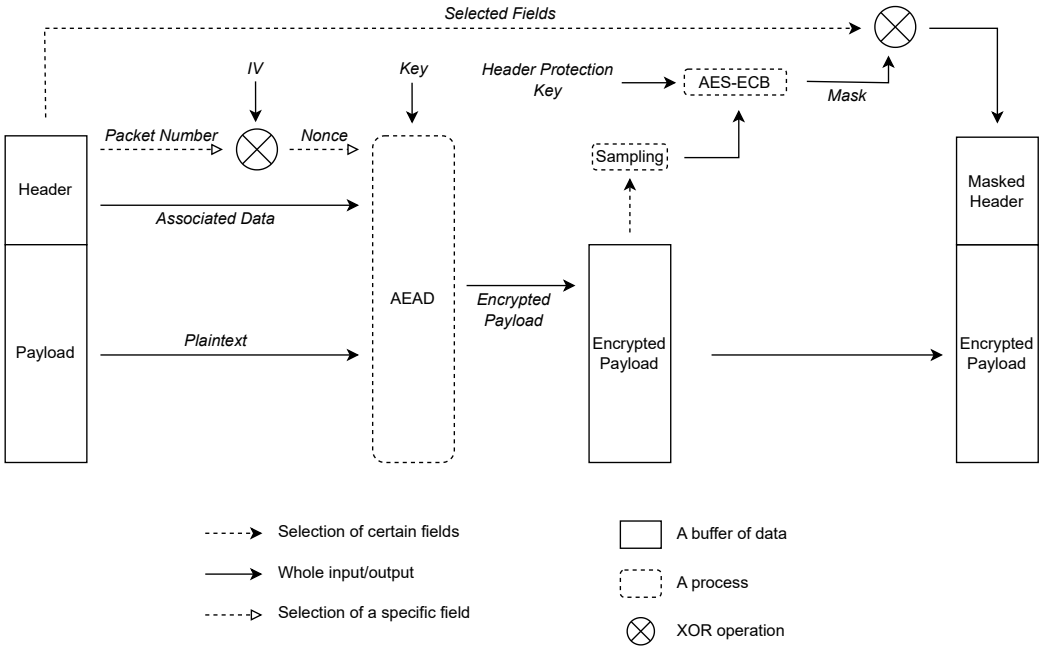


Figure 2.2: Packet protection in QUIC.

Each of these levels is mapped to a QUIC packet number space, and this means that the encryption keys used are different depending on the QUIC's packet type. As a result, certain frames cannot appear in certain packet number spaces. For example, `PADDING`, `PING`, `CRYPTO`, `ACK`, `CONNECTION_CLOSE` can appear in any packet number space, any other frame can only be sent with the application data packet number space.

2.1.4 Congestion Control

Packet Number Space and Packet Numbers

As mentioned in the previous subsection, the packet type infers the encryption level, indicating the packet number space. A *packet number space* is the context in which a packet can be protected, processed and acknowledged. In QUIC, there are three number spaces [12]:

- Initial space, used to handle `Initial` packets, which are protected using the TLS's *initial* encryption level keys.
- Handshake space, used to handle `Handshake` packets, which are protected using keys from TLS's *handshake* encryption level keys.
- Application data space, used to process all the remaining packets except `Version Negotiation` and `Retry` packets. This packet space's packets are protected using TLS's 0-RTT keys if the packet is a `0-RTT` packet. Otherwise, TLS 1-RTT keys are used.

A *packet number* is an integer incremented monotonically starting from 0 to $2^{62} - 1$. Within a packet number space, whenever a packet whose header includes a *packet number* is sent, the *packet number* is incremented by at least one. Within a packet number space, each endpoint has a packet number for sending and receiving packets. Every packet contains the packet number except `VersionNegotiation` and `Retry` packets. The packet number has two major roles in QUIC :

- As a cryptographic nonce to the AEAD encryption algorithm for packet protection as seen in the figure 2.2.
- To indicate the transmission order of packets for congestion control.

In congestion control, the *transmission order*, is the order in which packets are being sent and the *delivery order* is the order in which packets need to be delivered to the receiver. TCP's congestion control mixes transmission order at the sender and with the delivery over as TCP uses only the sequence number. This results in a problem known as *retransmission ambiguity*, which occurs when an acknowledgment arrives for a retransmitted TCP packet, and the receiver cannot tell which transmission is acknowledged. *Retransmission ambiguity* problem occurs, especially when the TCP timestamps extension is not being used. In QUIC, the packet number within a packet number space indicates transmission order. So, for a packet sent before another, the former's packet number will be lower than that of the later packet's packet number. Hence, upon loss of an ack-eliciting packet, the sender will include adequate frames present in the lost packet and then send a new packet with a higher packet number. Upon reception of an acknowledgment, there is clarity on which packet is acknowledged, resulting in more accurate RTT measurements. Finally, delivery order is dictated by the *offset* in the `STREAM` frames as seen in the previous subsection. It is worth noting that the `STREAM` frames boundaries are not required to stay the same upon retransmission. This means that the *offset* and *length* field values can change between the lost and retransmitted packets.

Acknoweldgement

In order to support reliability, a QUIC receiver uses `ACK` frames to confirm the reception of a packet to the sender. In QUIC, there are packets called *ack-eliciting* packets. These packets require that the receiver send at least one `ACK` within `max_ack_delay` transport parameter. Failing to do so may result in the sender performing unnecessary retransmissions. Packets that are not *ack-eliciting* are acknowledged when the receiver has to send an `ACK` for other purposes. For the `VersionNegotiation` and `Retry` packets, as they do not contain packet numbers, the receiver implicitly acknowledges these packets by sending an `Initia l` packet.

Loss Detection and Recovery

As acknowledgments are performed per *packet number space*, loss detection is also done per *packet number space*. If the server sends a packet p_x and then sends a packet p_y. The loss detection considers a packet, p_x as lost if [13]:

- It is in-flight and unacknowledged and was sent before an acknowledged packet, p_y.
- And the packet was sent *kPacketThreshold* packets before an acknowledged packet, p_y or the time interval between the send of p_x and p_y packet is greater than a threshold. This time threshold is based on the `smoothed_rtt` and the latest RTT alongside other factors.

The *kPacketThreshold* is a threshold for the maximum number of packets that can be received after a prior sent packet. This follows TCP's best practices and is required that an endpoint can't set this value below 3. Given that the packet numbers are encrypted in QUIC, middleboxes can cause reordering, leading to spurious retransmission. Therefore, increasing this threshold in networks with more reordering could increase performance.

In addition to the loss detection mechanism, QUIC uses *probe timeout (PTO)* to ensure that acknowledgments are received. By definition, PTO occurs when a sender waits for a packet acknowledgment. This timeout triggers the sender waiting for an `A CK` of an ack-eliciting packet that is not acknowledged yet to send one or two probe datagrams. The reception of these probe packets causes the receiver to respond with acknowledgments. The PTO's primary purpose is to allow recovery from the loss of tail packets and acknowledgments. Therefore, an expiration of PTO timeout does not mean prior unacknowledged packets are lost [13]. However, the reception of new acknowledgments due to the PTO timeout allows the connection timeout to continue functioning effectively.

Congestion Control Mechanism

Unlike loss detection, which operates per packet number space, congestion control and RTT measurements are done across multiple packet number spaces, as the latter are properties of a path. Congestion control is the process of regulating the data flowing between endpoints in order to prevent network congestion or network collapse. In the literature, there have been multiple proposed algorithms, such as CUBIC [14] and TCP NewReno [15]. Congestion control and loss recovery work in hand to get the most of the available bandwidth. However, a faulty loss detection implementation may cause performance degradation as congestion control is reactive upon detecting packet losses. QUIC provides generic signals allowing different server-side congestion control algorithms.

Appendix A.3 and B.1 of the QUIC's congestion control and loss detection RFC [13] defines some of these signals as variables: `latest_rtt`, `smoothed_rtt`, `pto_count`, `largest_acked_packet` per packet number space, `kInitialWindow`, etc.

2.1.5 Handshake

Transport parameters

QUIC's transport parameters are unilaterally authenticated declarations that govern the connection. These values are exchanged during the handshake as *transport parameters* within the TLS's cryptographic handshake. Once the handshake is complete, the endpoints obey the restrictions imposed by the peer's values. Amongst these parameters, some values can later be changed by the sender's new control frames, for example, `initial_max_data` that indicates the initial values for the flow control and can later be modified by a `MAX_DATA` frame.

Connection IDs

In QUIC, *Connection ID* is an identifier of a connection. However, a single connection can have multiple *Connection IDs*. Endpoints independently choose and exchange these IDs for the peer to use when communicating. There are two types of *Connection IDs*: *Source Connection IDs* and *Destination Connection IDs*. The first pair of connection IDs are exchanged within the `Initial` packets, where each endpoint sets the *Source Connection ID* field in the header to the *Destination Connection ID* that the peer's subsequent packets will be using. Once the handshake is established, additional *Connection IDs* can be sent to a peer using a `NEW_CONNECTION_ID` frame. These IDs route incoming packets to the proper connection and are used to derive cryptographic keys. The primary purpose of *Connection IDs* is to prevent packets from being delivered to the incorrect endpoint due to changes that may happen at the lower layers (i.e., UDP, IP).

Types of Handshake

One of the key advantages of QUIC over TCP+TLS1.3 is the quick connection handshake in terms of latency. This is because the TCP+TLS1.3 handshake first does the TCP handshake and then the TLS handshake since the two protocols interact in a layered manner. QUIC, on the other side, integrates the TLS cryptographic handshake and optionally extends it by adding QUIC transport parameters. QUIC has three possible handshake procedures (seen in figure 2.3).

The first one is *1-RTT handshake*, where the client initiates the connection with a QUIC `Initial` packet containing the `ClientHello` data. Once the server receives the client's `Initial` packet, it replies with an `Initial` packet with TLS's *ServerHello*, a `Handshake` packet carrying TLS's *encrypted extensions, certificate, and certificate verify chain* and *FIN*. Subsequently, after receiving the client's `Initial` packet, the server can send the application data with `1-RTT` packets. Since QUIC packets can be coalesced, all the server's packets can be sent into one UDP datagram. At the reception, the client replies with a `Handshake` packet with TLS's *FIN*, and the client can start sending data with the `1-RTT` packets. As described in the above encryption subsection, these packets are protected differently with different keys where applicable.

The second handshake procedure is *0-RTT connection resumption*, where the endpoints already established a complete handshake in the past. This requires the server to have set the TLS's *early_data* extension in a NewSessionTicket within the previous TLS session, alongside a *pre_shared_key*. For this handshake flow, the client sends an `Initial` packet with *ClientHello* and then sends application data within a `0-RTT` packet. Upon verification of the ClientHello, the server replies with an `Initial` packet containing the *ServerHello*, a `Handshake` packet with the TLS's *encrypted extensions* and *FIN*. The server can reply with a `1-RTT` packet containing the application data.

The next handshake procedure is a *stateless reset* where the server replies to the client's `Initial` packet with a `Retry` packet containing a token, and the client responds with an `Initial` packet alongside the received token. Then, the handshake continues as for the *1-RTT* handshake. A server can force a *stateless reset* to check that the client can send or receive on the path and that the endpoint cannot be used for a traffic amplification attack.

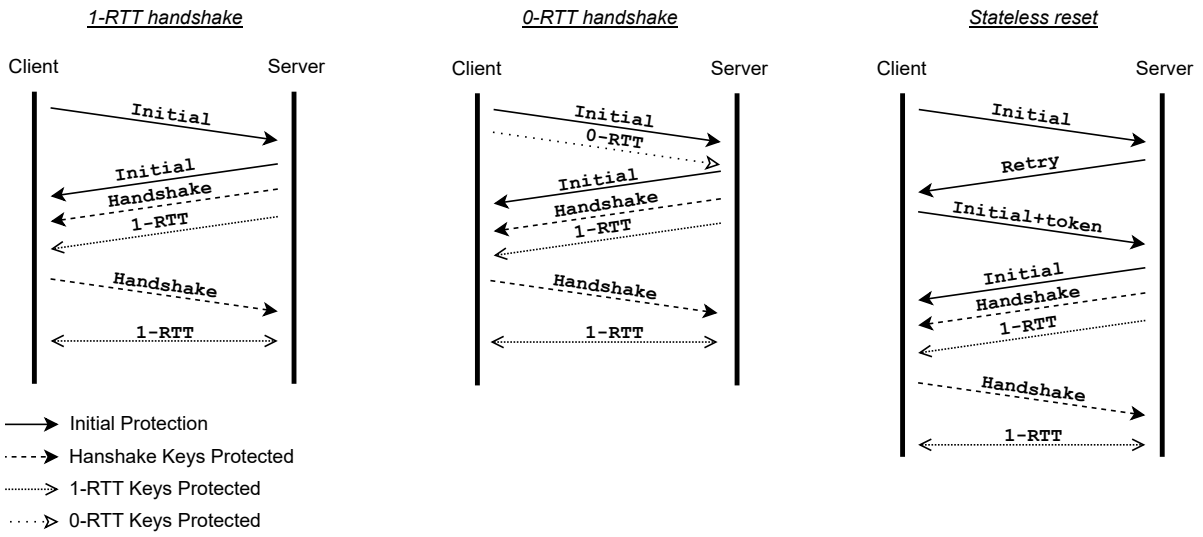


Figure 2.3: A simplified figure representing the different handshake flows in QUIC.

2.1.6 Connection Migration and Path Validation

QUIC supports situations where an endpoint can intentionally or unintentionally migrate into a new network, referred to as *Connection Migration*. The usage of Connection IDs enables connection migration. Connection migration can be caused intentionally by a mobile endpoint or unintentionally by a change of the 2-tuple (address, port) due to a NAT. *Path Validation* is a unilateral test of reachability, meaning that peer A checks whether peer B can receive and respond to their packets. In this case, path validation does not check whether the path towards A is valid. However, the remote endpoint can also perform a path validation to verify. An endpoint sends a `PATH_CHALLENGE` frame containing unpredictable data to the peer to initiate a path validation. Upon reception, the peer responds to the `PATH_CHALLENGE` frame with a `PATH_RESPONSE` containing the data in the former frame. The RFC requires endpoints not to delay `PATH_RESPONSE` unless

restricted by the congestion control and also needs to send the response on the same path as they received the `PATH_CHALLENGE`.

2.1.7 Packet Formats

QUIC packets consist of one header and one or more QUIC frames, except for `Version Negotiation` and `Retry` packets, which do not contain frames. There are two types of headers :

- *Long headers*: These are used for packets sent during the handshake establishment and contain more information than short headers.
- *Short headers*: These are used for 1-RTT packets after connection establishment is completed.

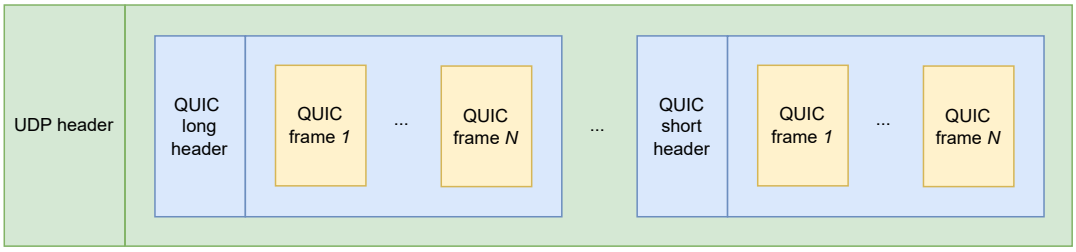


Figure 2.4: An example of two QUIC packets within a UDP datagram. This is known as packet coalescing.

QUIC allows packet coalescing. This means putting multiple separate and complete packets within a single UDP datagram. Coalescing packets is done to construct *Path Maximum Transmission Unit (PTMU)* packets. The receiver can perform connection routing based only on the Connection IDs in the first packet. Hence, a sender can only coalesce packets with the same *Destination Connection IDs*. In addition, packets with no length field in the header cannot be followed by another packet in the same UDP datagram. Other than in the UDP datagram, certain frames cannot be put in the same QUIC packet depending on the connection state and the packet type.

2.2 MPQUIC

Multipath QUIC [10] is a Multipath extension of QUIC that takes advantage of the heterogeneous network capabilities of end devices such as WiFi and 5G. The main motivations of the protocol are to utilize the pool of resources available to transport data for a single connection and increase resiliency to connectivity failures [16]. In this section, we discuss some changes brought by MPQUIC since it tries to use most of QUIC's mechanisms. Figure 2.5 compares MPQUIC and QUIC network layers.

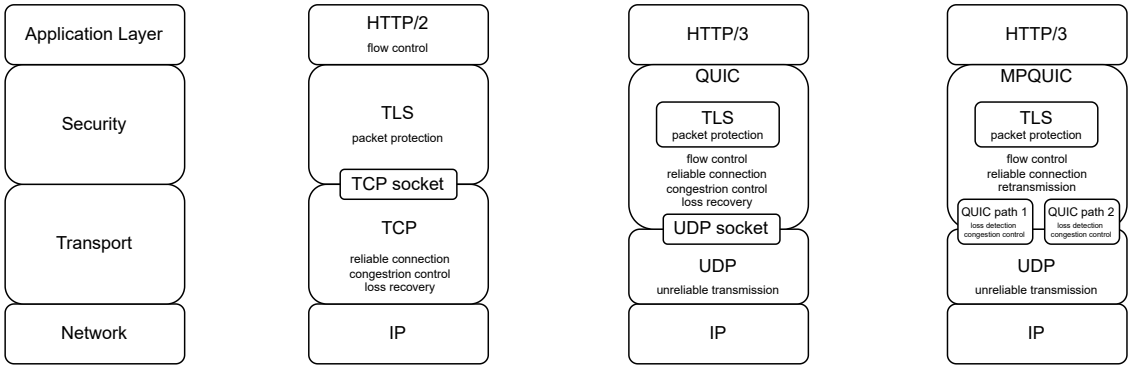


Figure 2.5: A figure showing this study's stack of discussed protocols.

2.2.1 Packet Number Spaces and Connection IDs

MPQUIC uses different packet number spaces per path, allowing loss recovery and congestion control per path. As a result, the extended protocol has to map incoming packets to the right path. In QUIC, we use Connection IDs to map incoming packets to a connection, ensuring that changes in the lower layer (UDP, IP) do not affect the routing of the packets to the proper connection. In MPQUIC, we also use Connection IDs to map the packets to the proper packet number space. As a consequence, the extended protocol uses different Connection IDs per path.

2.2.2 Encryption

In QUIC, the packet number is used as a nonce in the AEAD encryption algorithm, as seen in figure 2.2. Since we use different packet number spaces per path, this slightly changes how we construct the nonce we give to the packet protection algorithm for the 1-RTT packets. To meet the nonce's uniqueness constraint, the nonce in MPQUIC becomes a combination of packet protection IV, packet number, and the last 32 bits of the *Destination Connection ID*. Due to the specific construction of this new nonce, endpoints are mandated to execute a key update after every 2^{32} Connection ID usage.

2.2.3 Congestion Control

Acknoweldgement

An MPQUIC connection starts by using `ACK` frames for acknowledgments. Once MPQUIC has been negotiated and the handshake completed, the protocol uses a new control frame `ACK_MP` to replace the QUIC's `ACK` frame for acknowledgments of 1-RTT packets. This new control frame extends `ACK` by adding a Destination Connection ID field to route the acknowledgment to the proper packet number space. Before the handshake completion, packets must be acknowledged using the `ACK` frames with a Connection ID of sequence number 0. A special note is that `ACK_MP` can be sent on any path.

Congestion Control

The congestion control remains per path as in QUIC. However, two or more paths may share a common bottleneck when using multiple paths. In this scenario, standard

congestion control results in an unfair bandwidth distribution as a multipath connection gets more bandwidth than single-path connections along the same bottleneck. The scientific literature has introduced coupled congestion control schemes such as LIA [17] to mitigate this problem. These congestion control schemes originally thought for Multipath TCP can also be used in MPQUIC, given that they are adapted accordingly.

2.2.4 Packet Scheduling and Retransmissions

Loss detection remains per packet number space. However, because the connection now has multiple paths, it can choose a retransmission strategy between retransmitting lost frames on the same path, retransmitting frames on a different path, or sending lost frames on multiple paths simultaneously. This choice is not only made for retransmission, it is another operational point that MPQUIC brings.

Packet Scheduling is the mechanism of deciding on which path to send the next packet in the presence of multiple paths. The scheduling decision is made locally at an endpoint based on the application's requirements and the MPQUIC implementation. In network stacks, having a shorter control loop results in having a more performant execution. The performance gain is due to quickly updating congestion control or flow control. For example, if flow control is blocked, sending a `MAX_DATA` frame on a path with an RTT of 100ms is better than sending the same frame on a path of 300ms. To achieve this in MPQUIC, one may send `ACK_MP` frames on the smallest RTT path. The lowest-RTT-First [18] is a packet scheduling algorithm that sends packets to the lowest RTT paths first, for which there is an available congestion window.

2.2.5 Handshake and new path establishment

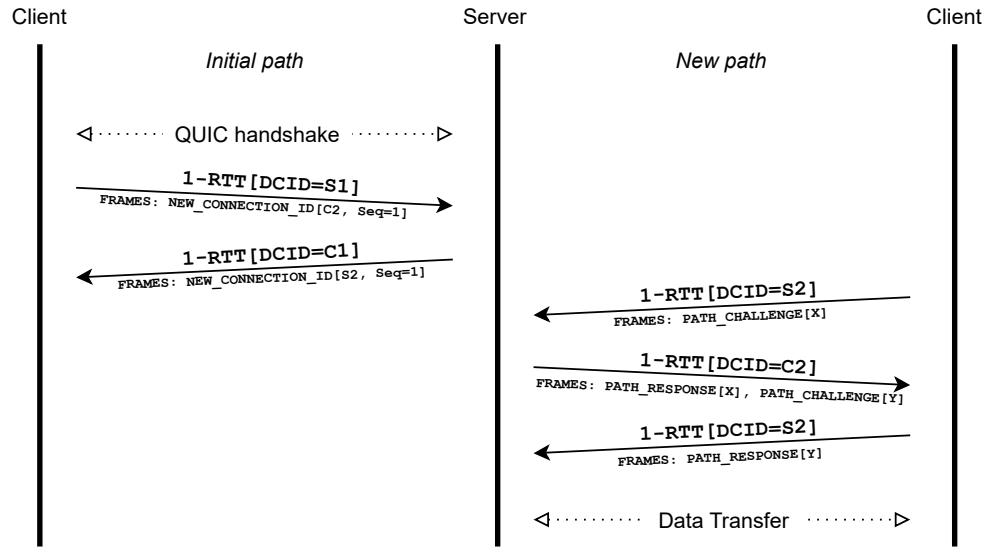


Figure 2.6: MPQUIC new path establishment example.

MPQUIC extends QUIC's Handshake with just a new zero-length value transport parameter `enable_multipath` that is used to negotiate the usage of MPQUIC. The protocol extends the usage of some of QUIC's transport parameters for other usages such as `active_connection_id_limit`, initially limiting the number of useable Connection

IDs. In MPQUIC, `active_connection_id_limit` is also used to limit the number of concurrent paths.

Once the normal QUIC handshake is complete and both peers have enabled multipath, the client can try to establish a new path as seen in figure 2.6. Only the client can establish a new path, not the server. The server can only validate paths initiated by the client. The client must first check if unused Connection IDs are available for both endpoints. Then, it performs a path validation, as seen in the previous section, using the unused Connection Destination ID in a 1-RTT packet. Once the path validation is complete, the client can start sending packets on the path. The server also performs a path validation on the new path if it uses it.

2.3 Implementations

As seen in figure 2.5, QUIC is a transport protocol that uses UDP. QUIC designers chose to do so for mainly two reasons [19]: iterative deployment of implementations without waiting for new kernel releases and for internet-scale deployment, as UDP is a transport protocol supported by middleboxes. A new transport protocol over IP will most likely get blocked by firewalls or any other middleboxes that do not implement it, preventing internet traversal. For example, this is the case for SCTP, a transport protocol standardized in the 2000s that is not widely deployed due to the lack of support from middleboxes [20]. By using UDP, QUIC can be implemented in userspace and rely on the UDP's kernel implementation. This choice allows developers to adopt various approaches across multiple dimensions, such as I/O processing, programming languages, and multicore architecture.

2.3.1 Performance Evaluation

Jaeger et al. [2] have evaluated mainstream implementations of QUIC. Their findings reveal that goodput varies across implementations from 90 to 3500 Mbps on a 10Gbps bandwidth link. The authors highlight the impact of UDP receive buffers and recommend increasing them by at least an order of magnitude. Through profiling, they identify the main bottleneck for QUIC implementations as the packet I/O. Additionally, they observe that the performance can increase by up to 20% by simply running benchmarks on newer-generation CPUs.

Another work by Zhang et al. [4] shows that in a QUIC connection, the bottleneck is the *receiver-side* processing. When comparing it to TCP, they observe that the client issues much more packet reads than TCP. The tested implementation did not use UDP GRO because it is far less deployed than TCP's Segmentation Offload. In addition, the authors highlight the importance of making UDP GSO/GRO QUIC friendly, as these offloading techniques require UDP packets to be of the same length. Nevertheless, this is not always the case for QUIC packets, making them inefficient, especially GRO.

2.3.2 Quiche

Quiche [1] is an open-source implementation of QUIC by Cloudflare. The library implementation is written in Rust programming language. *Rust* is a modern programming language that provides semantics that ease memory-safety and thread-safety without

[1] `https://github.com/cloudflare/quiche`

compromising performance. The architecture design of Quiche provides functions that implement QUIC's logic and lets the application that uses the library dictate the flow of operations of the connection. For example, the library does not provide socket handling. This implementation will be further analyzed in the next chapter since our solution is based on an MPQUIC pull request from the library.

2.3.3 MsQuic

Regarded as one of the best-performing QUIC implementations, MsQuic is a high-speed multithreaded QUIC implementation written in C/C++ [5]. The implementation takes advantage of the multicore architecture to provide scalable performance with the number of cores for the server. The performance gain of MsQuic compared to other implementations is thanks to the fact that it is multithreaded. MsQuic's thread model is made of two types of threads:

- Datapath threads: These threads handle UDP I/O and basic QUIC validation. These threads interact with the underlying UDP sockets in a platform-dependent manner to achieve the highest performance possible.
- Core threads: These threads handle the queues of every connection object, performing most of the connection processing.

In a presentation [21], one of the developers shares the leading best practices:

- Batching: They batch between Application/QUIC and QUIC/UDP in both directions. Batching, in general, allows to amortize the complexity of passing a chunk of data from one layer to another. In order to understand the gain of batching, one can look at improvements brought by using Generic Segmentation Offloading in UDP, where the throughput nearly doubles [22].
- Core Affinity: The implementation keeps all the logic of handling a connection on a single core to mitigate bottlenecks, such as cache-misses and lock contentions, that deteriorate performances [21, 23, 24]. This practice allows to profit Receive Side Scaling [25] on the server efficiently.

2.4 MP-H2 and mHTTP

MP-H2 [26] and mHTTP [27] are client-only modifications of HTTP/2 that take advantage of the multihoming of the client to download different chunks of the same file from a remote server [26] or possibly multiple remote servers [27]. They are very similar as they rely on HTTP/2's byte range requests, a built-in feature in HTTP/2 and HTTP/1.1, to determine which chunks to send on which path. Both implementations achieve performance that is similar to MPTCP without modifying the server.

In their testbed, the links bandwidths are relatively small—5 Mbps and 10 Mbps for [26] and 100 Mbps for [27]. Since the network is the bottleneck in these bandwidth scenarios, the end host can take advantage of an additional path, which is why using multiple paths is beneficial.

2.5 Multicore Network Stacks

As the Internet expands and hardware innovation evolves, data transfer speeds within networks have surged to hundreds of gigabits per second. In earlier times, when transfer speeds were a few hundred megabits per second, regular end hosts with general-purpose operating systems could easily keep pace with line speeds. However, as transfer speeds increase, it becomes increasingly challenging to match these speeds, especially on a single core, without going around the kernel due to the cost of context switches and the latency of system calls. Researchers have noted that the architecture of network I/O in the Linux stack may need to be more optimally suited for these rising bandwidths [8, 9].

2.5.1 multicore TCP

Jeong et al. [9] introduced multicore TCP (*mTCP*), a user-level TCP stack designed for multicore systems. Their approach involved creating an *mTCP* thread for each application thread, aiming to minimize the effort required to adapt existing applications and ensure TCP correctness based on timing constraints. They localize all connection resources on one core to reduce inter-core contention and facilitate Receive Side Scaling (RSS) for flow-level core affinity. They credit their significant performance improvement, which is 25 times better than the Linux stack performance, to several primary factors: batching packet I/O in both directions to reduce context switch overhead, using data structures without locks, placing threads in a way that considers the cache, and managing resources efficiently for each core.

2.5.2 NetChannel: Disaggregated Host Network Stack

Cai et al. [8] present in their work some of the limitations of the Linux stack:

First, Linux's static and dedicated pipelines do not scale with long flows. Their work shows that data-copy operations take most of the CPU cycles in the Linux stack. Even with optimizations, the main limitation of long flows is that the packet processing remains on a single core, which cannot handle hundreds of gigabit traffic.

In addition, the Linux stack does not efficiently handle isolation between latency-sensitive and throughput-sensitive applications. This inefficiency is highlighted when the number of applications exceeds the available cores. In this scenario, the latency-sensitive application can be blocked by the processing of a throughput-sensitive application, increasing tail latency in latency-sensitive applications.

To overcome these challenges, they propose a disaggregated architecture composed of three layers: a virtual network system, a NetDriver, and finally, a NetScheduler.

Firstly, the *Virtual Network System* layer acts as the standard POSIX socket interface with the exact semantics of the regular socket. In addition, this layer maintains a per-core worker thread that copies data between the userspace and the kernel (when applicable). The data copy operations are divided and distributed across data copy workers on all cores. Secondly, using channel abstraction, the *NetDriver* decouples network layer processing from the sockets. This layer allows scaling the number of channels between a pair of endpoints independently of the other applications running on the hosts. Lastly, the *NetScheduler* layer schedules application data to channels, scaling the number of channels and orchestrating data copy requests.

One of the main benefits of NetChannel [8] is that it allows applications to benefit from available resources without having to write the multicore implementations required by the default Linux stack. The researchers show that an application can saturate a 200Gbps link with just a single socket. In addition, their design offers a better isolation of latency or throughput-sensitive applications than the Linux stack.

Conclusion

In a top-to-bottom approach, we have seen in this chapter the following:

QUIC provides stream multiplexing for data transfer, providing an operating point for an application that aims to achieve higher throughput. Concurrent data transfer can be done with multiple concurrent streams, as done in MP-H2 or mHTTP, using multiple range requests or concurrent data flows. However, for maximum concurrency like MP-H2 or mHTTP, the assumption is that data and the underlying layers provide multipath capabilities, and the transfer must be done simultaneously.

MPQUIC is an extension of QUIC that offers multipath capabilities within a single QUIC connection. This means a multihomed client can send data concurrently over multiple paths without establishing additional connections, distinguishing it from MP-H2 and mHTTP.

Lastly, multicore network applications (MsQuic) and multicore network stacks (netchannel, mTCP) achieve higher performance than alternatives. Their architectural design leverages the resources of a multicore host more effectively, resulting in better performance.

Chapter 3

Multicore MPQUIC

In this chapter, we discuss the motivations that led to the design of our solution entitled *mcMPQUIC* (multicore MPQUIC), then detail its architecture. To design our solution, we start by profiling the implementation of Quiche. Once we identify the bottlenecks, we will harness the MPQUIC semantics to parallelize the operations for better scalability.

3.1 MPQUIC in Quiche

The architecture of Quiche allows the application to control the execution flow of the ongoing connection. This is achieved through an API provided by a struct called `Connection` (listing 3.1), detailed in the following sections. This struct implements MPQUIC logic, enabling the application to manage the connection's event loop (e.g., timers, UDP I/O) via its methods. The repository includes examples of applications, such as a client and a server, which we use as the basis for our analysis.

```
/// A QUIC connection.
pub struct Connection {
  ...
  /// Packet number spaces.
  pkt_num_spaces: packet::PktNumSpaceMap,
  /// The path manager.
  paths: path::PathMap,
  /// Streams map, indexed by stream ID.
  pub streams: stream::StreamMap,
  ...
}
```

Listing 3.1: A summary of discussed fields of the `Connection` struct. The full structure can be found in the listing A.1.

3.1.1 Provided Applications

Application loop

The MPQUIC client starts by configuring the connection and sends the initial packet. Once this is done, the execution enters a main loop until the connection is closed. The pseudocode can be found in listing 3.2. On the other hand, the server waits for the incoming packets by listening to the UDP socket and checking if there is an ongoing connection with the Connection IDs used in the packet. If there is none, it attempts to

establish the QUIC handshake if applicable. Once the handshake is complete, it maps the connection IDs to a client connection to handle multiple concurrent clients.

3.1.2 Profiling

```
fn endpoint(config) {
  let mut buffer: [u8] = [...];
  let (conn, sockets) = setup_connection();
  loop {
    let events = poll_sockets(conn, sockets);
    if events.empty(){
      conn.on_timeout();
    } else {
      for event in &event {
        'read: loop {
          let (len, from, to) = sockets.recv_from(&mut buffer);
          conn.recv(&mut buffer[..len], {from, to});
        }
      }
    }
    handle_app_proto(conn);
    let scheduled_path_infos = path_scheduler(conn);
    for (local, peer) in scheduled_path_infos{
      'write: loop {
        let (write, info) = conn.send_on_path(&mut buffer);
        sockets.send_to(&out[..write], info);
      }
    }
  }
}
```

Listing 3.2: MPQUIC endpoints pseudocode.

Looking at the provided pseudocode, we can observe an initial performance issue in how the packets are passed to the connection. In the context of MPQUIC, a host receives packets from multiple sockets. With the provided applications, while an endpoint passes the received packet to the `Connection` struct methods, other packets are pilling up on other sockets. Hence, handling packets from one path impacts the waiting time for packets from other paths if they have already arrived at the host. Under the assumption that the number of packets is constant across paths, increasing the number of paths increases this delay at the host, where packets spend more time before passing to the connection.

Another point of improvement is within the Quiche core library. The Quiche library abstracts most of the internal connection's semantics so that Quiche can be used in other binaries with Rust's *ffi* interface. However, this prevents the application from taking advantage of the parallelism of certain operations. For example, the `Connection` structure has a field entitled `PathMap`, which maintains the state of the paths. As the name hints, the structure maps a path ID to a `Path` structure, which contains information about the concerned path, such as the congestion control mechanism and recovery. This abstraction to the application means that even though MPQUIC's path logic is nearly independent across paths, a multithreaded application cannot take advantage of the multiplexing of paths. The same idea applies to streams, where there is an entire data structure `StreamMap` whose role is to map stream ID to a data structure representing the stream. The `StreamMap` also contains the Transmission/Reception buffers of data being transferred through

the stream before being sent on the wire or to the application. Therefore, a multicore application cannot send data concurrently on streams, as we can only access one stream at a time.

To assess where the endpoints spend most of the execution time to further parallelize those sections, we use `perf`[1]. We run the endpoints using loopback addresses on Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz. The figures 3.1 and 3.2 show CPU cycles-based profiles for the client and server, respectively. The profiles were captured with `perf` using a single path with a loopback address over a 10-second capture period, utilizing the *last branch records (lbr)* callgraph. The figures depict trees with nodes representing methods and the number of CPU cycles consumed by each method and its children. Packet I/O uses 37.92% of the CPU cycles, while `Connection` related tasks accounts for up to 45.59%. Within `Connection` related tasks, packet protection accounts for 16% of total CPU cycles.

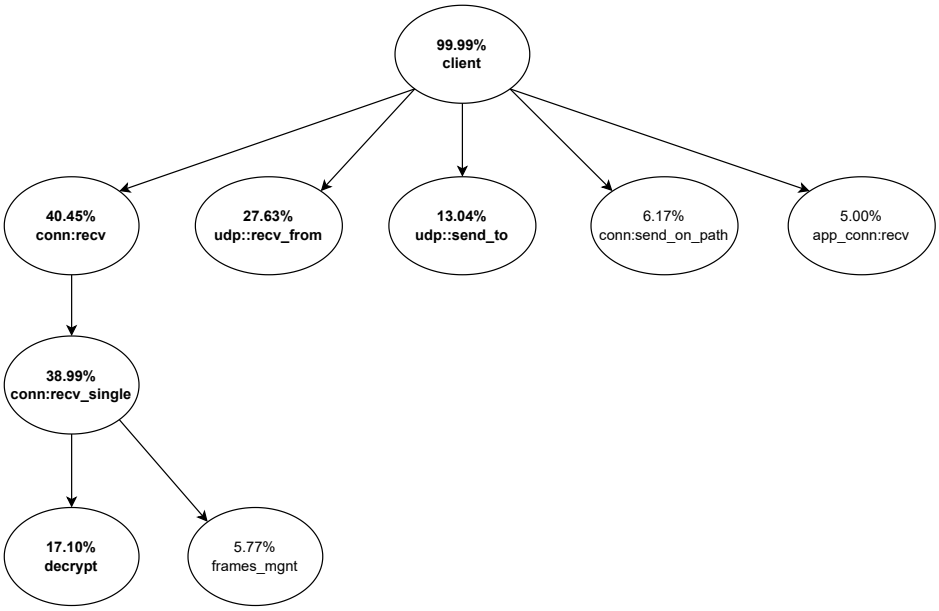


Figure 3.1: The client profile

[1] `https://perf.wiki.kernel.org/index.php/Main_Page`

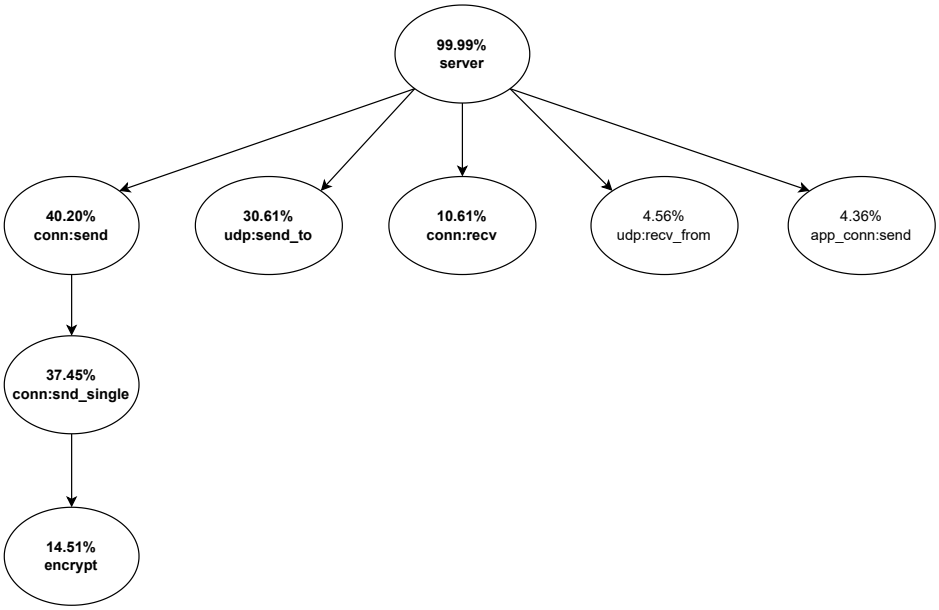


Figure 3.2: The server profile

3.1.3 Possible Theoretical Gains

Given that we are trying to scale the MPQUIC connection with more cores, we can rely on Amdahl's law to calculate the theoretical speedup. The Amdahl law is a computer architecture formula that computes a task's theoretical speedup at constant problem size [28]. The function is provided in figure 3.3. It takes input as two parameters p and s. We define p as the proportion of the CPU cycles that can be parallelized and s as the number of cores available to execute the parallel portion p. In this case, we will assume that the locking of concurrent structures is negligible or near zero and that no external factors influence the performance of our system.

$$y = \frac{1}{(1-p) + \frac{p}{s}}$$

Figure 3.3: The Amdalh's law, y is the theoretical speedup gain.

Based on the profiles, there are two approaches to implementing a multicore architecture. The first approach is to parallelize packet I/O, which accounts for nearly 38% of CPU cycles on a host. The second approach is to parallelize both packet I/O and connection management using the Quiche library. This latter approach increases the parallelizable portion to approximately 84%. As shown in Figure 3.4, the second approach scales significantly better, while parallelizing only packet I/O struggles to achieve a two-fold improvement.

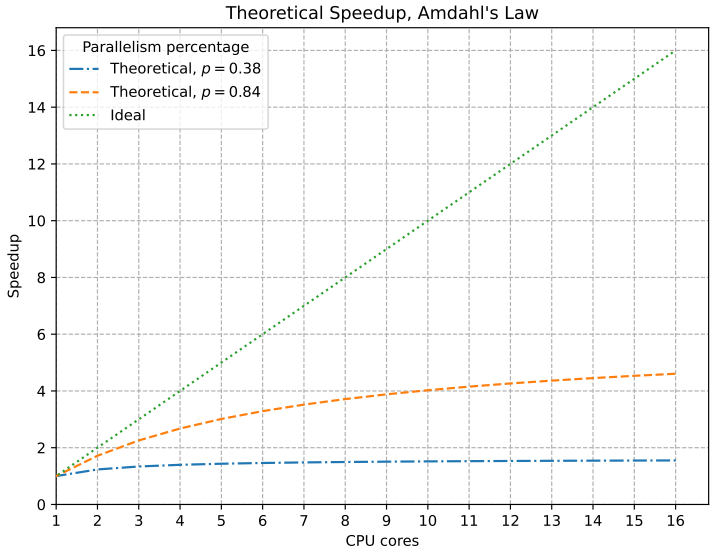


Figure 3.4: The expected speedup with relation to the number of cores for each architecture goal.

3.2 Our solution

As the last section shows, MPQUIC does not support concurrent transfer on both streams and paths. This means two different threads cannot send on two different streams or paths due to the API used. The high-level goal of *mcMPQUIC* is to make the application data transfer as concurrent as possible so that we can further use multiple cores to scale the transfer. By leveraging QUIC's stream multiplexing and MPQUIC's multipath capabilities, mcMPQUIC aims to maximize concurrency within a single connection. Therefore, we pin QUIC streams to paths, supposing the application-level protocol sends data concurrently on multiple streams. The data sent on that stream will only be sent on the specified path by pinning a stream to a path. Then, optionally, we pin the thread executing the path's logic on one core, including the UDP packet I/O.

3.2.1 Modified Application

The multicore application duplicates the pseudocode shown in listing 3.2 per thread. This means that instead of having one core manage multiple paths at a host, mcMPQUIC uses a single thread to handle only one path. This thread will be entitled *Path Thread* for the rest of this manuscript. Figure 3.5 depicts a comparison between the initial MPQUIC architecture and the mcMPQUIC architecture. A *Path Thread* is also in charge of initiating path establishment and path validation for a specific 4-tuple. Once a *Path Thread* finishes sending application data on its path, it notifies the initial *Path Thread*. Hence, the initial *Path Thread* has to wait to receive messages from all other paths and finish its application data transfer to close the MPQUIC connection gracefully.

As we can observe in figure 3.5, MPQUIC has a single data path after receiving data from the UDP sockets. This is the bottleneck that we identified in the previous section. On the other hand, mcMPQUIC has a concurrent data path per thread. However, the control path, which is meant for control frames such as `ConnectionClose` and `NewConnectionId`, is still not concurrent as the frames are passed to the connection through mutual exclusion, as we shall see in the next subsections.

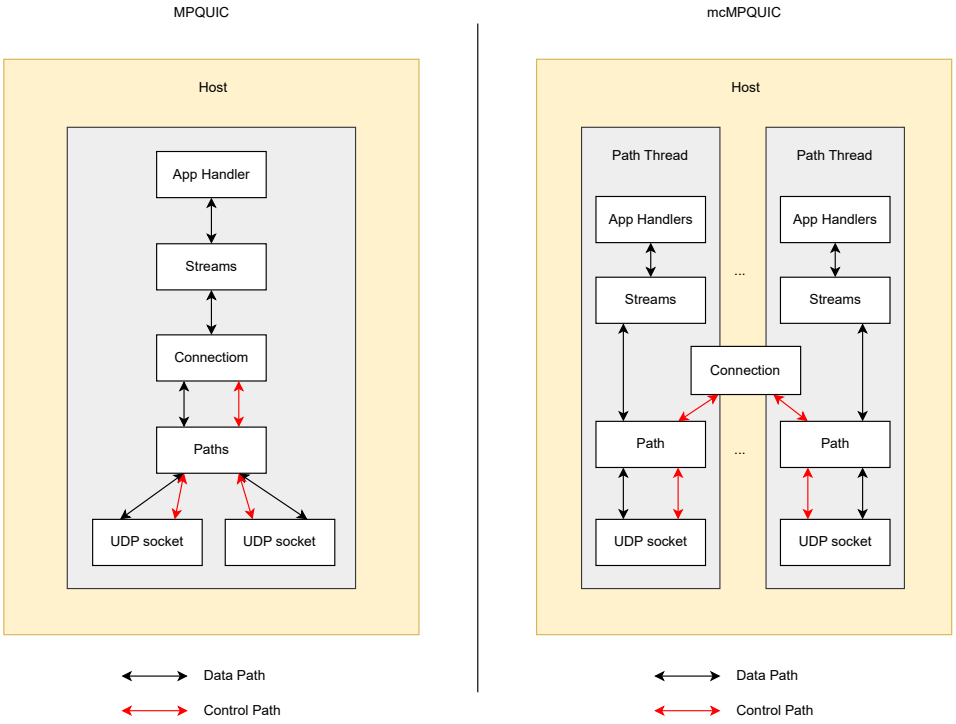


Figure 3.5: mcMPQUIC architecture compared to the single core MPQUIC architecture. This figure shows an example of two paths. In multiple threads, we duplicate the *Path Thread*.

3.2.2 (MP)QUIC semantics

Streams

The QUIC RFC [1] defines two operational points for flow control: connection-level flow control and stream-level flow control. In our design of mcMPQUIC, we perform path-level flow control instead of connection-level flow control to maximize parallelism. This means the `MAX_DATA` frames will handle the offset and manage resources on a single path. We perform synchronization at the connection level in order to respect the constraints on the limit of the maximum streams allowed and also to ensure that the ID of the stream is unique across the connection.

Per-path flow control allows data to be transmitted almost independently across different paths, enhancing concurrency. This approach requires synchronization at the connection level only during the creation and closure of a stream. At all other times, I/O operations are handled on a per-path basis. However, there are a couple of downsides. First, a stream can only use one path for data transfer, though multiple streams can share the resources of a single path. Second, a significant concern is the lack of resilience in case of path failure. If a path fails, streams using that path cannot continue to transmit from the current offset on another active path. Instead, the application must handle the failure by starting from offset 0 and transmitting the data on another stream using a different path.

Acknowledgments

The MPQUIC RFC [10] states that the application may choose whether to send `ACK_MP` frames on another path than the one it is acknowledging the packets. In our case, this would require an extra synchronization overhead to handle acknowledgments across paths. Hence, we opt not to send `ACK_MP` frames on another path than the one we are sending data on to eliminate this potential overhead.

Parallel Packet Protection

MPQUIC uses different packet number spaces per path. This allows for parallel packet protections, as the only dynamic input to the packet protection algorithm is the packet number, which is maintained within each packet number space (see figure 2.2). Once the handshake is completed, the keys needed to derive secrets are well known. Therefore, we can duplicate these keys and maintain a TLS context per path to protect the application data per packet number space. This TLS context duplication allows the parallelization of the packet protection that took more or less 16% of the CPU cycles.

Connection IDs

A path thread only needs information about its connection IDs. However, due to the path establishment mechanism in MPQUIC, the initial path must first communicate available connection IDs to the peer to establish new paths further. Hence, the initial path starts by sharing the available source connection IDs with the peer. Then, other path threads can start their path establishment by checking if the connection has available connection IDs. This is done in a critical section, ensuring that each path thread gets a unique connection ID. Once the path validation is complete, the path copies the connection IDs to a path-local structure and uses this structure to map connection IDs to packet number spaces, map scid to dcid, and vice versa.

Packet Scheduling

Our solution does not use a packet scheduler, as this would increase synchronization across paths to get path-related information such as the RTT, sent packets, and available congestion window. In addition, it would also waste CPU cycles, as the operating system would schedule a path thread, but the path would not do any valuable work if the packet scheduler did not send a packet on the path. The drawback of this design choice is that packet scheduling should be done transparently at the application level. The application could implement a work-stealing mechanism by sending chunks of data to different streams since the path characteristics will be transparent to the application. For example, a path with a higher RTT would take more time to transfer an equal payload, this way a *First-Come-First-Served* work pool of data chunks allows faster paths to send more data.

Server's additional paths

The current MPQUIC draft does not define how to get the server's addition paths, nor does QUIC. To multithread the server using a single address, we would need to use `SO_REUSE_PORT`. The issue with this solution is that it would require more effort to steer a packet to the correct path thread since all paths would share the same server's address. An RFC draft [29] explains how one could exchange additional addresses for

migration or even multipath transfer. The patch from the later RFC can be integrated into mcMPQUIC the same way the additional connection IDs are shared among the peers. Then, synchronization can be used to ensure serializable allocation across path threads. In mcMPQUIC, to pass additional addresses to the server, we use command line arguments to the binaries of both endpoints.

3.2.3 Quiche library decoupling

To better parallelize the data transfer, we divide the operations handled by the `Connection` structure into new abstractions. A comparison is shown in figure 3.6.

From Connection to MulticoreConnection

```
// A QUIC connection suited for multicore
pub struct MulticoreConnection {
  ...
  pkt_num_spaces: packet::PktNumSpaceMap,
  /// Instead of StreamMap we have StreamIds
  stream_ids: stream::MulticoreStreamIds,
  /// Events to for a path.
  per_path_unprocessed_events: BTreeMap<usize, VecDeque<
   MulticorePathPendingEvent>>,
  /// Paths stats of known paths
  paths_stats: BTreeMap<usize, path::PathStats>,
  ...
}
```

Listing 3.3: A summary of discussed fields of the `MulticoreConnection` struct. The full structure can be found in the listing A.2.

`MulticoreConnection` (listing A.2) servers the same purpose as the `Connection`, which is to have all the information representing a QUIC connection. For the datapath, `MulticoreConnection` removes the single entry bottleneck problem seen in `Connection` created by encapsulating `PathMap` and `StreamMap`. We remove the data structure `PathMap`, and we only maintain information about paths. This information is, for example, the lowest active path ID, received/transmitted packets, and path state (active, unused, etc). However, they are provided to the `MulticoreConnection` through method calls by the path thread, which encapsulates the `Path` structure. We apply the same process for the `StreamMap`. We keep the information about a stream's state, such as whether the stream is closed or active.

Replacing Connection API

In mcMPQUIC, the `MulticoreConnection` is no longer the interface to the application for the data transfer. The I/O interface is through the new structure, `MulticorePath`. This structure, as depicted in the figure 3.6, contains most of the necessary information for concurrent data transfer. However, to each method call, the `MulticorePath` takes as an argument the *Read-Write* lock of `MulticoreConnection`. According to the path's status and the connection, it decides when to lock and access the `MulticoreConnection`.

Once the handshake is complete and a path has been established, much information can be copied from the `MulticoreConnection` to reduce overhead. Each `MulticorePath` structure has a local copy of information of the overall connection it needs locally. This

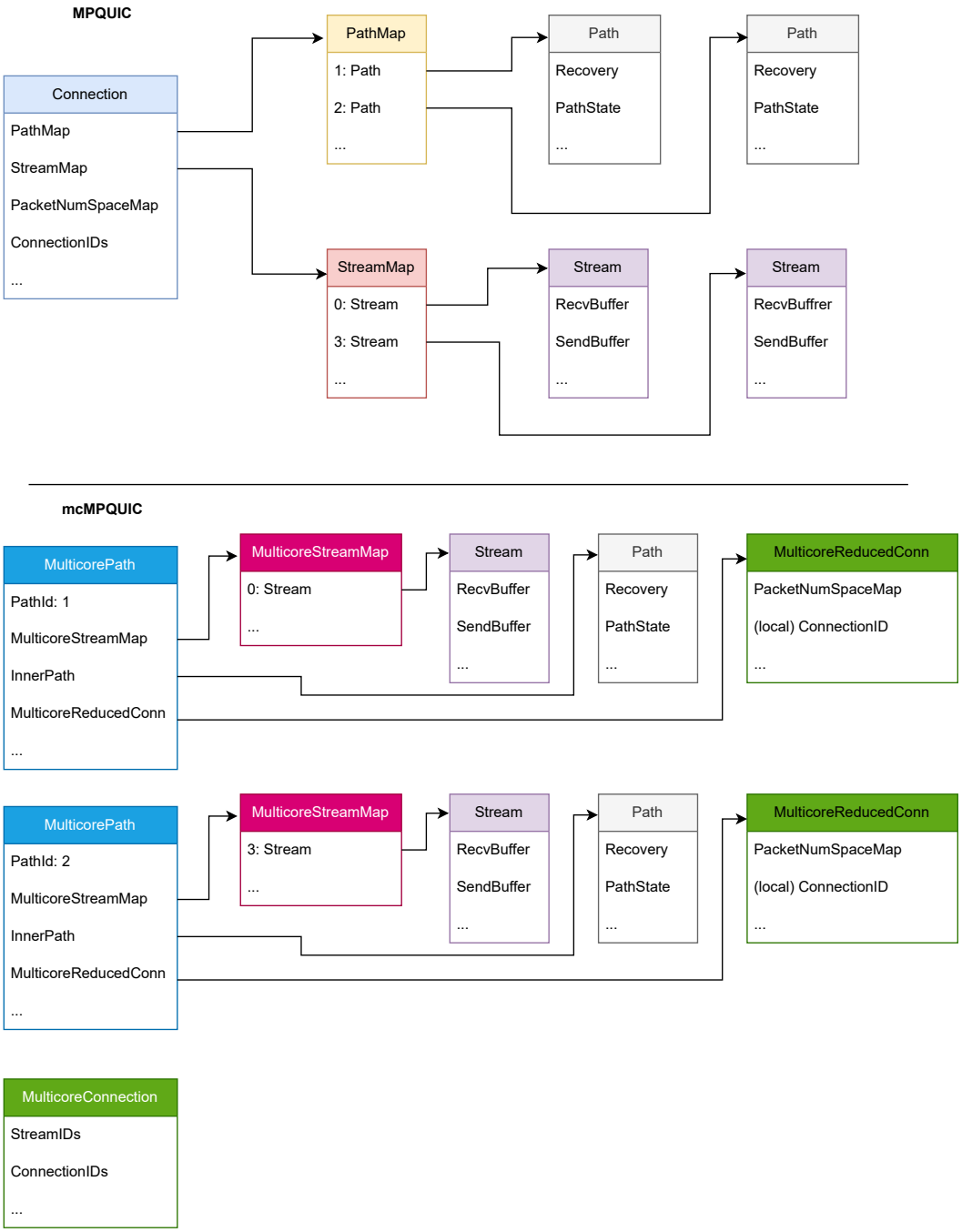


Figure 3.6: A simplified figure showing new data structures by mcMPQUIC

is within a new structure `MulticoreReducedConnection`. This allows each path thread to modify and read data independently, thanks to the independence introduced by the per-path packet number space. The packet protection TLS secrets from the handshake are among the information copied. In addition, we also copy the Connection ID mapping between the Source Connection ID and the Destination Connection ID used for the specific path.

3.2.4 Synchronisation Mechanisms

Path Threads have to synchronize the state of the connection and the state of the application. We decouple QUIC-level synchronization from application-level synchronization. This allows application-level synchronization to communicate application-related events directly, without needing to go through the QUIC connection.

Application level synchronization

The application synchronization mechanism relies on Rust's channels. Rust channels are powerful message-passing pipes to send data across threads. For our simple use-case, we choose *Single Consumer Multiple Producer* (scmp) channels. We choose this channel type because we only want a non-initial path thread to signal the initial *path thread* when they encounter an error or have finished transmitting data. This way, the initial path thread must wait for all paths to finish their transfer to close the connection gracefully. This allows path threads to synchronize without creating additional bottlenecks to the QUIC connection. This application-specific constraint can be changed based on the use case and the workload. In our case, when a path thread has finished sending application data, the initial path thread is notified using these channels.

QUIC connection synchronization

The synchronization on the connection is done using *Read-Write* locks in Rust. We protect the `MulticoreConnection` structure as it contains all the information about the connection that cannot be accessed concurrently.

In Rust, mutual exclusion is done differently than in other programming languages like C or Java. The particularity is that the primitive used for exclusion encapsulates the data structure. Therefore, they can be up to two threads with the protected structure without owning the lock, making it easier to ensure mutual exclusion. Any change in the connection that requires synchronization on all paths is done through queues. The `MulticoreConnection` maintains an event queue per path where if a path has to notify all other threads of an event, it pushes the event to the back of each thread's queue. The concerned thread pops the events from the front and executes the associated processes to handle the event. This allows our design to have less contention and be more asynchronous-friendly, even though we do not use Rust's asynchronous programming. For example, when the remote endpoint sends a connection close request to the initial path, it uses event queues to notify other paths of the connection closure and updates the connection closed boolean in the MulticoreConnection struct.

Our strategy to maintain consistency in the connection is based on two principles. First, locally on a host, shared information about the state of the connection must be written to the `MulticoreConnection` struct. This allows other threads to be aware of the state connection changes, whether it is by modifying a flag in the fields `MulticoreConnection` or sending an event to another path's event queues. Second, only the initial path thread can send connection-wide control frames that change the overall state of the connection to the peer. It ensures that the peer receives and sends connection control frames from the initial path thread. We use this choice in our cases since we do not consider path failures (i.e., a path failing due to link failure at the lower layers). However, the paradigm can be changed to only sending the control frames to the active path with the lowest source path ID.

Conclusion

We initially investigated only parallelizing the Packet I/O and then locking the connection to pass the packets received from the host's network stack. This design actually performed worse than the initial implementation. This was due to the heavy locking of the connection and the critical section being long since each path has to process the whole packet before releasing the lock. On the testbed, the performance halved when compared to MPQUIC.

In this chapter, we first showed the limitation of the baseline MPQUIC implementation. This limitation was due to the Quiche API not being multithread-friendly as there could only be one data path regardless of the multiplexing levels offered by MPQUIC. We decoupled the Quiche library by creating new structures that are multithread aware. From the profiles made using `perf` and Amdahl's law, we estimated the potential gain if we increase the number of paths. We presented the architecture of mcMPQUIC, which aims to duplicate the data path per path and pin streams to paths, and further optionally pin the thread handling the data path to a unique core.

Chapter 4

Measurement Framework

This chapter presents the measurement framework we used to evaluate the performance of the Multipath QUIC implementations as described in the previous chapter.

4.1 Automated framework

To assess the performance of the implementation, we had to design an experimental setup that matched the following three requirements:

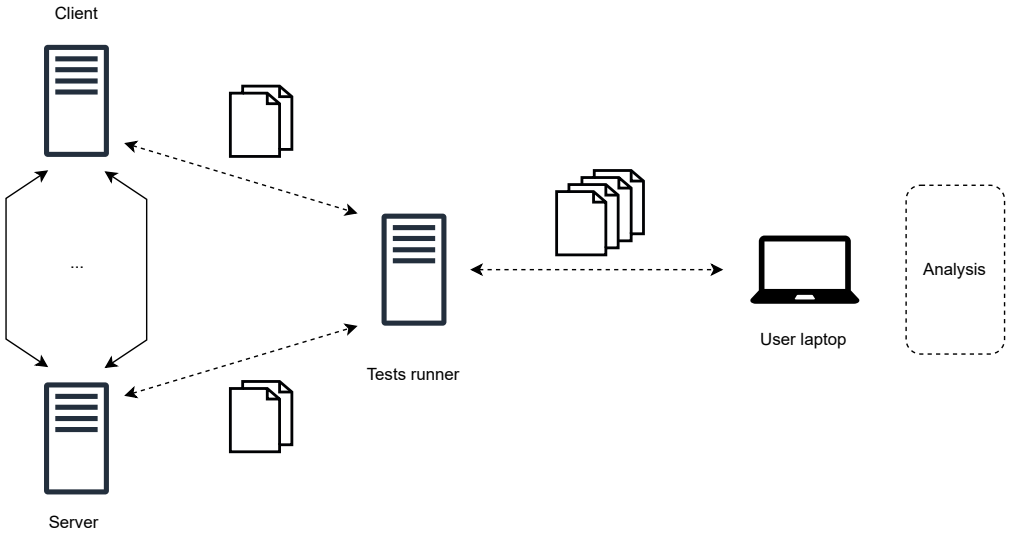


Figure 4.1: The above figure represents how the framework operates. The framework runs on the *Tests runner* but can also be on a user PC. For our setup, we ran the framework on a separate host to prevent connection disruption that may affect the SSH sessions.

- **Repeatability**: Repeatability is the capacity to repeat an experiment easily given the same environment. This requirement allows further students to improve the work done in this thesis. In addition, we want the research community to benchmark and further explore the newly proposed multicore MPQUIC implementation.
- **Reproducibility**: This requirement allows external researchers to run our implementation in different environments (i.e., machines, networks) than ours.

- **Flexibility**: Even though we only consider one implementation in our testbed, our framework should allow other implementations of MPQUIC for an easily configurable performance evaluation.

To design this framework, we took inspiration from the existing QUIC-Interop-Runner implemented by Seemann et al. [30], later extended by Jaeger et al. [2]. Initially, Jaeger et al.'s framework focuses on QUIC performance evaluation between multiple implementations. We rewrote the framework with MPQUIC in mind and focused solely on performance evaluation. In other words, our framework is a subset of their framework alongside some extensions. The repository of this framework is made up of 4504 lines of Jupyter Notebook, 2058 lines of Python, and 736 bash scripts, according to Codetabs[1]. All of the code used for the experiments and the analysis of the logs can be found at:

`https://github.com/vanyingenzi/master_thesis_utilities`

4.2 Configuring an experiment

To run an experiment, the framework requires two files:

Testbed file This file describes the network settings of the hosts (i.e., client, server) the experiment will run on. This file is in a JSON format. The fields specified by this file are: the hostnames used to connect to machines via SSH, a list of IP addresses to be used only by the two endpoints, and finally, a list of interfaces to which the later IP addresses are assigned. An example of this file is the listing 4.1.

```
{
    "server": {
        "hostname": "amd183.utah.cloudlab.us",
        "ips": [
            "10.10.1.1",
            "10.10.2.1"
        ],
        "interfaces": [
            "enp65s0f0np0",
            "enp65s0f1np1"
        ]
    },
    "client": {
        "hostname": "amd204.utah.cloudlab.us",
        "ips": [
            "10.10.1.2",
            "10.10.2.2"
        ],
        "interfaces": [
            "enp65s0f0np0",
            "enp65s0f1np1"
        ]
    }
}
```

Listing 4.1: An example of a testbed JSON file

[1] `https://codetabs.com/count-loc/count-loc-online.html`

Description file This YAML file defines the parameters of the experiment. Within this file, a user specifies the implementations to test, the number of paths, and custom parameters to the binaries. An example of the elements that this file can contain can be found in the listing 4.2.

```yaml
implementations: [mpquic, mcmpquic]
measurement_metrics: [throughput]
repetitions: 10
duration: 20 # seconds
nb_paths: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]
client_prerunscript:
    - path: pre-post-scripts/off-ht.sh
    - path: pre-post-scripts/run-ethtool.sh
    - path: pre-post-scripts/run-netstat.sh
    - path: pre-post-scripts/run-rx-interrupts.sh
    - path: pre-post-scripts/set-rcvbuf.sh
    - path: pre-post-scripts/start-cpu-monitor.sh
      blocking: true
    - path: pre-post-scripts/run-bandwidth-monitor.sh
      blocking: true
client_postrunscript:
    - path: pre-post-scripts/run-ethtool.sh
    - path: pre-post-scripts/run-netstat.sh
    - path: pre-post-scripts/run-rx-interrupts.sh
    - path: pre-post-scripts/set-buffers-default.sh
    - path: pre-post-scripts/stop-cpu-monitor.sh
    - path: pre-post-scripts/stop-bandwidth-monitor.sh
server_prerunscript:
    - path: pre-post-scripts/off-ht.sh
    - path: pre-post-scripts/run-ethtool.sh
    - path: pre-post-scripts/run-netstat.sh
    - path: pre-post-scripts/run-rx-interrupts.sh
    - path: pre-post-scripts/set-rcvbuf.sh
    - path: pre-post-scripts/start-cpu-monitor.sh
      blocking: true
    - path: pre-post-scripts/run-bandwidth-monitor.sh
      blocking: true
server_postrunscript:
    - path: pre-post-scripts/run-ethtool.sh
    - path: pre-post-scripts/run-netstat.sh
    - path: pre-post-scripts/run-rx-interrupts.sh
    - path: pre-post-scripts/set-buffers-default.sh
    - path: pre-post-scripts/stop-cpu-monitor.sh
    - path: pre-post-scripts/stop-bandwidth-monitor.sh
client_implementation_params:
    rmem_value: 100000000 # 100MB
    app_cpu_aff: "0-16"
server_implementation_params:
    rmem_value: 100000000 # 100MB
    app_cpu_aff: "0-16"
build_script: "build_optimised.sh"
```

Listing 4.2: An example of a configuration JSON file

4.3 Implementations

To support flexibility, the initial QUIC-Interop-Runner defines a bash script-based API. A new implementation must provide at least three bash scripts: `build.*.sh`, `run-client.sh`, and `run-server.sh`. These files must be in a directory of implementations (`implementations/`), with the directory's name containing these files as the implementation name. For example, mcMPQUIC will have a directory `implementations/mcmpquic`.

`build.*.sh` is a bash script meant to build the executables. The wildcard is there to show that there can be multiple build scripts, and in order to specify the one to build, a user uses the `build_script` field in the description YAML file.

`run-(client|server).sh` are scripts that run the endpoints binaries. There are environment variables passed to these binaries that they should use, such as the logs dir, the IP address to bind/connect to, the name of the testcase they will run, etc.

4.4 Setting the environment at the hosts

We create directories in each host's `/tmp` directory at each run. These directories depend on the host's role. For a server the folders are going to be : `certs_<rs>`, `logs_<rs>`, `logs_<rs>`. The only repository we create for a client is `logs_<rs>`. `<rs>` stands for a random string. It ensures that interrupted runs cannot interfere with the next runs. Once a run finishes, the directories are copied back to the local host for post-processing.

In addition to the directories, we create a JSON file, `/tmp/interop-variables.json`. This JSON is sent to each endpoint containing the endpoint's related information but also the *implentation parameters* provided by the *Description YAML file*. An example of the content of this file can also be found in the appendixes B.2 and B.1. The framework allows the user to provide bash scripts that will be run before and after each run. These scripts can access the variables within the `/tmp/interop-variables.json` to further customize the setup. For example, in the provided appendices examples, a script reads these variables to set the default UDP receive buffers size before each run and then reset them to the standard size afterward.

4.5 Testcases and Metrics

Within the framework, a testcase is a class that implements the logic behind the timeout of the run and how to calculate a metric based on the files in the logs of each endpoint. When looking at the description file, we have a field named `measurement_metrics`. This array contains measurement testcases that the framework will run. In our case, we focused on the throughput, where we used the logs about the bandwidth at the client.

4.6 Running an experiment

To run the experiment, one executes the framework's binary like below :

```
$ python3 run.py -c throughput.yaml -t testbed.json
```

In the description YAML file, four variables influence the number of runs: `implementations`, `nb_paths`, `measurement_metrics`, and `repetitions`. The number of runs is equal to the product of these variables. The runner will set up the hosts at the start by

installing the required packages. Then it copies the concerned implementations to the `$HOME` directory. For each implementation, the runner compiles with the specified `build_script` and executes the necessary number of runs. In the end, it writes the result in a JSON file named *results.json*. Alongside this JSON file, the runner creates a hierarchical directory for each run, as shown below:

```
+-- logs_<datetime>/
|-- <implementation_0>_<implementation_0>/
|   |-- <testcase_0>/
|   |   |-- <run_idx_0>/
|   |   |   |-- client/
|   |   |   |-- server/
|   |   |...
|   |   |-- <run_idx_n>/
|   | ...
|   |-- <testcase_x>/
|-- <implementation_y>_<implementation_y>/
```

Conclusion

This chapter introduced an automated measurement framework tailored to evaluate Multipath QUIC implementations. Inspired by Seemann et al. [30] and extended by Jaeger et al. [2], our framework simplifies configuration using YAML and JSON files. The framework stores results and logs in a hierarchical directory structure for detailed analysis. This framework is used for our performance evaluation in the following chapter.

Chapter 5

Evaluation

This chapter evaluates our solution, mcMPQUIC, and compares it to the baseline MPQUIC implementation. We first present the evaluated metrics and studied factors. We then describe the experimental design. Finally, we present the experiments and the obtained results. We add a couple of Linux optimization techniques to evaluate their impact on our implementation.

5.1 Metrics and Factors

The metrics we are interested in our evaluation are :

- *Throughput*: The throughput is the amount of data transmitted by the transport protocol over time. Here, we will use Gbps as the unit. The throughput also includes the headers and the protocol's control frames. As stated in the previous chapters, our goal is to maximize the utilization of available bandwidth for application data transfer. We calculate this metric by relying on the `ifstat` logs at the client. When calculating the throughput, we discard the first and last two seconds to only look at the stable inflight throughput. Due to the congestion window's slow start [13], the throughput is low at the start of the connection. At the end of the transfer, the connection is a *draining state* [1], meaning we give the connection enough time to complete gracefully.

- *Goodput*: The goodput is the amount of application data transmitted by the transport protocol over time. Like throughput, the unit will be in Gbps. We aim to evaluate this metric to ensure that our gain in throughput is not due to retransmissions and that the increase in performance is also visible at the application level. This metric is calculated by analyzing the client's logs to get the total bits received by the application protocol and dividing by the total duration of the connection.

- *CPU usage*: Since we are in a multicore environment, we want to ensure that our solution can use the available cores and scale accordingly. CPU usage is measured as the sum of the usage percentages for each core. For instance, an eight-core CPU is considered to be at maximum usage when the total CPU usage reaches 800%. We use `mpstat` logs to monitor the CPU usage during the data transfer throughput.

Amongst the factors that can influence the performance of our implementation, we solely focus on the number of paths. Multipath clients are generally equipped with two

network interface cards (i.e., WiFi and cellular/ethernet). Hence, common multipath transport performance evaluations test with only two paths. In our case, we are interested in seeing how multicore machines can profit from multipath transport protocols to scale packet I/O. Therefore, we will increase the number of paths till we reach the maximum number of cores available on the machine.

The research community highlighted other parameters impacting QUIC's performance, such as the UDP buffer sizes and CPU frequency [2]. We do not evaluate them here. We set the value to 100MB for the UDP buffer sizes so that this parameter does not limit us. We let the Linux Kernel handle the CPU frequency.

5.2 Experimental setup

Topology and machines

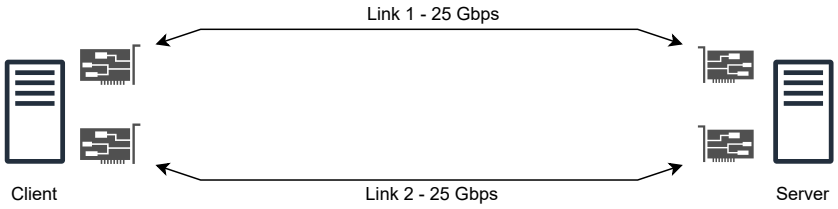


Figure 5.1: The topology on which we run our experiments. The machines and the NICs are symmetric.

For the hosts used in the experiments, we solely use machines from Cloudlab [31] for ease of *repeatability*. Cloudlab [31] is a cloud computing infrastructure designed for researchers. Two 25GbE links connect the client and the server. Unless stated differently, 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 version 5.15.0-86-generic x86_64. These machines are equipped with a 16-core AMD 7302P CPU at 3.00 GHz. Each host has 128GB ECC memory (8x 16GB 3200MT/s RDIMMs). Most importantly, they each feature two dual-port Mellanox ConnectX-5 25Gb GB NICs (PCIe v4.0, x16 lanes). The hosts are multihomed and connect with two networks using the dual port technology of the NIC as seen on figure 5.1.

Default settings

As defined by the NIST [32], we use the suffixes Kilo, Mega, and Giga to represent 10^3, 10^6, and 10^9 respectively. We acknowledge the confusion of these scientific prefixes in the field of computer science, where Kibi (2^{10}), Mebi (2^{20}), and Gibi (2^{30}) are used instead. When comparing our results to related work, we assume they use the NIST standard.

Unless stated otherwise, each experiment run lasts 20 seconds, and we repeat the experiment ten times. We use the framework presented in the previous chapter to automate the measurement process. For congestion control, we use CUBIC, and Hyperthreading is deactivated. The Mellanox NICs used in the setup allow for RSS based on the 4-tuple. We activate this option because otherwise, there is a high risk that a single queue is handling packets, introducing a bottleneck at the NIC layer. No other processes are running on the

machines other than the Cloudlab control plane, whose presence is insignificant due to its near zero CPU usage and a throughput of less than 6.86×10^{-5} Gbps used.

In our CloudLab testbed, the hosts are connected by two subnetworks, limiting us to two paths under normal circumstances. However, to study the scalability of MPQUIC, we require more paths. Therefore, we perform a round-robin assignment, distributing the number of paths between the two networks. For example, if we experiment with five paths, the first network receives three paths, and the second receives two. If the number of paths is even, both networks receive equal paths.

Our application-level protocol operates as follows: After establishing and validating the path, it sends data over a unidirectional, server-initiated stream for a specified duration or until a certain number of bytes have been sent. At the end of the data transmission, an empty FIN stream payload is sent. When the client receives all FINs, it gracefully closes the connection. Finally our binaries are always compiled in production/release mode.

5.3 Experiment 1: Scaling with the number of paths

With this experiment, we aim to test the scalability of our solution with the number of paths. We do not set the CPU affinity on *path threads* of mcMPQUIC, leaving the placement of threads to the OS.

We introduce *shardQUIC* as another baseline to compare our solution to. This implementation is just a bash script that launches concurrent connections that are equal to the number of paths. Each of the connections uses a unique single path, and the connections do not share any state. shardQUIC implements the mechanism of sharding discussed in the chapter 1.

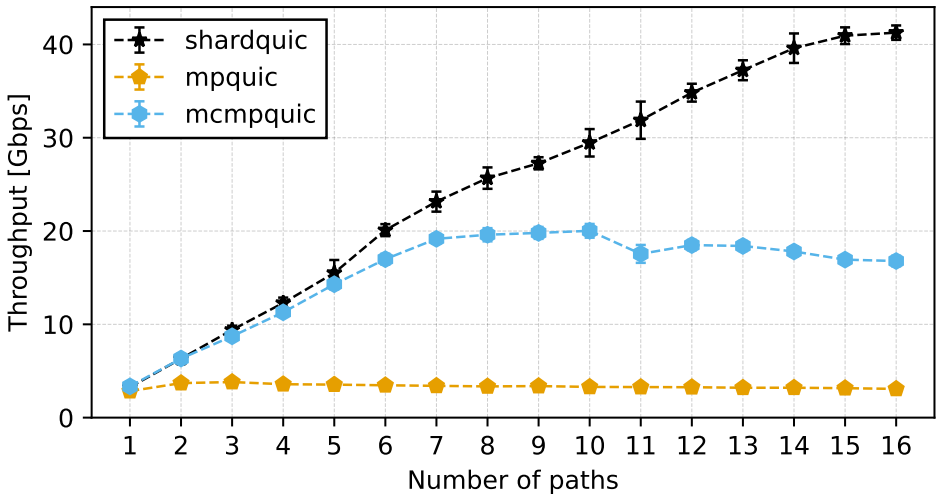


Figure 5.2: A figure displaying the throughput as observed by `ifstat` as we increase the number of paths for each implementation.

mcMPQUIC vs. MPQUIC Figure 5.2 shows that mcMPQUIC achieves a maximum average throughput of 20 Gbps, significantly higher than the 3.8 Gbps achieved by baseline MPQUIC. This represents a throughput gain of over 5x, aligning closely with the theoretical expectations shown in figure 3.4.

MPQUIC Single Core Limitation As shown in Figure 5.2, a single core struggles to handle multiple paths, as discussed in the profiling section in Chapter 3. Figure 5.3 specifically examines the MPQUIC implementation, where we observe a 30% increase in throughput with two paths. This increase reaches a maximum of 34% with three paths. Beyond three paths, connection management becomes a bottleneck because the host cannot process incoming packets any faster, leading to a slight increase in packet delay at the host. Consequently, the congestion control mechanism reduces the transfer rate. Therefore, the benefits of adding more paths diminish after three paths.

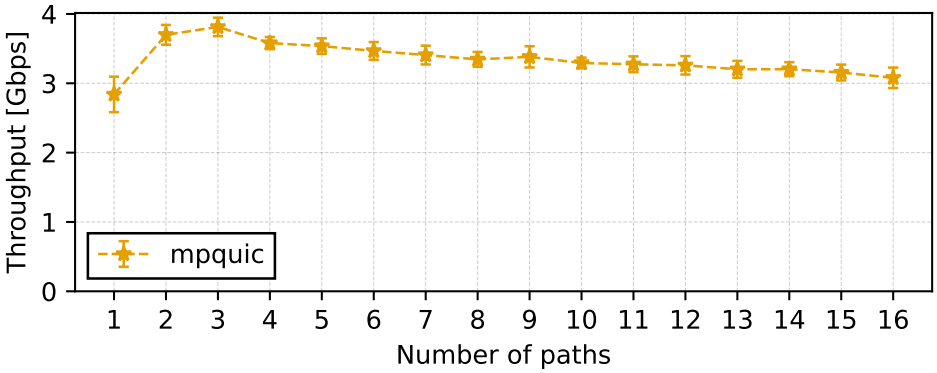


Figure 5.3: A figure displaying the throughput as we increase the number of paths for MPQUIC only.

mcMPQUIC vs. shardQUIC Unlike shardQUIC, mcMPQUIC uses a single connection across multiple paths. Hence, mcMPQUIC has an overhead due to the synchronization across paths. Nonetheless, shardQUIC and mcMPQUIC perform similarly when the number of paths is three or fewer. As the number of paths increases, the performance gap widens in favor of shardQUIC due to the increasing synchronization overhead in mcMPQUIC. Additionally, mcMPQUIC demonstrates linear scaling beyond a single path, reaching knee capacity with eight paths at 19 Gbps and a usable capacity with ten paths at 20 Gbps. The following subsection and experiment are dedicated to identifying why mcMPQUIC stagnates at 20 Gbps.

5.3.1 Identifying the bottleneck

As seen in figure 5.2 beyond ten paths, mcMPQUIC's throughput drops by a few Gbps. Analysis of the NIC queues, as shown in figure 5.4, indicates that this drop is due to an imbalance in the flow distribution among queues and cores. While the unbalanced NIC queues account for the slight decline, further investigation is needed to understand why the throughput flattens at 20 Gbps, especially since the implementation should ideally reach 40 Gbps, as shardQUIC does.

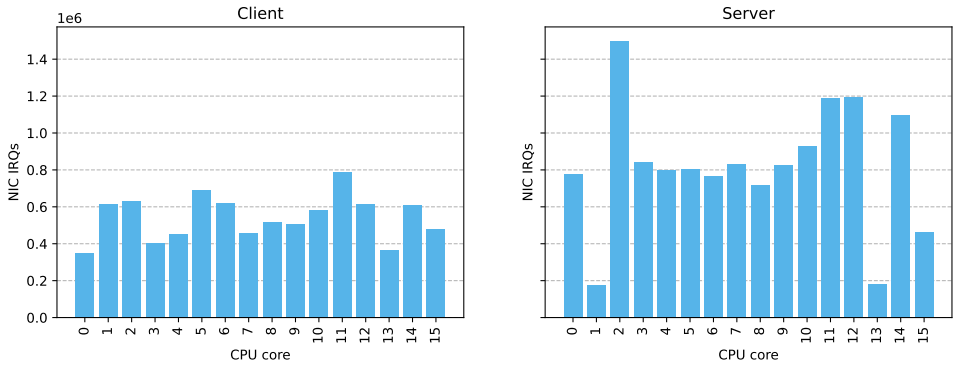


Figure 5.4: This figure shows the total number of NIC interrupts handled per core during the ten runs of mcMPQUIC, with the number of paths equal to 16. Ideally, all the IRQ should be equally distributed, but we can observe that some of the queues are handling up to x2 IRQ at the client. The same phenomenon can be seen on the server.

To identify the bottleneck, we first look at the CPU usage. As seen in figure 5.5, the client's CPU usage scales linearly as the number of paths increases. However, CPU usage at the server also scales linearly but at a lower rate compared to the client. Additionally, the CPU usage plateau at the server when we reach eight threads at 400%. In our experiments, the server sends the data to the client. Since the transport protocol has mechanisms to prevent the server from overwhelming the client, the server will not increase its workload if the client struggles to handle packets. As a result, the server does not operate at full capacity. This indicates that the client is the bottleneck in QUIC, as highlighted in [4].

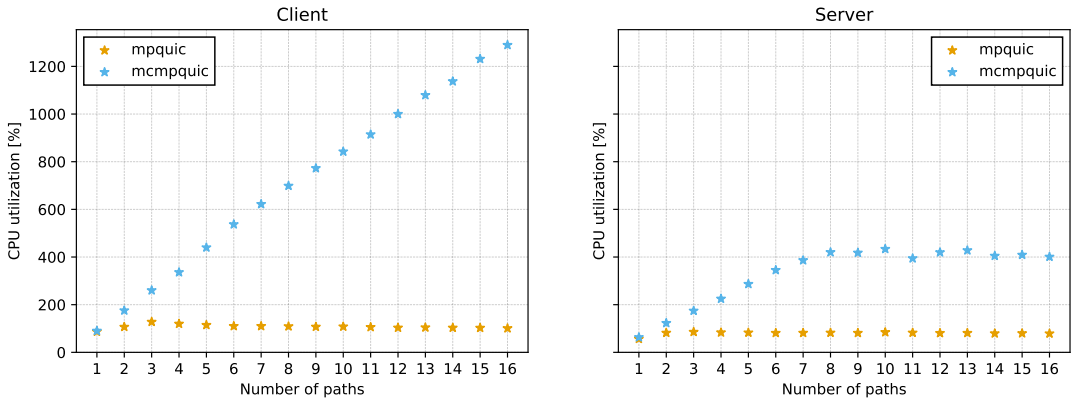
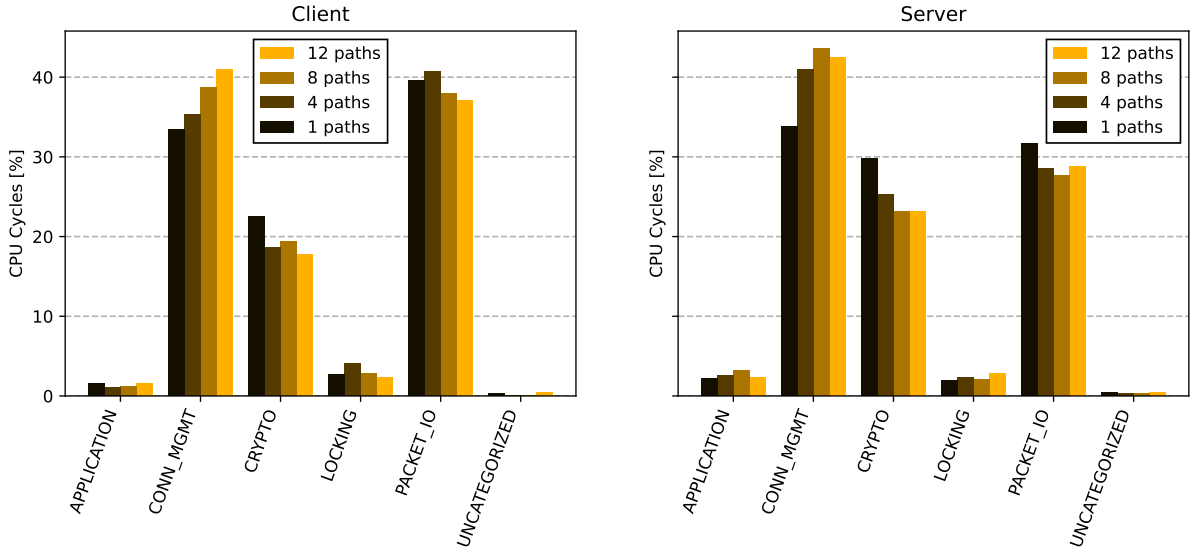


Figure 5.5: The average CPU usage at endpoints, obtained by using `mpstat`.

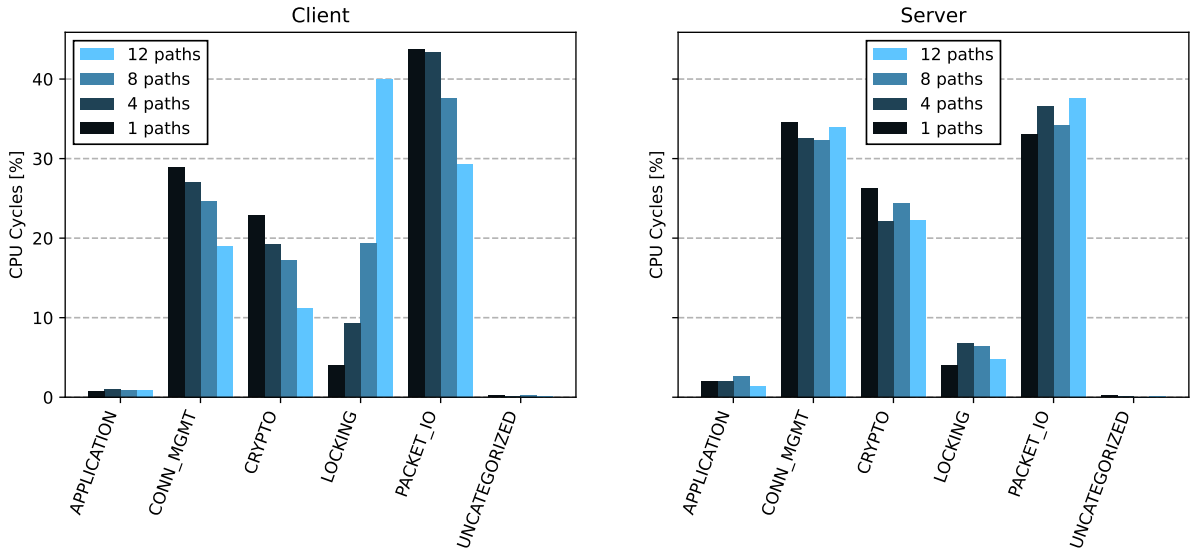
5.4 Experiment 2: Perf Profile on Testbed

To understand the mcMPQUIC throughput plateau at 20 Gbps, we run another set of experiments on the testbed using `perf`. For this experiment, we only profile one endpoint at a time. This means that when we are profiling the client, the server remains untouched and vice-versa. For MPQUIC, we attach `perf` to the process running the binary. For mcMPQUIC, we set CPU affinity and then profile the initial path thread running on the CPU core 0 as the system is nearly symmetric. Indeed, the initial path thread has a smaller quantity of overhead than other path threads as it has to check if it has not

received a message on the application level synchronization. However, this overhead is negligible. We also ran the experiment twice instead of the default 10 times. This is because the `perfreport` command takes a long time to parse the recorded events. For this experiment, we automate the categorization of `perf` samples using regex to map a *callchain* to a category. A *callchain* is a sequence of function call as the were called during an event sampling. We use the function names within the callchain to map the entire callchain to a single category. The mapping details and the scripts for automated categorization are available in the automated framework repository (see chapter 4).



(a) Profile for mpquic



(b) Profile for mcmpquic

Figure 5.6: `perf` profile per implementation as we increase the number of paths.

We reuse some of the categories defined by Jaeger et al. [2] in order to have comparable results. The categories are as follows:

- *Application*: Represents our application-level protocol.

- *Connection Management*: Includes all packet processing done after the packet is read from the socket, except for encryption/decryption.
- *Crypto*: Groups packet protection-related tasks.
- *Locking*: Includes all calls to synchronization primitives in Rust or system calls to `futex`.
- *Packet I/O*: Represents network UDP-related tasks such as sending and receiving.
- *Uncategorized*: Shows the percentage of call chains we were unable to categorize, which in our case is always near zero.

mcMPQUIC Bottleneck By analyzing the mcMPQUIC profile in Figure 5.6b, we observe that *Packet I/O* consumes most of the CPU cycles at the client, up to 8 paths. The expected profile is that *Packet I/O* continues to take most of the CPU cycles. As we increase the number of paths, the profile should remain similar to the initial one-path profile because we are increasing the number of cores/resources, and our system is symmetric across path threads. The fact that we keep more or less the same profile explains the linear scalability as we increase the number of paths, as shown in Figure 5.2. However, upon reaching eight paths, nearly 20% of CPU cycles are spent on synchronization tasks due to mcMPQUIC. This significant increase in *Locking* at the client occurs because the path thread has to lock on the `MulticoreConnection` to check its event queues after receiving UDP packets from a peer. The server handles a relatively constant number of packets, primarily acknowledgments, whereas the client sees an increase in packet numbers, leading to more frequent synchronization. Furthermore, as the number of paths grows, the contention on the lock also rises, resulting in even higher CPU cycles spent on synchronization. At 12 paths, the synchronization overhead becomes significant, reaching 40% of CPU cycles. This is the primary reason mcMPQUIC cannot scale beyond 20 Gbps.

Revisiting MPQUIC Figure 5.6a confirms once again that MPQUIC's *Connection Management* on a single core becomes the main bottleneck as the number of paths increases. As shown in the figure, this category takes more CPU cycles as we increase the number of paths, resulting in *Packet I/O* consuming fewer and fewer CPU cycles.

5.5 Experiment 3: Applying optimizations

Now that we have identified the bottleneck and observed that our solution can't surpass 20 Gbps due to synchronization overhead, we will apply the following kernel software optimization techniques to see if we can achieve 20 Gbps with fewer resources.

CPU Affinity CPU affinity pins a process to a specific CPU core. For our use case, we will pin each thread to a unique CPU core such that no two *path threads* share the same core. Our implementation allows a list of CPU cores to be passed as an argument to set core affinity. One of the benefits of setting CPU affinity is that a process improves cache utilization as it always accesses data on the same core.

Receive Flow Steering (RFS) RFS [33] is a recently proposed optimization technique implemented in the kernel to steer the received traffic to the most appropriate CPU, depending on the location of the application consuming the packet. This optimization aims to increase CPU cache hit efficiency and reduce network latency.

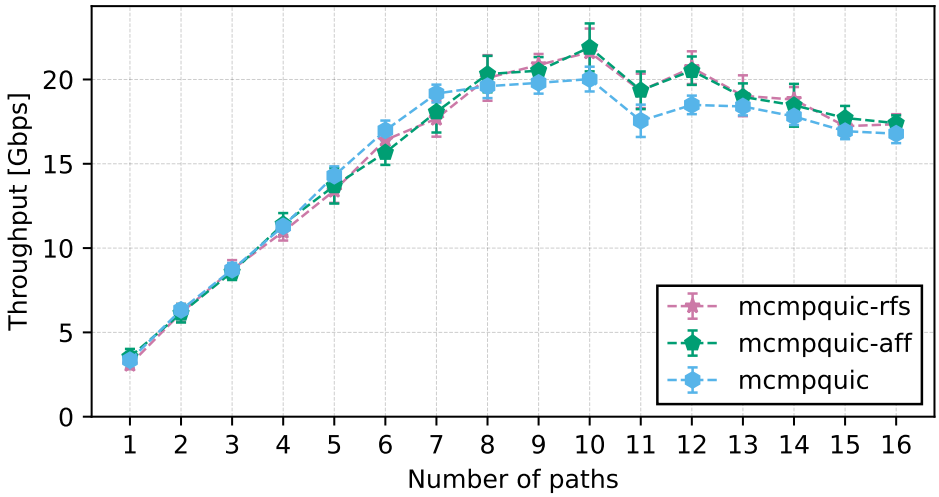


Figure 5.7: This figure shows variations of mcMPQUIC implementation with RFS and CPU Affinity as *mcmpquic-rfs* and *mcmpquic-aff* respectively.

Figure 5.7 illustrates that we can achieve a new maximum throughput of 21 Gbps with ten paths with these optimizations. Before reaching eight paths, the unoptimized mcMPQUIC appears to perform better. However, the optimizations demonstrate an average increase of 1 Gbps in throughput beyond eight paths. These optimizations are supposed to reduce CPU overhead and increase cache hits, we cannot explain why the advantages are only visible after eight paths.

5.6 Goodput of implementations

This section presents the goodput of implementations seen in Experiments 1 and 3.

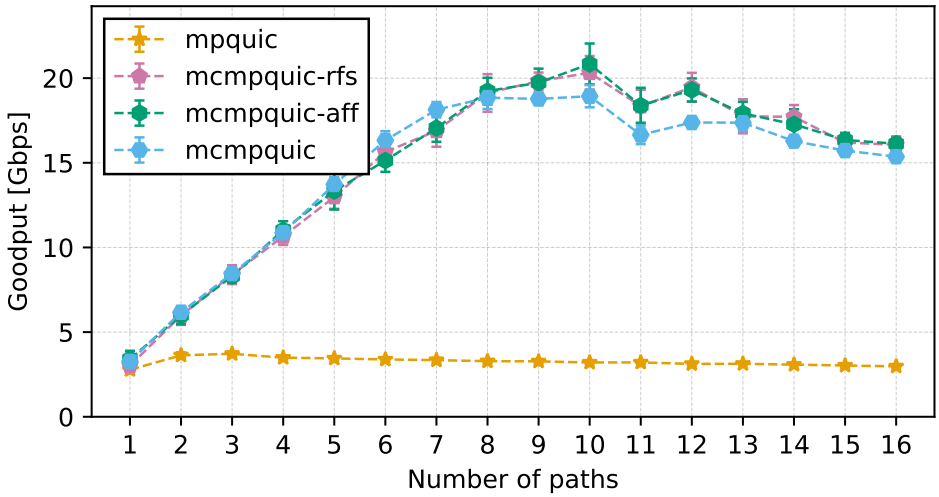


Figure 5.8: The goodput of different MPQUIC implementations as we increase the number of paths.

In figure 5.8, we compare the goodput of all the mcMPQUIC variations discussed in the previous experiments. Overall, we can observe that the goodput has the same line plot shape as the throughput seen in Figure 5.7. This confirms that the performance improvements brought about by mcMPQUIC are visible even at the application level protocol. The highest attained average goodput is 20.8 Gbps with mcMPQUIC with CPU affinity. Compared to MPQUIC, this represents a 5x gain, as observed for the throughput in Experiment 1.

5.7 Experiment 4: Different architecture - CPUs

As with any program running on a machine, the underlying hardware significantly impacts the program's performance. We run mcMPQUIC (with RFS) on `sm220u`[1] servers featuring each Two Intel Xeon Silver 4314 16-core CPUs at 2.40 GHz and Dual-port Mellanox ConnectX-6 DX 100Gb NIC. Regardless of the Intel processors having 2 CPUs, we only use one NUMA node for fair comparison. We do this by setting the CPU affinity to cores of only one NUMA node, and we use Receive Flow Steering to ensure that the cores handle the IRQs on the same NUMA node. The Intel CPU performs better, as seen in figure 5.9, and can attain a maximum average throughput of 27 Gbps with 13 paths. Hence, a throughput increase of 28.5% compared to the maximum achieved throughput of the AMD CPU.

[1] `https://docs.cloudlab.us/hardware.html`

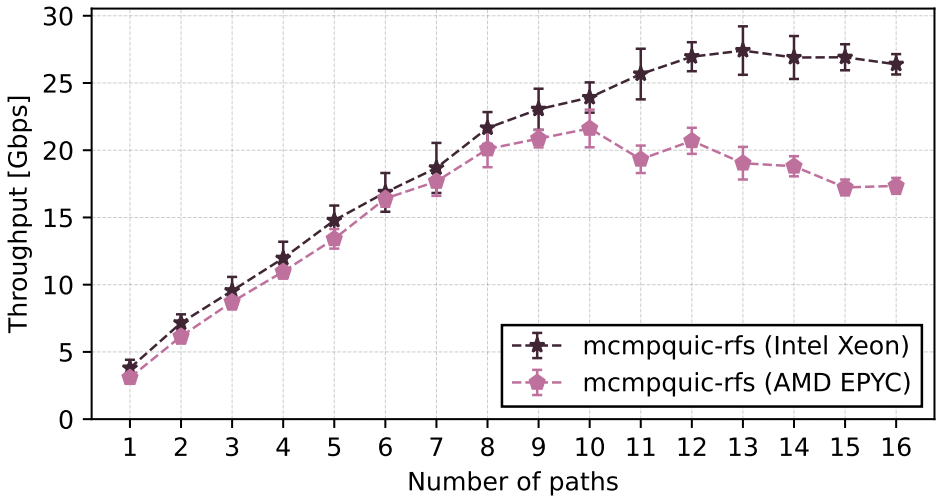


Figure 5.9: Comparison of the throughput achieved on the default AMD EPYC CPU and an Intel Xeon Silver. We use the mcMPQUIC implementation with Receive Flow Steering.

We perform a `perf` profile on the client in figure 5.10 to understand the performance gain observed on the Intel CPU. In comparison to the profiles obtained on the AMD CPU in figure 5.6b, the mcMPQUIC client running on the Intel CPU spends less time on the locking. As synchronization is the bottleneck for the mcMPQUIC client, reducing the CPU cycles spent on locking allows more cycles for Packet I/O, resulting in better performance. The reason why Intel CPUs seem to support locking better than AMD is beyond the scope of this thesis. In conclusion, hardware significantly impacts the scalability of mcMPQUIC as the number of paths increases.

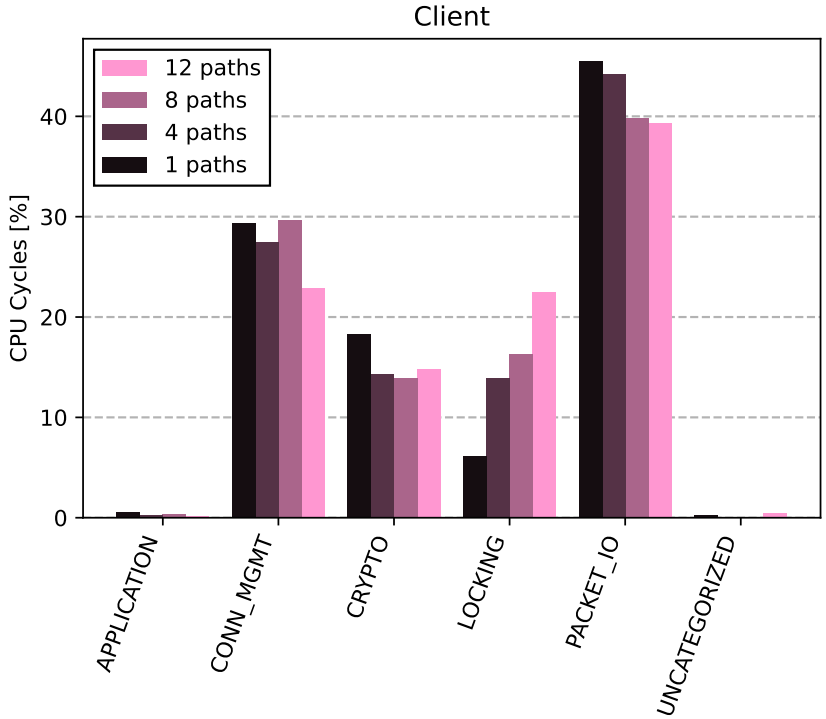


Figure 5.10: Perf profile of the mcMPQUIC client on an Intel Xeon Silver CPU.

Conclusion

In this chapter, we demonstrated that mcMPQUIC achieves a 5x gain in throughput and goodput compared to the baseline MPQUIC. This performance improvement is attributed to mcMPQUIC's ability to utilize multicore environments effectively. We identified synchronization across path threads as the primary bottleneck preventing mcMPQUIC from scaling beyond 21 Gbps on AMD EPYC 7302 CPUs. Moreover, running mcMPQUIC on an Intel Xeon Silver CPU showed even better performance, achieving a throughput of 27 Gbps due to better support for synchronization on the Intel CPU compared to the AMD architecture.

Chapter 6

Conclusion and Further Directions

In this thesis, we have demonstrated that existing research findings on bottleneck issues in QUIC connections also apply to MPQUIC [10]. Packet I/O remains the most dominant bottleneck, particularly on the receiving side. Furthermore, we developed mcMPQUIC, a multicore implementation of MPQUIC based on a pull request in the Quiche library. The primary goal of this architecture was to maximize concurrent data transfer. We designed the architecture by profiting from the parallelization offered by MPQUIC semantics. In addition, we decoupled the Quiche library and introduced multithread-friendly data structures. Our implementation scales linearly up to a throughput of 21 Gbps with 10 paths. We achieve five times faster data transfer than the baseline MPQUIC at this maximum average throughput. We applied kernel optimization techniques to further enhance our implementation to 21 Gbps and we demonstrated that our solution's scalability depends on the hardware it runs on, achieving a maximum average throughput of 27 Gbps on an Intel CPU. Our implementation is well-suited for throughput-sensitive applications with many concurrent data flows. We identified the primary bottleneck of mcMPQUIC as the synchronization overhead, which increases with the number of paths. To facilitate further research and comparison, we proposed a simple framework for reproducing our results and comparing other MPQUIC implementations with ours.

Further Directions

Keeping up with MPQUIC versions Our work on MPQUIC is based on the IETF draft version 6 [10]. MPQUIC is evolving, with newer protocol versions being released at the IETF. By the end of this thesis, version 7 had been released, introducing per-path Connection IDs. This enhancement is particularly beneficial for multicore implementations of MPQUIC, as it reduces the synchronization overhead of Connection IDs. A future direction would be to revisit multicore support for new MPQUIC versions.

Complete implementation Our implementation is a proof of concept and still needs to support connection migration, key updates, and handle path failures. The latter mechanisms can be implemented in mcMPQUIC using the synchronization mechanisms seen in chapter 3.

Synchronisation Overhead We have identified a synchronization bottleneck that limits our solution to scale beyond ten paths. Even though our implementation decoupled the

Quiche library structures to support more concurrency, there is still room for improvement. First, we could use Nonblocking data structures for mutual exclusion. We rely on Rust's RWLock, which blocks the thread when the `MulticoreConnection` is contended. The second improvement is providing cache-friendly data structures as done in [9].

Flow Control While our flow control mechanism is efficient for our current use case, it may become limiting when sending data flows across multiple streams. In future work, we could study and implement concurrent data structures to maintain QUIC's connection-level flow control while enabling highly concurrent stream I/O operations.

Congestion Control CUBIC is a congestion control scheme that can be effectively used in MPQUIC. However, researchers have pointed out that when using classic congestion control schemes such as CUBIC and BBR in a multipath transport protocol, we do not achieve Pareto optimality in terms of bandwidth usage [17], especially when the paths used share a common section in the network. Therefore, it would be beneficial to evaluate coupled congestion control schemes and assess how they impact connection performance when mixing single-path QUIC connections and MPQUIC connections on a link.

Path Scheduler Our current implementation does not incorporate a path scheduler, delegating this responsibility to the application-level protocol. We discussed this approach in chapter 3. A future direction could be to implement a lightweight thread that schedules packets on paths. However, it is crucial that the scheduler remains transparent to the *mcMPQUIC application thread* to avoid wasting CPU cycles.

Appendix A

Quiche Data Structure

A.1 Connection Structure

```rust
/// A QUIC connection.
pub struct Connection {
  /// QUIC wire version used for the connection.
  version: u32,
  /// Connection Identifiers.
  ids: cid::ConnectionIdentifiers,
  /// Unique opaque ID for the connection that can be used for logging.
  trace_id: String,
  /// Packet number spaces.
  pkt_num_spaces: packet::PktNumSpaceMap,
  /// Peer's transport parameters.
  peer_transport_params: TransportParams,
  /// Local transport parameters.
  local_transport_params: TransportParams,
  /// TLS handshake state.
  handshake: tls::Handshake,
  /// Serialized TLS session buffer.
  session: Option<Vec<u8>>,
  /// The configuration for recovery.
  recovery_config: recovery::RecoveryConfig,
  /// The path manager.
  paths: path::PathMap,
  /// List of supported application protocols.
  application_protos: Vec<Vec<u8>>,
  /// Total number of received packets.
  recv_count: usize,
  /// Total number of sent packets.
  sent_count: usize,
  /// Total number of lost packets.
  lost_count: usize,
  /// Total number of packets sent with data retransmitted.
  retrans_count: usize,
  /// Total number of bytes received from the peer.
  rx_data: u64,
  /// Receiver flow controller.
  flow_control: flowcontrol::FlowControl,
  /// Whether we send MAX_DATA frame.
  almost_full: bool,
  /// Number of stream data bytes that can be buffered.
  tx_cap: usize,
```

```
    // Number of bytes buffered in the send buffer.
    tx_buffered: usize,
    /// Total number of bytes sent to the peer.
    tx_data: u64,
    /// Peer's flow control limit for the connection.
    max_tx_data: u64,
    /// Last tx_data before running a full send() loop.
    last_tx_data: u64,
    /// Total number of bytes retransmitted over the connection.
    /// This counts only STREAM and CRYPTO data.
    stream_retrans_bytes: u64,
    /// Total number of bytes sent over the connection.
    sent_bytes: u64,
    /// Total number of bytes received over the connection.
    recv_bytes: u64,
    /// Total number of bytes sent lost over the connection.
    lost_bytes: u64,
    /// Streams map, indexed by stream ID.
    pub streams: stream::StreamMap,
    /// Peer's original destination connection ID. Used by the client to
    /// validate the server's transport parameter.
    odcid: Option<ConnectionId<'static>>,
    /// Peer's retry source connection ID. Used by the client during
     stateless
    /// retry to validate the server's transport parameter.
    rscid: Option<ConnectionId<'static>>,
    /// Received address verification token.
    token: Option<Vec<u8>>,
    /// Error code and reason to be sent to the peer in a
     CONNECTION_CLOSE
    /// frame.
    local_error: Option<ConnectionError>,
    /// Error code and reason received from the peer in a
     CONNECTION_CLOSE
    /// frame.
    peer_error: Option<ConnectionError>,
    /// The connection-level limit at which send blocking occurred.
    blocked_limit: Option<u64>,
    /// Idle timeout expiration time.
    idle_timer: Option<time::Instant>,
    /// Draining timeout expiration time.
    draining_timer: Option<time::Instant>,
    /// List of raw packets that were received before they could be
     decrypted.
    undecryptable_pkts: VecDeque<(Vec<u8>, RecvInfo)>,
    /// The negotiated ALPN protocol.
    alpn: Vec<u8>,
    /// Whether this is a server-side connection.
    is_server: bool,
    /// Whether the initial secrets have been derived.
    derived_initial_secrets: bool,
    /// Whether a version negotiation packet has already been received.
     Only
    /// relevant for client connections.
    did_version_negotiation: bool,
    /// Whether stateless retry has been performed.
    did_retry: bool,
    /// Whether the peer already updated its connection ID.
```

```
  got_peer_conn_id: bool,
  /// Whether the peer verified our initial address.
  peer_verified_initial_address: bool,
  /// Whether the peer's transport parameters were parsed.
  parsed_peer_transport_params: bool,
  /// Whether the connection handshake has been completed.
  handshake_completed: bool,
  /// Whether the HANDSHAKE_DONE frame has been sent.
  handshake_done_sent: bool,
  /// Whether the HANDSHAKE_DONE frame has been acked.
  handshake_done_acked: bool,
  /// Whether the connection handshake has been confirmed.
  handshake_confirmed: bool,
  /// Key phase bit used for outgoing protected packets.
  key_phase: bool,
  /// Whether an ack-eliciting packet has been sent since last
   receiving a
  /// packet.
  ack_eliciting_sent: bool,
  /// Whether the connection is closed.
  closed: bool,
  // Whether the connection was timed out
  timed_out: bool,
  /// Whether to send GREASE.
  grease: bool,
  /// TLS keylog writer.
  keylog: Option<Box<dyn std::io::Write + Send + Sync>>,
  #[cfg(feature = "qlog")]
  qlog: QlogInfo,
  /// DATAGRAM queues.
  dgram_recv_queue: dgram::DatagramQueue,
  dgram_send_queue: dgram::DatagramQueue,
  /// Whether to emit DATAGRAM frames in the next packet.
  emit_dgram: bool,
  /// Whether the connection should prevent from reusing destination
  /// Connection IDs when the peer migrates.
  disable_dcid_reuse: bool,
  /// A resusable buffer used by Recovery
  newly_acked: Vec<recovery::Acked>,
  /// Structure used when coping with abandoned paths in multipath.
  dcid_seq_to_abandon: VecDeque<u64>,
}
```

A.2 MulticoreConnection Structure

```
// A QUIC connection suited for multicore
pub struct MulticoreConnection {
  version: u32,
  ids: cid::ConnectionIdentifiers,
  trace_id: String,
  pkt_num_spaces: packet::PktNumSpaceMap,
  peer_transport_params: TransportParams,
  local_transport_params: TransportParams,
  handshake: tls::Handshake,
  blocked_limit: Option<u64>,
  session: Option<Vec<u8>>,
```

```
    recovery_config: recovery::RecoveryConfig,
    application_protos: Vec<Vec<u8>>, //* Clonable,
    odcid: Option<ConnectionId<'static>>,
    rscid: Option<ConnectionId<'static>>,
    token: Option<Vec<u8>>,
    local_error: Option<ConnectionError>,
    peer_error: Option<ConnectionError>,
    ack_eliciting_sent: bool,
    undecryptable_pkts: VecDeque<(Vec<u8>, RecvInfo)>,
    alpn: Vec<u8>,
    is_server: bool,
    derived_initial_secrets: bool,
    did_version_negotiation: bool,
    did_retry: bool,
    got_peer_conn_id: bool,
    peer_verified_initial_address: bool,
    parsed_peer_transport_params: bool,
    handshake_completed: bool,
    handshake_done_sent: bool,
    handshake_done_acked: bool,
    handshake_confirmed: bool,
    key_phase: bool,
    closed: bool,
    timed_out: bool,
    keylog: Option<Box<dyn std::io::Write + Send + Sync>>,
    qlog: QlogInfo,
    emit_dgram: bool,
    multipath: bool,
    draining: bool,
    dcid_seq_to_abandon: VecDeque<u64>,
    /// Instead of StreamMap we have Stream_ids
    stream_ids: stream::MulticoreStreamIds,
    /// Incremented each time we create a path to create unique ids
    path_id_counter: usize,
    /// Lowest active path id
    lowest_active_path_id: usize,
    /// Lowest active source connection ID
    lowest_active_scid_seq: Option<u64>,
    /// Lowest active destination connection ID
    lowest_active_dcid_seq: Option<u64>,
    /// Paths stats of known paths
    paths_stats: BTreeMap<usize, path::PathStats>,
    /// The mapping from the (local `SocketAddr`, peer `SocketAddr`) to
     pat ids
    addrs_to_paths: BTreeMap<(SocketAddr, SocketAddr), usize>,
    /// Path identifiers requiring sending PATH_ABANDON frames.
    path_abandon: VecDeque<usize>,
    /// Whether a connection-wide PATH_STATUS frame should be sent.
    /// Send a PATH_AVAILABLE is true, PATH_STANDBY else.
    path_status_to_advertise: VecDeque<(usize, u64, bool)>,
    /// Events to for a path.
    per_path_unprocessed_events: BTreeMap<usize, VecDeque<
     MulticorePathPendingEvent>>,
    /// Whether the connection received application close from peer
    pub recv_application_close: bool,
    /// Whether the connection received connection close from peer
    pub recv_connection_close: bool,
}
```

Appendix B

Automated framework

B.1 Interop JSON file at endpoints

```
{
    "implementation": "mpquic",
    "interfaces": [
        "enp65s0f0np0",
        "enp65s0f1np1"
    ],
    "hostname": "amd183.utah.cloudlab.us",
    "log_dir": "/tmp/logs_server_119es63e",
    "www_dir": "/tmp/www_axmfoq39/",
    "certs_dir": "/tmp/certs_u_girly3/",
    "role": "server",
    "filesize": 0,
    "duration": 20,
    "listen_addr": "10.10.1.1:4433",
    "extra_server_addrs": [],
    "rmem_value": 100000000
}
```

Listing B.1: An example of an interop JSON file

```
{
    "implementation": "mpquic",
    "interfaces": [
        "enp65s0f0np0",
        "enp65s0f1np1"
    ],
    "hostname": "amd204.utah.cloudlab.us",
    "log_dir": "/tmp/logs_client_y_nmzah6",
    "sim_log_dir": "/tmp/logs_sim_md1sz3_d",
    "download_dir": "/tmp/download_ojb_w248/",
    "role": "client",
    "server_ip_port": "10.10.1.1:4433",
    "connect_to": "10.10.1.1:4433",
    "extra_server_addrs": [],
    "client_addrs": [
        "10.10.1.2:6780"
    ],
    "rmem_value": 100000000
}
```

Listing B.2: An example of an interop JSON file

Bibliography

[1] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000, May 2021.

[2] Benedikt Jaeger, Johannes Zirngibl, Marcel Kempf, Kevin Ploch, and Georg Carle. Quic on the highway: Evaluating performance on high-rate links. In *International Federation for Information Processing (IFIP) Networking 2023 Conference (IFIP Networking 2023)*, 2023.

[3] Nikita Tyunyayev, Maxime Piraux, Olivier Bonaventure, and Tom Barbette. A high-speed quic implementation. In *Proceedings of the 3rd International CoNEXT Student Workshop*, pages 20–22, 2022.

[4] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z Morley Mao, Feng Qian, and Zhi-Li Zhang. Quic is not quick enough over fast internet. *arXiv preprint arXiv:2310.09423*, 2023.

[5] Manuel Bünstorf and Benedikt Jaeger. Msquic–a high-speed quic implementation. *Innovative Internet Technologies and Mobile Communications (IITM)*, 2023.

[6] Marcel Kempf, Benedikt Jaeger, Johannes Zirngibl, Kevin Ploch, and Georg Carle. Quic on the fast lane: Extending performance evaluations on high-rate links. *Computer Communications*, 223:90–100, 2024.

[7] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 65–77, 2021.

[8] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 767–779, 2022.

[9] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. *mtcp*: a highly scalable user-level *tcp* stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[10] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. Multipath Extension for QUIC. Internet-Draft draft-ietf-quic-multipath-06, Internet Engineering Task Force, October 2023. Work in Progress.

[11] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[12] Martin Thomson and Sean Turner. Using TLS to Secure QUIC. RFC 9001, May 2021.

[13] Jana Iyengar and Ian Swett. QUIC Loss Detection and Congestion Control. RFC 9002, May 2021.

[14] Injong Rhee, Lisong Xu, Sangtae Ha, Alexander Zimmermann, Lars Eggert, and Richard Scheffenegger. CUBIC for Fast Long-Distance Networks. RFC 8312, February 2018.

[15] Andrei Gurtov, Tom Henderson, Sally Floyd, and Yoshifumi Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, April 2012.

[16] Quentin De Coninck and Olivier Bonaventure. Multipath quic: Design and evaluation. In *Proceedings of the 13th international conference on emerging networking experiments and technologies*, pages 160–166, 2017.

[17] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. Mptcp is not pareto-optimal: Performance issues and a possible solution. *IEEE/ACM Transactions On Networking*, 21(5):1651–1665, 2013.

[18] Per Hurtig, Karl-Johan Grinnemo, Anna Brunstrom, Simone Ferlin, Özgü Alay, and Nicolas Kuhn. Low-latency scheduling in mptcp. *IEEE/ACM transactions on networking*, 27(1):302–315, 2018.

[19] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.

[20] Scott Hogg. What about stream control transmission protocol (sctp)? `https://www.networkworld.com/article/741188/cisco-subnet-what-about-stream-control-transmission-protocol-sctp.html`, 2012.

[21] Nick Banks. Msquic - quic performance talk. `https://youtu.be/Icskyw17Dgw?si=AFEnInrxTAxjxEui`, May 2021.

[22] Alessandro Ghedini. Accelerating udp packet transmission for quic. `https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/`, 2020.

[23] Matthew Faulkner, Andrew Brampton, and Stephen Pink. Evaluating the performance of network protocol processing on multi-core systems. In *2009 International Conference on Advanced Information Networking and Applications*, pages 16–23, 2009.

[24] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 337–350, 2012.

[25] Tom Barbette, Georgios P Katsikas, Gerald Q Maguire Jr, and Dejan Kostić. Rss++ load and state-aware receive side scaling. In *Proceedings of the 15th international conference on emerging networking experiments and technologies*, pages 318–333, 2019.

[26] Ashkan Nikravesh, Yihua Guo, Xiao Zhu, Feng Qian, and Z Morley Mao. Mp-h2: a client-only multipath solution for http/2. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[27] Juhoon Kim, Ramin Khalili, Anja Feldmann, Yung-Chih Chen, and Don Towsley. Multi-source multi-path http (mhttp): A proposal, 2013.

[28] Gene M. Amdahl. Computer architecture and amdahl's law. *Computer*, 46(12):38–46, 2013.

[29] Maxime Piraux and Olivier Bonaventure. Additional addresses for QUIC. Internet-Draft draft-piraux-quic-additional-addresses-02, Internet Engineering Task Force, March 2024. Work in Progress.

[30] Marten Seemann and Jana Iyengar. Automating quic interoperability testing. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, pages 8–13, 2020.

[31] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[32] NIST. Prefixes for binary multiples. `https://physics.nist.gov/cuu/Units/binary.html`, 1998.

[33] RedHat. 8.8. receive flow steering (rfs). `https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rfs`.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**