Moving QUIC: From interface to device migration with Portable QUIC State

Vany V. Ingenzi*
UCLouvain
Belgium
vany.ingenzi@uclouvain.be

Louis Navarre
UCLouvain
Belgium
louis.navarre@uclouvain.be

Maxime Piraux
UCLouvain
Belgium
maxime.piraux@uclouvain.be

Olivier Bonaventure
UCLouvain & WELRI
Belgium
olivier.bonaventure@uclouvain.be

ABSTRACT

A reliable transport connection has traditionally been bound to the hosts that are communicating. However, as network systems become more complex and require high availability, there is a need to migrate an established connection beyond its initial hosts.

In this paper, we propose Portable QUIC State (PQS), an implementation-independent representation of a live QUIC connection, with operations to move it across devices. Our solution allows a live migration that does not require network support. We describe the information PQS carries, and we prototype it in two different stacks, quiche and aioquic. We benchmark the solution to show that state serialization is lightweight and has low overhead in the production-ready Quiche implementation. We argue that this will improve network systems by providing a new mechanism to move QUIC endpoints at per-connection granularity, without requiring network support.

1 INTRODUCTION

The end-to-end argument [30] places complex mechanisms, such as reliable transmission, on end hosts to keep the network simple. This is one of the key design choices of the Internet protocols [11]. A transport connection is identified by its 4-tuple: Source's IP, Source's Port, Destination's IP, and Destination's Port. When TCP was separated from IP, it kept the IP addresses of the endpoints in the checksum computation. As a result, TCP connections are bound to their endpoints, and any change to the 4-tuple triggers TCP connection closure. This is a problem for mobile devices whose addresses cannot change without affecting established TCP connections. Several solutions, such as IP mobility [26], multipath extension [15], and migratory TCP [31], have been proposed to preserve the TCP connections while hosts move. It is in this scope that `TCP_REPAIR` was proposed to enable Checkpoint/Restore of ongoing TCP connections [4]. The research community has leveraged `TCP_REPAIR` to migrate live connections at a per-connection granularity. This has led to multiple use cases: connection offload [17, 22], non-stop routing [24], and malicious traffic redirection [8, 13]. However, this mechanism has a practical limitation: it requires network support for live connection migration, specifically packet redirection. This means the network must be well coordinated with the application, which is difficult in practice [22]. Despite these efforts, it remains difficult to move established TCP connections from one location to another on a single device, and even more difficult to move them across devices.

Recently, Network Function Virtualization (NFV) has also increased the need for live migration. By decoupling network functions from hardware, software systems deployed in containers can migrate across hardware. There are various use cases where it would be useful to migrate established connections from one device to another without changing the underlying network [8, 13, 17, 22, 24]. A per-connection migration has lower overhead than migrating a whole docker container (from seconds [9, 28] to milliseconds [17, 22]), and offers a finer-grained control.

QUIC [19] could be a game changer. This modern transport protocol already serves roughly 30% of HTTP requests [1]. It addresses some of the shortcomings of TCP with improved mobility support, byte-stream multiplexing, integrated security, and enhanced privacy. QUIC additionally natively supports connection migration on the client side. This enables, e.g., a smartphone to switch from cellular to Wi-Fi while preserving the established connection. Multipath QUIC [23] goes one step further by enabling simultaneous use of multiple paths for a single connection.

In this paper, we reconsider the assumption that transport connections are bound to the devices where they were initiated. A connection now becomes a temporal association

*Vany V. Ingenzi is a FRIA grantee of the Fonds de la Recherche Scientifique - FNRS.

between processes, which can move at any time from one device to another. To enable the migration of connections *across devices*, rather than simply from one address on a device to another address on the *same* device, we propose a portable representation of an ongoing QUIC connection. We call this Portable QUIC State (PQS). PQS aims to provide an **implementation-agnostic format to enable migration across implementations at a per-connection granularity**. Through analysis of the standard specification, we define a limited subset of necessary information. We implement the design in two different QUIC stacks to show its interoperability. Our benchmarks show that the produced state is lightweight at 2.5 kB, and has a low serialization overhead of 80 µs, when carrying a small chunks of application data. With PQS, network systems can unlock use cases that are not yet possible for QUIC or improve existing network systems with a faster migration time and low overhead.

Ethics. This work does not raise any ethical issues.

2 MOTIVATION

In this section, we first shed light on use cases for live transport connection handover. Then, we discuss the challenges and opportunities of proposing a portable format for a live QUIC connection.

2.1 Why move a live transport connection?

Edge devices offloading. Due to the limited resources on devices such as smartphones and portable computers, there have been works on offloading wireless transfers to reduce resource consumption [7, 20, 32, 36]. Some of these works [7, 20] present solutions in which clients *delegate* their network presence to a proxy, thereby allowing the battery-powered device to enter low-power mode. Tai et al. [32] aims to enable temporal offloading of upload transfers in wireless crowd events, thereby reducing completion times. Yuan et al. [36] offloads packet acknowledgments from the data receiver, allowing the data sender to react more quickly to losses and leading to higher goodput.

L7 load-balancer offload. A Layer-7 load balancer terminates the transport, parses the application request, and selects a backend server. Prior work on TCP [17, 22] shows that handing the established connection directly to the backend removes the balancer from the data path, saving CPU and drastically increasing capacity. As HTTP/3 adoption surges [1], QUIC L7 balancers need similar handoff mechanisms. A previous attempt at per-connection QUIC offload migrates state partially [34], splitting a single connection across multiple backends. Although this increases server throughput by up to 12x, the load balancer must still relay client acknowledgments to each corresponding backend server. This mandatory ACK-routing keeps the proxy in the data path, causing a severe CPU bottleneck of ~45 % at its maximum capacity.

Hiding reboots from peers. On the commercial Internet, some network devices expect no downtime and may handle long-lasting QUIC connections with peers. However, when reboots are needed due to security or performance patches [6], hiding these connection breaks from the peer is an ideal goal. This is called Non-Stop Routing (NSR) [2, 5]. Let's take an example of two eBGP routers running BGP over TCP. For reboots, the local device can move the TCP session to a backup router, potentially leading to minimal impact on the inter-domain traffic during the absence of the initial router. *Tensor* [24] presents a lightweight kernel-modification-free design for Non-Stop-Routing, which relies on TCP state synchronization. However, the industry has started to standardize BGP over QUIC [29], given its benefits [35]. Hence, once BGP-over-QUIC is adopted, a mechanism that allows live QUIC connection migration would enable similar Non-Stop Routing schemes to Tensor, or even seamless router-scheduled handover.

Security Redirection. Public-facing servers receive harmless and malicious traffic. An Intrusion Detection System can decide to drop suspected malicious traffic or move it into a sandbox. Previous works [8, 13] have proposed mechanisms in which suspicious flows are redirected. This way, non-malicious flows are less impacted, and the servers can either learn from the malicious behavior or further inspect the traffic pattern to ensure it's not an error. In both works, they use `TCP_REPAIR` to move TCP connections to either sandboxes or other dedicated machines.

2.2 The opportunities of live migration QUIC

The previous subsection discusses scenarios in which TCP live migration improves network systems. In this subsection, we show how QUIC live migration differs and offers other opportunities.

Multiple Stacks. Unlike `TCP_REPAIR`, which moves state within the single Linux kernel's TCP implementation, QUIC stacks are implemented in user space by design. Each implementation is fine-tuned to its own needs. This raises an opportunity on how to offer a generic format that allows different libraries to interoperate. This is a step further than what `TCP_REPAIR` needs to achieve.

No Need for Network Support. For connection identification, TCP relies on the 4-tuple (source and destination IP addresses and ports). Hence, if `TCP_REPAIR` migrates across devices without network support, this would break the TCP connection. Instead, they rely of network support to hide the change from the remote endpoint by performing traffic steering. On the contrary, QUIC uses Connection Identifiers

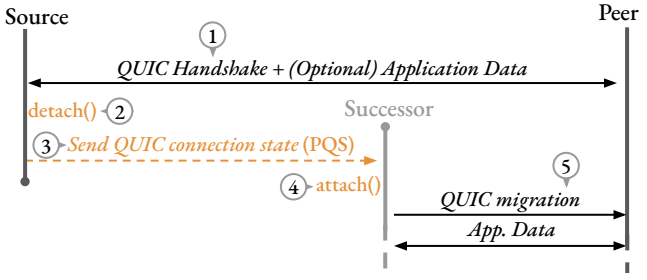

Figure 1: Operations when migrating a client-side connection across devices with PQS. orange text indicates PQS operations, black colored text indicates operations already supported by QUIC.

(CIDs), which separate the connection identity from the underlying network path. This allows a connection to continue even in case of NAT rebinding or network handovers. As a result, the connection does not require network support (e.g traffic steering) for interface/network migration. We leverage this opportunity of QUIC design to even *migrate across devices*.

QUIC's Interface Migration. QUIC [19] has a mechanism called client-side *connection migration*, which means that changing the client's network path is handled seamlessly by the QUIC server. Hence, we could leverage this to migrate client connections across devices without modifying current deployments. However, in the current QUIC specification, a server has no standard way to tell a client mid-connection that its address has changed. This requires specification extensions (Section 3.6).

3 MOVING A QUIC CONNECTION

3.1 Overview

To better understand PQS, we define the three concerned endpoints as *source*, *successor*, and *peer*. The *source* and *peer* are the initial endpoints of the QUIC connection to be moved, and the *successor* replaces the *source* endpoint after the handoff. In our example, only the client-side hands over its connection. We discuss how server-side migration could be supported in Section 3.6.

Figure 1 shows the high-level steps when handing over a connection with Portable QUIC State (PQS). ① The source starts by initiating a QUIC connection to a remote endpoint (peer). During this stage, the source can already exchange application data with the peer. ② When the source decides to move the QUIC connection state, it initiates the *detach* process where the QUIC state is serialized into PQS. Then, ③, the source shares the state (PQS) with the successor, using another connection (Section 3.4). ④ Once the state arrives at the successor, it initiates the *attach* process, where it de-serializes the received state (PQS) and restores the connection on its side. ⑤ Finally, the successor initiates an interface migration to signal to the peer that the connection has changed its network path. The use of QUIC connection migration makes the handover seamless for the server, as the server only sees a standard migration caused by a change in the client's IP address.

3.2 Operations & Requirements

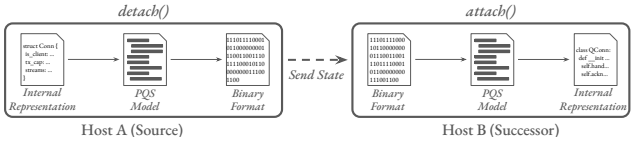

Figure 2: The serialization pipeline behind `detach()` and `attach()`.

Operations. To support Portable QUIC State (PQS), a QUIC library needs to provide two operations: *detach* and *attach*. The key objective of these functions is to enable movement of data from internal representations across different stacks to the PQS model, and vice versa.

- **`detach()`.** This operation returns the state of an ongoing QUIC connection. After the invocation of this operation, the QUIC stack should discard any incoming packets and not send packets to the peer anymore. This is to ensure that the state of the connection stays the same till the successor resumes the connection.
- **`attach()`.** This call allows the QUIC transport layer to resume a QUIC connection from the provided PQS. It is the reverse operation of `detach()`.

Design Goal. We present our design for a single handover. However, PQS should contain all the information necessary to describe an established QUIC connection. This means the successor should be able to move the connection back to the source, or even hand it off to another successor.

3.3 QUIC connection state

The QUIC protocol specifications [18, 19, 33] define the requirements for two endpoints to communicate, assuming the hosts remain identical throughout the connection lifetime. Since more than twenty QUIC stacks co-exist in the wild [3], we define the *Portable QUIC connection State* (PQS) that lets us move this state between implementations. Our format represents the state of an ongoing QUIC connection from a single endpoint's perspective.

Packet Number Spaces. We only keep the Application data's packet number space, as the other packet number spaces are used only during the handshake [19]. Within this packet number space, we store (i) the received packet numbers that need to be acknowledged; (ii) the largest packet number received; (iii) a *timestamp* indicating when the largest

packet was received; (iv) the next packet number to use when emitting future packets; and finally (v) the set of already received packet numbers.

Transport parameters. QUIC transport parameters [19] are exchanged during the connection handshake to define key protocol settings, such as the maximum UDP payload size or acknowledgment delays, ... The QUIC's specification [19] specifies a key-value format to exchange such parameters, which we reuse in PQS. Including this information ensures the successor adheres to the negotiated parameters and also protocol extensions.

Cryptographic Materials. We store the cryptographic secrets and the cipher suite negotiated during the handshake to enable packet encryption and decryption on the successor. This ensures that the successor derives the same encryption materials as the source. Storing the cryptographic secrets rather than just the derived keys allows, the successor to compute new secrets and perform key update if requested by the peer.

Connection IDs. Connection IDs (CIDs) uniquely identify a QUIC connection on a peer [19]. When storing CID information, we record all *non-retired* CIDs from each peer, along with their reset tokens and sequence numbers. PQS requires at least two active CIDs to be able to perform connection migration [19] at the successor. Additionally, we track the currently active pair of connection IDs, enabling the *successor* to migrate the connection using a distinct DCID. Finally, we store the active Connection ID limit for each endpoint to ensure that CID restrictions are consistently enforced throughout the connection.

Streams and Flow Control. Streams are the ordered byte-stream abstraction over which a QUIC connection transports application data [19]. The amount of data transmitted by streams is regulated by flow control. Since the successor must also transfer/receive application data, we represent the byte-stream abstraction as ordered range buffers for each stream. Therefore, we include each stream's receive and send buffers within the connection state. For the receive side, this represents data received by the QUIC stack that has not yet been delivered to the application. For the send side, the data is written to the QUIC stack but has not yet been sent or acknowledged by the peer. We also add control information such as flow control and `FIN` flags. This preserves the ordered nature of the data in each stream and enables the successor to send and receive information without disrupting the QUIC abstraction.

Congestion Control. QUIC endpoints perform congestion control per path [18], hence they must reset their congestion control state upon connection migration [19]. Since delegating a connection implies a migration, we do not include the congestion control state in the portable state. The same applies to the Path Maximum Transmission Unit discovery information.

PQS encoding. Figure 2 shows that the PQS Model is encoded in a binary format before sending state to the successor. Our prototypes encode the QUIC connection state using MessagePack [16] for the binary format. It is an efficient, compact, and widely supported format that supports mappings with keys of any data type, aligning well with the needs of connection state. From empirical measurements, we observe that the size of the connection state when delegating is approximately 2.5 kB in the absence of application/stream data.

3.4 When to transfer state?

The source should carefully time the handover to limit the performance impact on the peer. The strong requirement is that the handover must be done after the handshake is confirmed by both endpoints: the source and the peer. The specification of `detach()` (Figure 1) states that, once invoked, any subsequent packets from the peer should be dropped. This ensures that the state serialized at the source and the state received by the successor remain identical. These potential packet losses may degrade connection performance on the peer side. Further work, will quantify these loses and under which network conditions. We recommend the following measures:

Limit flow control. We recommend using QUIC's flow control to temporarily limit the amount of data the peer can send during the period leading up to migration. For example, in a deployment where handovers are expected, the source can communicate an initial flow control offset of a few kilobytes. The source can then migrate the connection to the successor, and the latter extends the offset once the handover completes. This can temporarily hold data transmission for some time (sub-milliseconds Section 5), but this delay is acceptable compared to incurring packet losses.

No in-flight packets at the source. Additionally, we recommend handing over the connection only once all packets sent from the source to the peer have been acknowledged. PQS carries information about packets still in flight, including stream data that has not yet been acknowledged. The successor might be unaware of acknowledgments sent by the source during handover. This results in the successor retransmitting data the peer has already received, wasting bandwidth.

Transferring QUIC connection state. Portable QUIC State (PQS) describes the connection state and how it should be *detached* from the source and *attached* to the successor. However, how this state is transported between devices depends on the deployment. An ideal handover should be done

over an authenticated, secure, and fast connection. In the proof-of-concept (Section 4), we implemented handover over HTTP/3 for uses-cases where security matters, and over TCP for short network latency.

3.5 Security Considerations

Access Only On The Handed Connection. PQS requires that the source and successor trust each other. This is because the source transfers the 1-RTT TLS cipher suite and secrets to the successor, giving it full control over the QUIC connection. For server-side handover between machines owned by the same operator, this trust assumption is reasonable. It is also acceptable for devices belonging to the same ecosystem and user, for example, an Apple laptop and iPhone. We discuss an extension of PQS that supports *untrusted* successors as further work (Section 6). Furthermore, PQS transfers only the 1-RTT cryptographic material, not the 0-RTT material. This is because 1-RTT keys are valid only for the moved connection, whereas 0-RTT relies on a pre-shared key (PSK) that can be reused in future connections. Excluding 0-RTT material ensures that the successor capabilities are limited to the ongoing session.

Application Level Consideration. Beyond transport-layer secrets, migrated stream data may itself carry sensitive application data, such as OAuth tokens or API keys carried over HTTP. For client-side handover, we therefore recommend performing the handover only after application-layer authentication has completed, so the successor never needs to observe or reuse these credentials.

3.6 Moving QUIC connections on servers

Connection handover can be seen as a two-part process: migrating state from a source to a successor, and then signaling the network path change to the peer. Portable QUIC State (PQS) can be used to move the QUIC connection state on any device, either a server or a client. Network path signaling is the primary distinction between client and server migrations. When a successor loads a client-side PQS connection, it can easily restart the saved connection with the server, which automatically learns the new address of the client (step ⑤ in Figure 2). This does not require any change to the QUIC protocol. On servers, the situation is different. While the QUIC specification allows the server to indicate an alternate address with the `preferred_address` transport parameter, this information is only valid at the beginning of the connection. To use PQS on servers, we need to extend the QUIC protocol to allow servers to advertise their new addresses to clients. This requires a new QUIC frame. Two such proposals have already been made at the IETF [25, 27].

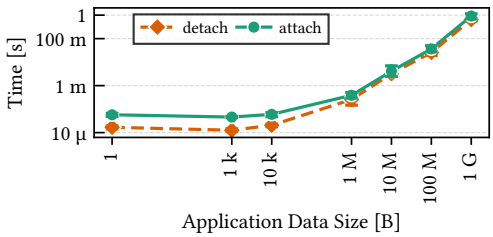

Figure 3: `detach` and `attach` latency in Cloudlfare's `quiche` implementatio, as a function of application data size, on a MacBook Air M2.

4 PROTOTYPE IMPLEMENTATION

To demonstrate the feasibility of migrating connections between different QUIC stacks, we implement PQS in two distinct QUIC implementations. First, Cloudflare `quiche` (`commit` dfeaa589), a production-ready implementation written in Rust. Secondly, `aioquic` (`commit` 9bc1e43), a minimal implementation of HTTP/3 written in Python. These implementations maintain different internal representations of connection state. Both, however, fully implement the QUIC protocol, including connection migration. To verify the empirical correctness of our design, we perform QUIC state transfers between the two implementations. We verify content integrity and confirm that the peer never issues a connection error after the handover. We conduct these correctness tests across different machines on CloudLab [14], and in an emulated network using Linux network name spaces. Both implementations will be made available upon acceptance of the paper. PQS required 2769 and 2732 lines of code in quiche and aioquic respectively, excluding interoperability scripts

5 EVALUATION

5.1 Benchmarking PQS

For PQS to be usable in a high-concurrent workload, it should be efficient, with acceptable overhead latency. As `quiche` and `aioquic` do not yet support a QUIC frame to announce a new server address [25, 27], we focus our evaluation on moving QUIC connections on clients.

Operations Latency. Figure 3 presents the mean time for the PQS's operations on a MacBook Air M2 with the *quiche* implementation, along with the minimum and maximum observed across five runs. We vary the amount of application data the successor must send over the moved connection. We observe that the latency of each operation increases with the size of the application data. This is expected as more application data means that we have more data to (de-)serialize. Additionally, the attach operation takes more time than the detach operation. From Figure 3, we observe that up to 10 kB of application data, we have a latency overhead of less than 20 µs for the detach process and 60 µs for the attach process. This latency is comparable to that reported by Li et al. [22]

	SMAQ [21]	QDSR [34]	PQS (Our work)
Exchange Application Data Before Handover	No	Yes	Yes
Enables Complete Handover	No	No	Yes
Application Data Encryption	Yes	No	Futher Work
Endpoint Migration	Client	Server	Client & Server

Table 1: Comparison of PQS with prior works that move a partial QUIC connection state.

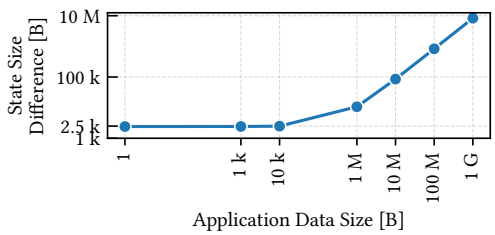

Figure 4: PQS net state size as application data size increases.

when they serialize TCP and TLS state: 13 µs for serialization and 54 µs for de-serialization.

Size overhead. PQS carries control information, and the application data that must be delivered to the application layer or transmitted to the network layer (Section 3.3). We define the *net* state size as the size of the total serialized state minus the size of the application data. Figure 4 shows the net state size as a function of the amount of stream/application data carried by PQS. Below 10 kB of stream data, this net state size stays close to the 2.5 kB baseline reported in Section 3.3. This means that for the transfer sizes typical of L7LB or Security Redirection use cases (Section 2), the overhead remains lightweight. Beyond 10 kB of stream data, the net state size itself starts increasing with the amount of stream data. This is because once the byte stream grows large, it is broken into smaller chunks within the stream's range-buffer representation (Section 3.3), each additional chunk requires its own metadata, adding to the overhead.

6 DISCUSSION

Our work presents a design that represents an established QUIC connection in a complete, implementation-agnostic format and defines operations that move it across devices and implementations. To our knowledge, we are the first work to propose a mechanism for QUIC connection migration across devices and implementations. This makes a range of use cases newly practical for QUIC [7, 20, 24, 32, 36] or improves the performance of existing systems that require live migration [34]. In this section, we discuss related work and future work together, since the comparison to prior systems directly motivates the directions that follow.

Comparison with `TCP_REPAIR`. `TCP_REPAIR` is a Linux kernel Checkpoint/Restore mechanism [12], that allows to serialize a TCP connection. The distinction between PQS and `TCP_REPAIR` matters beyond engineering effort. `TCP_REPAIR` solves a *serialization* problem: one implementation pauses and later resumes itself. PQS solves an *interoperability* problem: one implementation must pause itself in a way that a different, independently developed implementation can resume. Porting `TCP_REPAIR`'s approach to QUIC would only give us dump/restore within a single stack. Additionally, for cross-device live migration, this mechanism faces practical limitations, including the need for network support to redirect packets [17, 22]. PQS does not require such network support, as QUIC supports network path migration and does not rely on IP addresses for connection identification.

QUIC middleboxes. While QUIC was designed to prevent interference from middleboxes, several researchers have proposed QUIC-specific middleboxes. Some of these efforts [21, 34] aim to move part of a QUIC connection's state to improve performance. SMAQ [21] allows inserting a middlebox into a QUIC connection immediately after the handshake to forward application data, similar to a TCP Performance Enhancing Proxy (PEP) [10]. The SMAQ proxy relays stream data by maintaining two separate connections: one facing the client and one facing the server. Its design goal is to improve QUIC performance over heterogeneous network paths (e.g., satellite or cellular networks) by proxying the connection. QDSR [34], as discussed earlier in the motivation, is an L7LB that allows multiple servers to collaborate on a single connection to improve its throughput. QDSR achieves this by offloading different requests to different backend machines: Request I on stream 0 is handled by server X, while Request II on stream 4 is handled by server Y. A summary of the comparison is shown in Table 1. We detail the comparison of PQS across the following characteristics:

- *Exchange Application Data Before Handover:* This characteristic is useful when the source dynamically decides to offload. PQS and QDSR [34] both allow application data to be exchanged before the handover, unlike SMAQ [21].
- *Enables Complete Handover:* By design, both QDSR and SMAQ hand over only part of the connection state from the source, unlike PQS, which transfers the connection state in its entirety. This offers performance advantages in case of server-side handovers as it reduces CPU overhead at the source.
- *Application Data Encryption:* SMAQ allows deriving additional keys to encrypt application data between the server and the client, hiding it from the middlebox. QDSR does not provide an equivalent mechanism, since

offloaded backends handle application data in the clear. PQS could leverage a similar key-derivation mechanism to hide application data from the successor, but we leave this to future work.

- *Endpoint Migration:* SMAQ and QDSR are use-case-specific solutions that focus on one endpoint migrating, either the client or the server. Thanks to the complete state description of PQS we can migrate both endpoints. However, server-side migrating requires extensions (Section 3.6); QDSR uses Multipath QUIC [23], while PQS pushes for minimal client slide changes such as [25, 27].

Beyond the future work discussed above, there question that are to be investigated: extending PQS's state encoding to support other QUIC extensions; quantifying PQS's improvement for each use case in Section 2; comparing handover channels e.g HTTP/3, TCP, or TCP+TLS for minimal latency.

REFERENCES

[1] [n. d.]. Adoption & Usage Worldwide | Cloudflare Radar — radar.cloudflare.com. https://radar.cloudflare.com/adoption-and-usage. [Accessed 15-06-2026].

[2] [n. d.]. BGP Configuration Guide for Cisco 8000 Series Routers, Cisco IOS XR Releases - BGP nonstop routing with stateful switchovers [Cisco 8000 Series Routers] — cisco.com. https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/bgp/bgp-config-cisco8000/r-wrapper-for-graceful-maintenance/c-bgp-nonstop-routing.html. [Accessed 15-07-2026].

[3] [n. d.]. QUIC Interop Runner — interop.seemann.io. https://interop.seemann.io/. [Accessed 08-07-2026].

[4] [n. d.]. TCP connection. https://criu.org/TCP_connection

[5] [n. d.]. Understanding Nonstop Active Routing | Junos OS | Juniper Networks — juniper.net. https://www.juniper.net/documentation/us/en/software/junos/high-availability/topics/topic-map/nonstop-active-routing-understanding.html. [Accessed 15-07-2026].

[6] 2023. Oxy: the journey of graceful restarts — blog.cloudflare.com. https://blog.cloudflare.com/oxy-the-journey-of-graceful-restarts/. [Accessed 08-06-2026].

[7] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Victor Bahl, and Rajesh Gupta. 2009. Somniloquy: augmenting network interfaces to reduce PC energy usage. In *Networked Systems Design & Implementation (NSDI)*.

[8] Frederico Araujo, Kevin W Hamlen, Sebastian Biedermann, and Stefan Katzenbeisser. 2014. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 942–953.

[9] Thad Benjaponpitak, Meatasit Karakate, and Kunwadee Sripanidkulchai. 2020. Enabling live migration of containerized applications across clouds. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE, 2529–2538.

[10] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. 2001. Performance Enhancing Proxies Intended to Mitigate Link-Related Degradations. RFC 3135 (Informational). https://doi.org/10.17487/RFC3135

[11] David Clark. 1988. The design philosophy of the DARPA Internet protocols. In *Symposium proceedings on Communications architectures and protocols*. 106–114.

[12] J. Corbet. 2012. TCP Connection Repair. LWN, https://lwn.net/Articles/495304.

[13] Vitor A Cunha, Daniel Corujo, Joao P Barraca, and Rui L Aguiar. 2020. Using Linux TCP connection repair for mid-session endpoint handover: a security enhancement use-case. In *2020 IEEE conference on network function virtualization and software defined networks (NFV-SDN)*. IEEE, 174–180.

[14] 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. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

[15] A. Ford, C. Raiciu, M. Handley, O. Bonaventure, and C. Paasch. 2020. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684 (Proposed Standard). https://doi.org/10.17487/RFC8684

[16] Sadayuki Furuhashi. 2021. MessagePack. https://msgpack.org.

[17] Yutaro Hayakawa, Lars Eggert, Michio Honda, and Douglas Santry. 2017. Prism: a proxy architecture for datacenter networks. In *Proceedings of the 2017 Symposium on Cloud Computing*. 181–188.

[18] J. Iyengar (Ed.) and I. Swett (Ed.). 2021. QUIC Loss Detection and Congestion Control. RFC 9002 (Proposed Standard). https://doi.org/10.17487/RFC9002

[19] J. Iyengar (Ed.) and M. Thomson (Ed.). 2021. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC 9000 (Proposed Standard). https://doi.org/10.17487/RFC9000

[20] Miguel Jimeno, Ken Christensen, and Bruce Nordman. 2008. A network connection proxy to enable hosts to sleep and save energy. In *2008 IEEE International Performance, Computing and Communications Conference*. IEEE, 101–110.

[21] Mike Kosek, Benedikt Spies, and Jörg Ott. 2023. Secure Middlebox-Assisted QUIC. In *2023 IFIP Networking Conference (IFIP Networking)*. IEEE, 1–9.

[22] Shuo Li, Steven WD Chien, Tianyi Gao, and Michio Honda. 2025. Remote TCP Connection Offload with XO. In *Proceedings of the 9th Asia-Pacific Workshop on Networking*. 37–43.

[23] Yanmei Liu, Yunfei Ma, Quentin De Coninck, Olivier Bonaventure, Christian Huitema, and Mirja Kühlewind. 2026. *Managing multiple paths for a QUIC connection*. Internet-Draft draft-ietf-quic-multipath-21. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/21/ Work in Progress.

[24] Congcong Miao, Yunming Xiao, Marco Canini, Ruiqiang Dai, Shengli Zheng, Jilong Wang, Jiwu Bu, Aleksandar Kuzmanovic, and Yachen Wang. 2023. Tensor: Lightweight bgp non-stop routing. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 108–121.

[25] Marco Munizaga and Marten Seemann. 2026. *QUIC Alternative Server Address Frames*. Internet-Draft draft-munizaga-quic-alternative-server-address-00. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-munizaga-quic-alternative-server-address/00/ Work in Progress.

[26] C. Perkins (Ed.). 1996. IP Mobility Support. RFC 2002 (Proposed Standard). https://doi.org/10.17487/RFC2002 Obsoleted by RFC 3220, updated by RFC 2290.

[27] Maxime Piraux and Olivier Bonaventure. 2025. *Additional addresses for QUIC*. Internet-Draft draft-piraux-quic-additional-addresses-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-piraux-quic-additional-addresses/04/ Work in Progress.

[28] Carlo Puliafito, Luca Conforti, Antonio Virdis, and Enzo Mingozzi. 2022. Server-side QUIC connection migration to support microservice

deployment at the edge. *Pervasive and mobile computing* 83 (2022), 101580.

[29] Alvaro Retana, Yingzhen Qu, Jeffrey Haas, Shuanglong Chen, and Jeff Tantsura. 2026. *BGP over QUIC*. Internet-Draft draft-retana-idr-bgp-quic-08. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-retana-idr-bgp-quic/08/ Work in Progress.

[30] Jerome H Saltzer, David P Reed, and David D Clark. 1984. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)* 2, 4 (1984), 277–288.

[31] Florin Sultan, Kiran Srinivasan, Deepa Iyer, and Liviu Iftode. 2002. Migratory TCP: Connection migration for service continuity in the Internet. In *Proceedings 22nd International Conference on Distributed Computing Systems*. IEEE, 469–470.

[32] Hung-Ta Tai, Wei-Chun Chung, Chi-Jen Wu, Ray-I Chang, and Jan-Ming Ho. 2015. Sop: Smart offloading proxy service for wireless content uploading over crowd events. In *2015 17th International Conference on Advanced Communication Technology (ICACT)*. IEEE, 659–662.

[33] M. Thomson (Ed.) and S. Turner (Ed.). 2021. Using TLS to Secure QUIC. RFC 9001 (Proposed Standard). https://doi.org/10.17487/RFC9001

[34] Ziqi Wei, Zhiqiang Wang, Qing Li, Yuan Yang, Cheng Luo, Fuyu Wang, Yong Jiang, Sijie Yang, and Zhenhui Yuan. 2024. {QDSR}: Accelerating Layer-7 Load Balancing by Direct Server Return with {QUIC}. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 715–730.

[35] Thomas Wirtgen, Nicolas Rybowski, Cristel Pelsser, and Olivier Bonaventure. 2024. The Multiple Benefits of a Secure Transport for BGP. *Proceedings of the ACM on Networking* 2, CoNEXT4 (2024), 1–23.

[36] Gina Yuan, Matthew Sotoudeh, David K Zhang, Michael Welzl, David Mazières, and Keith Winstein. 2024. Sidekick:In-Network Assistance for Secure End-to-End Transport Protocols. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 1813–1830.