

Enhancing QUIC with TAPS

İlhan Can Ateş, Marcel Kempf*

**Chair of Network Architectures and Services*

School of Computation, Information and Technology, Technical University of Munich, Germany

Email: ilhan.ates@tum.de, kempfm@net.in.tum.de

Abstract—QUIC has introduced a modern, encrypted, and multiplexed alternative to TCP. However, its adoption in general-purpose applications is limited by network ossification and API complexity. The IETF Transport Services (TAPS) architecture addresses these challenges by providing a protocol-independent interface that decouples applications from the underlying transport protocols. This paper explores and analyzes the mapping of QUIC's modern features, such as multi-streaming and connection migration, to the abstract TAPS concepts of "Messages" and "Connections," based on an IETF individual draft. We evaluate the design decisions behind this mapping and discuss how the two standards benefit each other. We conclude that TAPS facilitates safe QUIC deployment through automated fallback mechanisms, while QUIC serves as the ideal protocol to demonstrate the benefits of the TAPS architecture.

Index Terms—quic, transport services, taps

1. Introduction

The TCP/IP protocol suite has remained the dominant network architecture since its standardization, with Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) as the principal transport protocols deployed across the Internet [1]. However, as the Internet evolves towards mobile, encrypted, and latency-sensitive applications, the architectural limitations of traditional TCP—namely, head-of-line blocking and vulnerability to middlebox ossification—have become a bottleneck [2]. To address these challenges, the IETF standardized the QUIC protocol [3]. Although originally conceived by Google to accelerate HTTP web traffic [2], the IETF formalized QUIC as a general-purpose transport protocol. QUIC runs over UDP and provides modern features such as stream multiplexing, connection migration, and encrypted transport state.

Despite these technical advantages, network ossification remains a significant barrier to QUIC deployment. Numerous middleboxes and firewalls interfere with unrecognized protocols or options, thereby increasing the likelihood of connection failures for applications adopting new protocols [4]. While QUIC mitigates network ossification by operating over UDP, some restrictive networks block UDP traffic entirely, which necessitates a TCP fallback to maintain reliability [5]. This challenge is further compounded by legacy socket APIs that tightly couple applications to transport protocols, requiring developers to make static protocol choices during the design phase [6]. Consequently, there is limited incentive to migrate to

newer alternatives such as QUIC, and developers frequently default to TCP.

The Internet Engineering Task Force (IETF) working group *Transport Services* (TAPS) was established to address deployment barriers associated with transport-layer protocols. The TAPS architecture specifies an abstract, protocol-independent interface that separates applications from underlying transport mechanisms [6]. Through this interface, applications can express requirements such as reliability and multiplexing, rather than selecting specific protocols. Additionally, the architecture supports dynamic racing mechanisms that attempt multiple protocols (e.g., QUIC and TCP) in parallel and automatically select the best available option without requiring application-level intervention.

This paper introduces the TAPS interface proposed by the TAPS working group, describes and analyzes the current draft mapping of QUIC for the TAPS interface, and discusses the remaining issues.

2. Background

To understand how a QUIC mapping can be built for TAPS, we first take a look at the QUIC protocol itself.

2.1. QUIC

QUIC is a transport protocol built on top of UDP, enabling stateful, encrypted, and reliable communication [3]. Rather than defining an entirely new protocol, it carries packets over UDP datagrams. This approach both circumvents severe forms of network ossification, which block non-standard transport protocols, and at the same time leverages the lightweight nature of UDP. As a result, QUIC reconstructs the reliability and state management features associated with TCP, while also adding confidentiality using integrated TLS [7].

A key feature of QUIC is its use of streams. These lightweight, independent byte streams are multiplexed within a single connection, ensuring reliable and ordered delivery [3]. Unlike TCP, where the loss of a single packet delays the delivery of all subsequent data until the missing packet is retransmitted—a vulnerability known as head-of-line (HOL) blocking—QUIC restricts this behavior to individual streams. Each stream manages its own flow control and retransmits lost packets independently, so packet loss in one stream does not affect others. The result is increased data transfer efficiency.

QUIC communication is encrypted by default, and QUIC handshakes include the TLS parameters. Setting

up a secure TLS layer on top of TCP requires two round trips: one TCP handshake and one TLS handshake. In comparison, QUIC allows for a significantly faster, one-round-trip (1-RTT) establishment of a secure channel. After a connection is established, 0-RTT connections are also possible, further reducing latency [7]. QUIC packets can contain multiple frames, with a frame being the smallest protocol unit. There are various frame types in QUIC. These objects are used to communicate not just application data (e.g., STREAM frames) but also to manage all protocol activity. Examples include ACK frames for acknowledgements, CONNECTION_CLOSE frames for immediate connection termination, and MAX_DATA and MAX_STREAM_DATA frames for flow control on a connection or stream basis, respectively [3]. By exchanging as much control information as possible in frames, QUIC ensures that this information cannot be read in transit, as the payload is encrypted. This design allows easy protocol updates by adding new frame types, unlike TCP, which is difficult to extend due to network ossification, as demonstrated by Honda et al. [4].

QUIC also considers the increasingly mobile nature of the Internet, where clients' connections may not always follow the same network path. Unlike TCP, which requires a new connection if the client IP changes, QUIC enables connection migration between networks (e.g., WiFi to 5G) through persistent connection identifiers, without the added delay of a new connection handshake [3].

Work on QUIC began at Google in 2012 [8]. An Internet Draft submission to the IETF followed in 2015, leading to standardization as an RFC in 2021 [3]. HTTP/3, the current latest version of the Hypertext Transfer Protocol (HTTP), uses QUIC at the transport layer. As of December 2025, about 36.8% of websites use HTTP/3, and roughly 8.6% use QUIC on its own [9], [10].

2.2. TAPS

Transport Services (TAPS) is an asynchronous and event-driven architecture designed to provide applications with a unified interface for transport protocols. As outlined in RFC 9621, its primary goal is to decouple the API from the underlying protocol. This abstraction enables flexibility in protocol selection without requiring applications to hardcode specific transport mechanisms.

2.2.1. Architecture and Objects. The TAPS architecture introduces several objects, defined in RFC 9621 and RFC 9622:

Preconnection: The Preconnection object contains the configuration before a connection is established. Applications specify their preferences, requirements, and constraints using Transport Properties. These can influence the selected protocol or path with Selection Properties, protocol-specific requirements used during connection establishment with Connection Properties, or specific details about how messages are transmitted with Message Properties. Security Parameters are used to specify the criteria and parameters for transport security protocols, such as identities or keys [6].

A Preconnection has different methods to establish a Connection or a Listener. The Listen method can be used to create a Listener that starts listening for connections. A new outbound Connection can be established

using Initiate, or one with 0-RTT, if supported, can be established with InitiateWithSend. A peer-to-peer Connection can be established using Rendezvous [6].

Because the TAPS API is asynchronous, these methods do not block application execution; instead, they initiate a process that progresses through various lifecycle states, eventually emitting an event when the Connection reaches the Ready state to indicate successful establishment.

Connection: The TAPS architecture keeps the definition of a Connection abstract, as each transport protocol has a different understanding of it. For TCP, it can be an active connection, whereas for UDP, it can be the 5-tuple. A set of connections can be part of a Connection Group, where they use the same Connection Properties and can be multiplexed together. Once established, a Connection can Send() and Receive() Messages, or be terminated forcefully with Abort() or gracefully with Close() [6].

Message: TAPS defines a Message as the smallest unit of data that will be transported. The Message abstraction allows applications to work with discrete, bounded units of data rather than unstructured byte streams. This closely matches how most modern applications use the transport layer [6]. For example, HTTP builds its own Request and Response objects on top of TCP, where data is sent as a stream of bytes.

Framer: A Framer can be added to a Connection to add message boundaries and structure to protocols that use unstructured streams of data to fit in the TAPS concept of Messages [6]. A Framer is required in the TAPS implementation of TCP, where data is sent as a byte stream and often restructured into a defined structure on the application layer. In contrast, UDP datagrams already have clearly defined boundaries, making a Framer unnecessary. Similarly, QUIC streams are byte-oriented and require framing to preserve TAPS Message boundaries, as discussed in Section 4.3.2.

2.2.2. Connection Initiation and Protocol Selection. One of the distinguishing features of TAPS is its ability to attempt multiple protocols concurrently during the connection establishment process [11]. For example, if both TCP and QUIC satisfy the application's requirements, the system may initiate both in parallel. QUIC might be unavailable in some environments; in such cases, the system falls back silently to TCP. This approach mitigates ossification by avoiding hardcoded protocol dependencies.

3. Mapping QUIC to TAPS

In this section, we will take a look at the draft mapping of QUIC to TAPS by T. Pauly [12].

3.1. TAPS Objects

A **Connection** is mapped to a single QUIC stream.

A **ConnectionGroup** is mapped to a single QUIC connection. Security parameters are shared between Connections of a ConnectionGroup, according to the TAPS definition [6]. These are mapped to security properties of QUIC, i. e. TLS parameters, which are also shared inside a single QUIC connection [3], [7].

A **Message** corresponds to data sent in a single `Send()` operation. This data is transmitted using QUIC STREAM frames, though a single message may be fragmented across multiple frames depending on the Maximum Transmission Unit (MTU), as well as QUIC's flow control and congestion control mechanisms.

3.2. Connections and Establishment

Calling `Initiate` on a `Preconnection` will establish a new connection and return a new QUIC stream. If there already was an established connection, only a new stream ID will be allocated for use in a new stream. `InitiateWithSend` does the same, but does the connection establishment with 0-RTT if possible.

A `Connection` may not reach the `Ready` state if the new stream cannot be created due to various reasons, for example a maximum stream count limit.

Calling `Clone` on a `Connection` will create a new QUIC stream on the same underlying QUIC connection. The new `Connection` will be put in a `ConnectionGroup` with the old `Connection`.

Multiple `Connections`, or mapped QUIC Streams, received by a `Listener` from the same QUIC `Connection` will be grouped into one `ConnectionGroup`.

3.3. Sending and Receiving

Send will send data using a STREAM frame. The ID stored in the `Connection` object will be used for the Stream ID. No use of a `Framer` is mentioned in the Pauly draft [12]. The need for a `framer` is discussed in Section 4.3.2.

Receive is called by the application to start receiving data.

3.4. Closing Connections

Close closes a QUIC stream using the FIN bit on the stream.

Abort closes a QUIC stream forcefully using a `RESET_STREAM` frame.

CloseGroup closes a QUIC connection after all streams have been closed, using a `CONNECTION_CLOSE` frame.

AbortGroup closes a QUIC connection forcefully using a `CONNECTION_CLOSE` frame.

3.5. Connection Properties

RFC 9622 [13] lists a set of Transport Properties for TAPS with explanations on how implementations should interpret them. The Pauly draft does not specify how Transport Properties will be mapped to QUIC. This seems to be a critical flaw in the current draft, as the mapping is essential for TAPS' protocol selection. However, RFC 9623 includes guidelines for implementing various TAPS features for different types of protocols and some examples, including multiplexed and stream based connections [11]. In the following, we evaluate a small subset of these properties and consider how they can be used within a QUIC implementation.

3.5.1. Reliability. The reliability Transport Property specifies whether the selected protocol should ensure that the messages are sent and received without loss [13]. Standard QUIC can be used if reliability is preferred or required. When an unreliable protocol is needed, the QUIC DATAGRAM extension offers unreliable delivery similar to UDP. However, these datagrams remain subject to the flow and congestion control mechanisms of the underlying QUIC connection. [14].

3.5.2. Multistream Connections. The multistreaming Transport Property specifies whether multiple `Connections` in a `ConnectionGroup` need to be provided by the selected protocol [13]. QUIC provides this by default. If strictly prohibited, this QUIC implementation should not be used. It is possible to make another QUIC mapping that makes this possible, which is discussed in 4.4.

3.5.3. Direction of Communication. The direction Transport Property specifies the direction of communication between the two hosts [13]. QUIC supports both unidirectional and bidirectional streams. These options can directly be used when initiating a new TAPS `Connection` (i. e. creating a new QUIC stream.) [3]

4. Discussion and Analysis

This section evaluates the mapping drafted by Pauly [12], discusses how QUIC and TAPS benefit from this mapping, and analyzes the decisions made in comparison to an alternative mapping.

4.1. How QUIC Benefits from TAPS

TAPS aims to help developers more easily adopt new protocols, such as QUIC. As noted in Section 2.2.2, when TAPS falls back to TCP, it enables developers to experiment with and deploy QUIC without requiring them to rewrite their applications. Developers do not need to create two distinct, tightly coupled implementations; instead, TAPS abstracts the underlying protocol, making integration of QUIC straightforward.

Since QUIC is primarily designed to transport HTTP/3, numerous libraries currently support it. This focus, however, creates a gap for other applications, including online gaming and virtual private networks (VPNs), which could benefit from QUIC's advanced transport features. For instance, online gaming applications can leverage 0-RTT handshakes to reduce initial connection latency, while VPNs can employ connection migration to maintain uninterrupted sessions when a client's underlying IP address changes. The Transport Services (TAPS) interface enables developers to access QUIC through a unified and simplified framework, reducing the complexity of lower-level APIs and facilitating broader adoption across diverse use cases.

Google developed and deployed QUIC, taking advantage of its control over Chromium, which represented more than 78% of the browser market as of September 2025 [15]. This dominant position enabled Google to rapidly implement QUIC support in both its browser and backend infrastructure [2]. In contrast, the process would likely have been much more challenging for a smaller

company with fewer resources and less influence. More broadly, TAPS aims to make protocol integration easier and more accessible to all developers.

4.2. How TAPS Benefits from QUIC

QUIC supports the TAPS objectives by providing protocol features unavailable in standard TCP, such as additional connection properties and protocol options (see Section 3.5). These features enable TAPS to expand its protocol options.

A primary benefit of QUIC for TAPS is the greater flexibility it offers after connection establishment, as demonstrated by QUIC's native support for connection migration. Furthermore, QUIC allows TAPS to fulfill the `zeroRttMsg` property, which enables applications to transmit data during connection setup.

TAPS applications instantiate `SecurityParameters` to request confidentiality and authentication. While legacy transports require a separate security protocol such as TLS, QUIC provides these security properties natively [7]. This enables a TAPS implementation to meet security requirements with a single, integrated protocol stack. As a result, complexity and handshake overhead are reduced.

The `TAPSreliability` parameter can be utilized with the QUIC DATAGRAM extension. This enables a single transport to support both reliable and unreliable message delivery for Transport Services [14]. It enhances flexibility and efficiency in diverse networking scenarios.

4.3. Analysis of Mapping Decisions

4.3.1. Multistreaming. Mapping one QUIC stream to one TAPS Connection does not utilize QUIC's multistreaming capabilities by default. This design can recreate TCP's HOL-blocking problem unless the application deliberately establishes multiple TAPS Connections. As stated in RFC 9621, the TAPS API should not be used to directly choose one protocol. It is designed to allow the caller to only specify requirements without concern for specific protocol choice [6]. However, to utilize QUIC's independent streams, the developer must recognize these mapping implications and ensure the application uses QUIC efficiently, which contradicts the intended use of the TAPS interface.

4.3.2. Framer. The Message-oriented API of TAPS assumes that message boundaries are preserved during transmission, allowing receivers to distinguish where one message ends and another begins. However, unless used with the QUIC DATAGRAM extension, QUIC streams provide a byte stream abstraction without inherent message delimiters, similar to TCP. The Pauly draft does not propose a message framing mechanism, leaving message boundary preservation as a concern for implementation [12].

If TAPS Messages are sufficiently small and each message fits entirely within a single QUIC STREAM frame, frame boundaries could implicitly serve as message boundaries. However, this method is unreliable because QUIC may fragment messages across multiple STREAM frames due to congestion control, maximum packet size, or flow control constraints. Furthermore, the STREAM frame structure is not exposed to the application layer.

Relying on frame boundaries would introduce an implicit dependency on QUIC implementation details that are beyond application control.

A lightweight framing protocol is needed to support TAPS Message semantics over QUIC streams. HTTP/3 uses a simple Type-Length-Value frame format [16]. A similar framing structure can be used for the TAPS mapping of QUIC.

4.4. Alternative Mapping

An alternative mapping is possible, which maps a single QUIC connection to a TAPS Connection, and a TAPS Message to a single QUIC stream. This approach has benefits but also represents a non-traditional use of QUIC depending on the application's needs.

QUIC streams have well-defined lifetimes. A stream is implicitly opened with its first STREAM frame, carries data, and is closed (typically with the FIN bit), with minimal protocol overhead. Using one stream per message removes the need for a TAPS Framer, because stream boundaries implicitly define message boundaries. A stream can even be opened and closed within a single STREAM frame by setting the FIN bit, making this approach efficient for small messages. RFC 9000 explicitly allows an essentially unlimited number of streams, constrained only by the receiving party's advertised limits [3].

However, this approach has limitations. TAPS has a `preserveOrder` Transport Property, which specifies whether Messages need to arrive in the same order they were sent. The one-message-per-stream mapping cannot fulfill this property, as QUIC only guarantees ordering within individual streams, not across them [3]. Additionally, for workloads involving many small messages, the per-stream state management and flow control accounting may become significant, and applications could exhaust the receiver's maximum concurrent stream limit.

TAPS does not natively expose the stream abstraction that sits between connections and messages in QUIC's design. The choice between Connection-to-Stream and Connection-to-Connection mappings can be guided by TAPS Connection or Message Properties, allowing for the full utilization of QUIC's features in each scenario.

5. Conclusion and Future Work

In this paper, we presented QUIC and TAPS, and analyzed an expired draft mapping of QUIC to TAPS proposed by T. Pauly [12].

A central challenge in mapping QUIC to TAPS is preserving message boundaries over QUIC's byte stream abstraction. We identified two viable approaches: using explicit message framing (similar to HTTP/3's frame format) within QUIC streams, or mapping individual TAPS Messages to separate QUIC streams. Each approach presents distinct tradeoffs between implementation complexity, ordering guarantees, and protocol overhead.

As mentioned in Section 4.1, QUIC was primarily designed to transport HTTP/3, which has reduced the immediate need for a generic API implementation. However, as QUIC adoption grows beyond HTTP in other applications, a standardized TAPS mapping becomes increasingly valuable. The availability of a TAPS mapping

democratizes access to QUIC for developers outside of tech giants, and provides the foundation that might allow QUIC to replace TCP in general-purpose applications beyond browsers. The expired draft mapping should be revisited and extended with explicit guidance on message framing to accelerate this process.

References

- [1] G. Fairhurst, B. Trammell, and M. Kühlewind, “Services Provided by IETF Transport Protocols and Congestion Control Mechanisms,” RFC 8095, Mar. 2017. [Online]. Available: <https://www.rfc-editor.org/info/rfc8095>
- [2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi, “The QUIC Transport Protocol: Design and Internet-Scale Deployment,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 183–196. [Online]. Available: <https://doi.org/10.1145/3098822.3098842>
- [3] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” RFC 9000, May 2021. [Online]. Available: <https://www.rfc-editor.org/info/rfc9000>
- [4] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, “Is it still possible to extend TCP?” in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 181–194. [Online]. Available: <https://doi.org/10.1145/2068816.2068834>
- [5] M. Kühlewind and B. Trammell, “Applicability of the QUIC Transport Protocol,” RFC 9308, Sep. 2022. [Online]. Available: <https://www.rfc-editor.org/info/rfc9308>
- [6] T. Pauly, B. Trammell, A. Brunstrom, G. Fairhurst, and C. Perkins, “Architecture and Requirements for Transport Services,” RFC 9621, Jan. 2025. [Online]. Available: <https://www.rfc-editor.org/info/rfc9621>
- [7] M. Thomson and S. Turner, “Using TLS to Secure QUIC,” RFC 9001, May 2021. [Online]. Available: <https://www.rfc-editor.org/info/rfc9001>
- [8] J. Roskind, “QUIC: Design Document and Specification Rationale,” 2013, Google Docs, Accessed: 2025-12-16. [Online]. Available: https://docs.google.com/document/d/1RNHkx_VvKWYwG6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34
- [9] w3techs.com, “Usage Statistics of HTTP/3 for Websites, 13 Dec 2025,” 2025, accessed: 2025-12-13. [Online]. Available: <https://w3techs.com/technologies/details/ce-http3>
- [10] —, “Usage Statistics of QUIC for Websites, 6 Dec 2025,” 2025, accessed: 2025-12-06. [Online]. Available: <https://w3techs.com/technologies/details/ce-quic>
- [11] A. Brunstrom, T. Pauly, R. Enghardt, P. S. Tiesel, and M. Welzl, “Implementing Interfaces to Transport Services,” RFC 9623, Jan. 2025. [Online]. Available: <https://www.rfc-editor.org/info/rfc9623>
- [12] T. Pauly, “A Transport Services Mapping for QUIC,” Internet Engineering Task Force, Internet-Draft draft-taps-quic-mapping-00, Mar. 2022, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/draft-taps-quic-mapping/00/>
- [13] B. Trammell, M. Welzl, R. Enghardt, G. Fairhurst, M. Kühlewind, C. Perkins, P. S. Tiesel, and T. Pauly, “An Abstract Application Programming Interface (API) for Transport Services,” RFC 9622, Jan. 2025. [Online]. Available: <https://www.rfc-editor.org/info/rfc9622>
- [14] T. Pauly, E. Kinnear, and D. Schinazi, “An Unreliable Datagram Extension to QUIC,” RFC 9221, Mar. 2022. [Online]. Available: <https://www.rfc-editor.org/info/rfc9221>
- [15] Cloudflare Radar, “Browser Market Share Report for 2025 Q3,” Sep. 2025, accessed: 2025-12-20. [Online]. Available: <https://radar.cloudflare.com/reports/browser-market-share-2025-q3>
- [16] M. Bishop, “HTTP/3,” RFC 9114, Jun. 2022. [Online]. Available: <https://www.rfc-editor.org/info/rfc9114>