QUIC overview

Flavien Haas, Louis Royer, Daniel Sanchez Pelegrin, Maïmouna Diouf, and Carine Koumsaga

UPSSITECH, Paul Sabatier University campus in Toulouse, France

Abstract. This paper explains the basic principle of QUIC protocol, located just above UDP, in its ways of standardization by the IETF. QUIC is going to be the transport protocol used in HTTP/3, this paper aims to describe how handshake and data transfer are orchestrated while comparing them to TCP HTTP/2. This approach allows explaining the need for QUIC and how it solves HTTP/2's issues like the lack of early data management, TCP's head-of-line blocking or higher number of RTT. Then, it shows an overview of QUIC's next implementations with current research issues and a summary table of the comparison with TCP.

Keywords: OSI model · Transport layer · QUIC · Internet Protocol

1 Introduction

The beginning of the 21st century has been marked with the use of the internet. For communicating, we used in first telephonic lines that have been made only for voice, and therefore haven't been reliable for data. Transport protocols like TCP [18] were created to assure that information will be delivered.

Many round trips are needed to verify that data are received correctly. Now, the complexity of our usages makes the internet an expanding place. In times where connections multiply on a very large scale, these connections must be always faster; and protocol study is necessary to meet those requierements.

QUIC is a new transport protocol developed by many companies. It aims to improve the internet efficiency. For instance, Google and IETF are the main contributors in this project. Google has developed gQUIC, its own version of the protocol and uses it for its own services. On the other hand, IETF's goal is to standardize the protocol to be used worldwide.

QUIC introduces HTTP/3 [1]. The goal is to solve problems related to HTTP/2 linked to the use of TCP: like minimizing the latency of the handshake, introducing early data and prevent TCP's head-of-line blocking [12]. In this document, we will discuss how IETF's QUIC solves these issues while explaining how it works.

The rest of the paper is organized as follows. Section 2 describes Crypto-based Handshake. Data transfer is presented in section 3. Section 4 presents research issues. Section 5 summarize differences between TCP and QUIC.

2 Crypto-based Handshake

To minimize connection establishment latency, QUIC relies on a combined cryptographic and transport handshake. Each connection starts with a handshake phase in which one client and one server establish a shared secret [11, section 5.3].

2.1 Handshake

TLS provides two handshake mode to QUIC [22, section 2.1]:

A full 1-RTT handshake : client is able to send Application Data after one round trip and the server can immediately respond.

A 0-RTT handshake : client uses information previously learned about the server and then is able to send Application Data immediately, but server can accept or reject early data sent with this handshake mode [11, section 17.2.3] (see section 2.1).

(a) 0-RTT handshake [11, Figure 4]. (b) 1-RTT handshake [11, Figure 3].

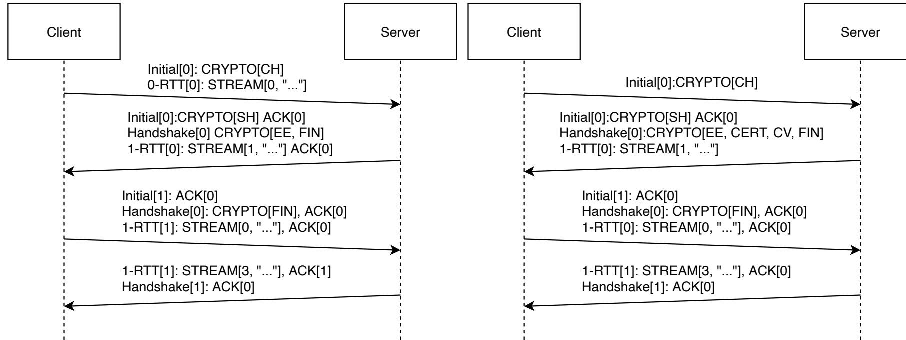


Fig. 1: Comparison between 0-RTT and 1-RTT handshakes.

Figure 1a shows the QUIC handshake with 0-RTT. Figure 1b shows the QUIC handshake with 1-RTT.

0-RTT connection establishment which recently contacted servers allows the client to send application messages before receiving any messages from the server (early data, see [11, section 17.2.3]).

However, 0-RTT lacks certain key security guarantees. In particular, there is no protection against replay attacks in 0-RTT. To enable 0-RTT, QUIC uses connection IDs [21, section 4.3] and also provides a way to terminate a connection no longer desired.

2.2 CRYPTO typed frames

To transmit the cryptographic handshake, QUIC relies on `CRYPTO` typed frames [11, section 19.6]. Each `CRYPTO` frame consists in a contiguous block of handshake data and can be sent in all packet types with the exception of 0-RTT.

As stated in draft IETF QUIC transport [11, section 19.6] the `CRYPTO` frame contains 3 fields:

– `Offset`: A variable-length integer specifying the byte offset in the stream for the data in this `CRYPTO` frame.
– `Length`: A variable-length integer specifying the length of the `Crypto Data` field in this `CRYPTO` frame.
– `Crypto Data`: The cryptographic message data.

There is a separate flow of cryptographic handshake data in each encryption level. Each of these starts at an offset of 0. This implies that each encryption level is treated as a separate `CRYPTO` stream of data.

As opposite to TLS records used with TCP, in QUIC multiple `CRYPTO` frames may appear in the same QUIC packet as long as they are associated with the same encryption level [22, section 4].

2.3 Handshake TLS

TLS is a protocol which permits to establish a secure channel between two communicating entities. It simply expected from the underlying transport a reliable, in-order data stream. TLS function in a client-server mode. It can allow the following security objectives:

– Authentication: The server has to authenticate; the client may not.
– Confidentiality of exchanged data.
– Integrity of exchanged data.

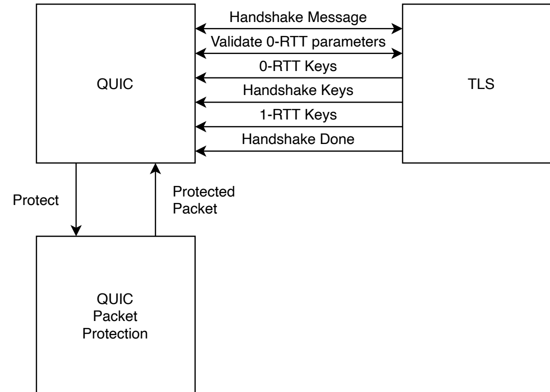
As stated in [22, section 3], QUIC provides the confidentiality and integrity of packets by using keys from TLS Handshake. As with TLS over TCP, once TLS handshake data has been delivered to QUIC, it is QUIC's responsibility to deliver it reliably. Each chunk of data that is produced by TLS is associated with the set of keys that TLS is currently using. If QUIC needs to retransmit that data, it must use the same keys even if TLS has already updated to newer keys [22, section 4].

TLS Handshake and Alert messages are carried directly over the QUIC transport, which takes over the responsibilities of the TLS record layer, while TCP carries TLS record inside TCP packets payload: unlike TLS over TCP, QUIC applications which want to send data do not send it through TLS `application_data` records [22, section 3].

There are two main interactions between the TLS and QUIC components showed in figure 2:

– The TLS component sends and receives messages via the QUIC component, with QUIC providing a reliable stream abstraction to TLS.
– The TLS component provides a series of updates to the QUIC component.

Fig. 2: QUIC and TLS Interactions [22, Figure 4].



As stated in [22, section 4.8], in TLS over TCP, the HelloRetryRequest feature can be used to correct a client's incorrect KeyShare extension as well as for a stateless round-trip check. From the perspective of QUIC, this just looks like additional messages carried in the Initial encryption level.

The inclusion of the QUIC transport parameters extension ensures that handshake and 1-RTT keys are not the same as those that might be produced by a server running TLS over TCP. To avoid the possibility of cross-protocol key synchronization, additional measures are provided to improve key separation [22, section 9.5].

3 Data transfer

In HTTP/2, TCP is used as the transport protocol. Because TCP is reliable and byte-ordered, when multiple streams are multiplexed within the same connection, head-of-line blocking can appear. Head-of-line blocking in TCP is a phenomenon that happens when all the streams of a connection have to wait for a packet of a single stream that is currently lost. This phenomenon occurs because streams are not independent of each other.

QUIC uses streams just as TCP but they are independent within the same connection. They are identified by a unique numerical value within the connection: the `STREAM ID`. This value is a 62-bit integer that can not be reused during the connection.

Within a QUIC connection, data is therefore sent in `STREAM` type frames. This type of frame is composed of the `STREAM ID`, the length of the data sent, the offset within the stream and the data itself. Thanks to the offset, the receiver can reorder data if it does not arrive in order. If a packet is lost and `STREAM` frames were sent within it, only the streams referred by the frames will only have to wait for retransmission. The other streams which are not involved will be able to continue receiving data.

3.1 Loss Detection and Retransmission

As TCP does, QUIC implements an acknowledgment-based loss detection with TCP's Fast Retransmit, Early Retransmit, FACK, SACK loss recovery, and RACK [10, section 5.1] but offers more detailed feedback information for loss detection than TCP [4, page 74].

When a packet is received, once packet protection is removed and all the frames in it have been processed, the receiver has to send an acknowledgment to the sender containing the number of the packet received. By receiving it, the sender can know that the packet sent was successfully received and no retransmission is needed. A timer is settled by the sender when a packet is sent. It prevents the sender to wait for an acknowledgment that never arrives. If an arbitrary amount of time, declared during the handshake, expires, the sender will declare the packet as lost.

In QUIC [10, section 2], an ack-eliciting is a packet that needs to be acknowledged within the maximum delay established. A no ack-eliciting packet is a packet containing `ACK`, `PADDING` or `CONNECTION_CLOSE` frames. This kind of packet does not need to be acknowledged within the maximum delay established. Therefore, the acknowledgment will be sent only if an ACK frame has to be sent for other reasons. For example, if an ack-eliciting packet arrives, the receiver can acknowledge the packet which just arrives and the non-ack-eliciting packet which arrived before.

Sending acknowledgments without control can increase the load between the sender and the receiver. The receiver has to balance its sendings. An acknowledgment of an ack-eliciting packet can be delayed as much as the maximum ACK delay time is not exceeded. In some cases, it will wait until more packets arrive to send a packet with more ACK frames rather than sending an acknowledgment every time an ack-eliciting packet arrives. However, an acknowledgment will have to be sent immediately in the following situations:

– An ack-eliciting packet is received out of order.
– A packet marked with the ECN-CE (Explicit Congestion Notification - Congestion Experienced, see section 3.2) codepoint is received.

An acknowledgment can only be carried in a packet using the same protection as the packet acknowledged. For example, an acknowledgment for an Application data packet will only be carried in an Application data packet and not in a Handshake packet.

An ack-eliciting packet will be determined as lost if:

– The packet is unacknowledged and was sent before and acknowledged packet.
– No acknowledgement for the packet has arrived in an arbitrary amount of time, maximum acknowledgement delay, that will be established by the endpoint.

If a packet is determined as lost the packet itself is not retransmitted. The information will be retransmitted in a new packet with new frames and only if it is necessary: Data in STREAM frames will not have to be retransmitted if a

`RESET STREAM` frame is received and data in `CRYPTO` frames will not have to be retransmitted if the keys of the encryption level are discarded for example.

3.2 Congestion control

QUIC uses TCP's NewReno [10, section 6.3] for congestion control, but unlike TCP, the congestion window is specified in bytes rather than segments. The mechanisms used in TCP to solve congestion problems are therefore implemented in QUIC with a few improvements that should be noted.

Each QUIC connection begins with a slow start. During slow start, QUIC increases the congestion window by the number of bytes acknowledged when each acknowledgment is processed; and in case of loss, the threshold is lowered to half of the last window and the slow start process is resumed. The slow start process ends once the value of the congestion window exceeds the defined threshold, and it switch to avoidance collision.

As in TCP, an additive increase multiplicative decrease (AIMD) is used in this phase. It consists of increasing the congestion window by 1 each time until the next loss. When a loss is detected, NewReno halves the congestion window and sets the slow start threshold to the new congestion window.

The recovery period is slightly different from the TCP recovery definition, which ends when the lost packet that caused the recovery is recognized. Because QUIC does not retransmit lost packets, the end of recovery is achieved when a new packet is sent. The loss of some packets such as Handshake, 0-RTT, and 1-RTT is ignored when their protection keys are not available when they arrive.

When persistent congestion is established, the sender's congestion window must be reduced to the minimum congestion window, which is similar to the sender's response on a Retransmission Timeout (RTO) in TCP after Tail Loss Probes(TLP).

A sender can use the pipeACK method to determine if the congestion window is sufficiently utilized. This underutilization may be due to pacing, which can delay sending packets. However, as in TCP, a sender can implement alternate mechanisms to update its congestion after periods of under-utilization.

QUIC can also use Explicit Congestion Notification (ECN) to signal network congestion before packet loss occurs. ECN is an IP's header field. It is only used if both endpoints and the nodes existing in the path can read and understand the ECN field. The use of ECN must be validated by the endpoints during connection establishment and when migrating to another path. This validation, when packets are sent to a host on a new path is done as follows:

- Setting the ECT(0) code point in the IP header of the first outgoing packets.
- The validation fails if all packets sent with the ECT(0) code point are lost.

If a receiver receives a QUIC packet without an ECT or CE code point in the IP header it does not increase the number of ECNs. The QUIC signals which control congestion are generic and are designed to support different algorithms. The use of NewReno is not mandatory as another congestion algorithm exists

such as Cubic. Moreover, endpoints can use different algorithms between each other.

4 Research issues

Various implementations of IETF's QUIC are already available [7]. One of the first available implementations was picoQUIC. This project is a minimal implementation of QUIC aiming to provide feedback on the development of a QUIC standard in the IETF QUIC Working Group [17]. It is actively tested against other implementations.

Another of the older QUIC implementations is ngtcp2, a community project developed in C. This project provides libraries for client and server roles, with samples of HTTP/3 client and server [15]. quant is a NetApp's implementation providing libraries for both client and server sides [14], but it does not implement HTTP/3.

Quiche is developed by Cloudflare. It provides a low-level API written in Rust for processing QUIC packets. It follows the latest version of QUIC's draft [2]. There is a patch for NGINX implementing Cloudflare's Quiche's server role with a Rust to C binding layer [3].

Recently, HTTP/3 support has been integrated into Curl, although the support is still experimental and needs to be enabled at build time [20]. There are two implementations avaible: one using ngtcp2 libraries and the other using quiche's [19].

Neqo has been developed by Mozilla, it implements a client and a server written in Rust. This project aims to be integrated into Gecko and to provide QUIC support to Firefox in the future [13].

There is also active related work [8] as multipath extension [6], HTTP over multicast quic [16], ns-3 module [5], or wireshark integration [9].

5 General Analysis

Some criteria have been chosen to recapitulate in a summary table, Table 1, the differences between QUIC and TCP. These can be classified into two main categories: general and specified criteria related to data transfer and the handshake.

Firstly, the level of each protocol in the OSI model which marks the possibility of updates in case of a security issue or functionality lack. They help the reader to have a first impression of each protocol.

Secondly, specified criteria allow an understanding of how each protocol will work in a practical case. The way they manage congestion, handle loss detection and retransmission of packets during data transfer is mandatory for transport protocols to understand the main differences.

TCP and QUIC both use Header compression with HTTP to transmit data more efficiently, but HTTP/3 design does not allow the use of HPACK because it would induce Head-of-line blocking. QPACK reuses core concepts from HPACK,

but is redesigned to allow correctness in the presence of out-of-order delivery, with flexibility for implementations to balance between resilience against head-of-line blocking and optimal compression ratio [12, section 1].

Being able to do independent streams allows early data transmission and to avoid Head-of-line blocking. The reduction of the RTT during the handshake is significant on networks with high latency and helps reduce the load on operator networks always overloaded.

Table 1: Comparison between TCP and QUIC for sending a secure web page

Criteria	**TCP (HTTP/2)**	**QUIC (HTTP/3)**
Layer (OSI model)	4	4.5
Ease of updates	hard (kernel space)	easy (user space)
Header compression	HPACK	QPACK
Congestion control algorithm	NewReno	
Loss detection	using ACK packets and timer	
Retransmission packets	same as lost ones	new packet with same data
Independent streams	no	yes
Early data transmission	no	if hosts already know each other and are allowed to

6 Conclusion

This paper showed an overview of the QUIC transport protocol.

The comparison between QUIC and TLS over TCP for a basic web page request showed that many round trips aren't needed to assure speed and security. HTTP/3 is basically HTTP/2 using QUIC transport protocol, it reduces the number of round trips compared to HTTP over TLS/TCP.

The use of QUIC doesn't compromise on security (early data responses used in 0-RTT lacks certain key security guarantees).

The main functionalities of a connected transport protocol for assuring the delivery are designed using already existing algorithms and protocols from TCP.

QUIC strength is that it's based on UDP transport protocol and this gave freedom of design: one of QUIC's power is to reside in the user space so it is more able to change.

There is an active community around the development of different implementations as shown in section 4. Network analysis tools like ns-3 or Wireshark are the firsts to get QUIC support as they're useful for development.

References

1. Bishop, M.: Hypertext Transfer Protocol Version 3 (HTTP/3). Internet-Draft draft-ietf-quic-http-27, Internet Engineering Task Force (February 2020), https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-27, Work in Progress
2. Cloudflare: Quiche (2020), https://github.com/cloudflare/quiche
3. Cloudflare: Quiche's nginx patch (2020), https://github.com/cloudflare/quiche/tree/master/extras/nginx
4. Cui, Y., Li, T., Liu, C., Wang, X., Kühlewind, M.: Innovating Transport with QUIC: Design Approaches and Research Challenges. IEEE Internet Computing **21**(2), 72–76 (Mars 2017). https://doi.org/10.1109/MIC.2017.44
5. De Biasio, A., Chiariotti, F., Polese, M., Zanella, A., Zorzi, M.: A quic implementation for ns-3. In: Proceedings of the 2019 Workshop on ns-3. p. 1–8. WNS3 2019, Association for Computing Machinery, New York, NY, USA (June 2019). https://doi.org/10.1145/3321349.3321351
6. De Coninck, Q., Bonaventure, O.: Multipath Extensions for QUIC (MP-QUIC). Internet-Draft draft-deconinck-quic-multipath-03, Internet Engineering Task Force (August 2019), https://datatracker.ietf.org/doc/html/draft-deconinck-quic-multipath-03, Work in Progress
7. IETF QUIC Working Group: Implementations (2020), https://github.com/quicwg/base-drafts/wiki/Implementations
8. IETF QUIC Working Group: Related Activities (2020), https://github.com/quicwg/base-drafts/wiki/Related-Activities
9. IETF QUIC Working Group: Tools (2020), https://github.com/quicwg/base-drafts/wiki/Tools
10. Iyengar, J., Swett, I.: QUIC Loss Detection and Congestion Control. Internet-Draft draft-ietf-quic-recovery-26, Internet Engineering Task Force (February 2020), https://datatracker.ietf.org/doc/html/draft-ietf-quic-recovery-26, Work in Progress
11. Iyengar, J., Thomson, M.: QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-27, Internet Engineering Task Force (February 2020), https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27, Work in Progress
12. Krasic, C., Bishop, M., Frindell, A.: QPACK: Header Compression for HTTP/3. Internet-Draft draft-ietf-quic-qpack-14, Internet Engineering Task Force (February 2020), https://datatracker.ietf.org/doc/html/draft-ietf-quic-qpack-14, Work in Progress
13. Mozilla: Neqo, an Implementation of QUIC written in Rust (2020), https://github.com/mozilla/neqo
14. NetApp: quant (2020), https://github.com/NTAP/quant
15. ngtcp2: ngtcp2 (2020), https://github.com/ngtcp2/ngtcp2
16. Pardue, L., Bradbury, R., Hurst, S.: Hypertext Transfer Protocol (HTTP) over multicast QUIC. Internet-Draft draft-pardue-quic-http-mcast-06, Internet Engineering Task Force (February 2020), https://datatracker.ietf.org/doc/html/draft-pardue-quic-http-mcast-06, Work in Progress
17. picoQUIC: picoQUIC (2020), https://github.com/private-octopus/picoquic
18. Postel, J.: Transmission Control Protocol. STD 7, RFC Editor (September 1981), https://www.rfc-editor.org/rfc/rfc793
19. Stenberg, D.: Curl's HTTP3 (and QUIC) support (2020), https://github.com/curl/curl/blob/faaa63f32359c5e7e91c02ad421105b3e2079d15/docs/HTTP3.md

20. Stenberg, D.: Curl's HTTP3 experimental support (2020), https://github.com/curl/curl/wiki/HTTP3
21. Thomson, M.: Version-Independent Properties of QUIC. Internet-Draft draft-ietf-quic-invariants-07, Internet Engineering Task Force (September 2019), https://datatracker.ietf.org/doc/html/draft-ietf-quic-invariants-07, Work in Progress
22. Thomson, M., Turner, S.: Using TLS to Secure QUIC. Internet-Draft draft-ietf-quic-tls-27, Internet Engineering Task Force (February 2020), https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-27, Work in Progress