Comparison of Different QUIC Implementations

Michael Kutter, Benedikt Jaeger*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: michael.kutter@tum.de, jaeger@net.in.tum.de

Abstract—**While the QUIC protocol was finalized by the IETF back in May 2021, the standard still leaves some design choices up to the developer. Especially for features like congestion and flow control, multiple streams, retransmission, packet size and 0-RTT, different approaches need to be considered. We give an overview of some of the considerations done by the developer and evaluate the performance of some implementations. We argue that future work needs to analyze the effect of the design choices on performance more in detail to find out which choice works best.**

Index Terms—**QUIC, implementation, design choices, performance**

1. Introduction

The QUIC protocol specifications were finalized on May 2021 after nearly five years of development [1]. It is built on top of UDP, which enables support for middleboxes, as no new transport layer protocol is defined. The goal of this protocol was to improve performance for HTTPS connections, while also achieving high security [2]. This is realized in multiple ways. QUIC exchanges cryptographic information during the connection establishment, thus reducing the round-trip times (RTT) and amount of packets during the initial handshake (1-RTT). When reconnecting to a server, it utilizes the already share keys to directly send data during the handshake (0-RTT). It uses connection IDs to identify connections after an IP address changed, thus allowing immediate reconnection to the server. To avoid the head-of-line blocking problem, QUIC uses multiple independent data streams.

During the five years of developing the specifications, different implementations have evolved. In this paper, we compare QUIC implementations and outline the different approaches they use. We focus on features which are up to the developer, like congestion and flow control, multiple streams, retransmission, packet size and 0-RTT [1] based on the study done by R. Marx et. al in [3]. We also try to compare the performance of some QUIC and TCP implementations based on the test results of the paper by A. Yu and T. A. Benson in [4].

In chapter 2 we list all the implementations we choose for this analysis. Chapter 3 then outlines all the different design choices considered by the developers. Afterwards in chapter 4, we evaluate the performance.

TABLE 1: QUIC implementations

Name	Developer	Language	Version
aioquic [5]	Jeremy Laine	Python	RFC 9000
lsquic [6]	LiteSpeed Technologies	C	RFC 9000
ngtcp2 [7]	Tatsuhiro Tsujikawa	C	RFC 9000
quic-go [8]	Lucas Clemente et. al	Go	RFC 9000
mvfst [9]	Facebook Inc.	C++	draft-29
picoquic [10]	Private Octopus Inc.	C	draft-34

2. List of Implementations

The QUIC implementations taken for analysis are shown in table 1.

3. Design Choices

When implementing the QUIC standard, different design choices can be considered. In the following sections we outline design choices made by the listed implementations.

3.1. Congestion Control

Sending packets as fast as possible can lead to overloading the network and result in routers dropping packets. These packets then need to be retransmitted, which leads to a longer transmission time. To avoid this, congestion control algorithms are used. These algorithms limit the number of inflight packets, by controlling the congestion window.

The QUIC standard defined by the IETF provides an exemplary congestion control algorithm which is similar to the TCP New Reno algorithm [11]. Therefore, it is up to the implementation side to choose the algorithm. The most used algorithms are New Reno, CUBIC and BBR. Compared to the implementation for TCP they slightly differ but the concepts stay the same.

New Reno: This algorithm is based on the Reno algorithm but improves during retransmission [12]. It begins with a “slow-start” phase, where it increases the congestion window by one for each acknowledged packet, resulting in exponential growth. Once multiple duplicate ACKs were received, or a packet was not acknowledged (retransmission timeout), it enters the “fast-recovery” phase. During this phase, it immediately retransmits the lost segments. When the retransmission was fully acknowledged it keeps the current congestion window. But if the retransmission was only partially acknowledged, it halves the current congestion window. After that it exits the "fast-recovery" phase and linearly increases the congestion window until the next packet was lost where it enters the "fast-recovery" phase again. This algorithm is a loss-based algorithm.

CUBIC: This algorithm is also a loss-based algorithm and also similar to the Reno algorithm [13]. It starts with the same "slow-start" phase. When a segment is lost it also enters the "fast-recovery" phase, where it retransmits the lost segments and halves the congestion window. The main difference is after the "fast-recovery" phase. Here, it uses a cubic function to increase the congestion window. In the beginning it increases the congestion window very slowly but increases it very fast later on. Compared to the Reno algorithm, it recovers faster from packet loss while not running into the next packet loss immediately. This algorithm is also the default congestion control algorithm in the Linux kernel for TCP.

BBR: This algorithm is called Bottleneck Bandwidth and Round-trip propagation time (BBR) and is a congestion-based algorithm developed by Google [14]. Compared to loss-based algorithm, it handles congestion based on the round-trip time. The algorithm tries to operate at the optimal point which is defined by the Bandwidth Delay Product $BDP = bandwidth \cdot RTT$. However, it is not possible to measure the bandwidth and the RTT at the same time, therefore estimated values are used [14]. The main advantage of this algorithm is that it does not fill the buffer of intermediate network nodes because this would lead to a bigger RTT.

Of the analyzed implementation New Reno is implemented by aioquic, ngtcp2, quic-go, mvfst and picoquic. CUBIC is implemented by lsquic, ngtcp2, quic-go mvfst and picoquic. BBR is implemented by lsquic, ngtcp2 mvfst and picoquic. We can see that a lot of implementations leave it to the user which algorithm to choose. This is escpecially beneficial for networks using different congestion control algorithms because some algorithms might outperform other algorithms which can lead to a sender barely sending any data due to a small congestion window [13].

3.2. Flow Control

While congestion control is about preventing the network from being overloaded, flow control is responsible for not overloading the receiver buffer. This is needed because the application might not be able to read data in the same speed the network delivers it, or the data was not received in the correct order. In TCP, each ACK packet provides the current receive window, which indicates the current available space in the receiver buffer [15]. Compared to TCP, QUIC allows multiple parallel data streams, therefore the QUIC protocol additionally applies flow control for each stream. The abstraction between stream level and connection level flow control is needed to prevent a single stream consuming the entire receivers buffer. This limitation is done through the MAX_STREAM_DATA (stream level) and the MAX_DATA (connection level) parameters [1]. Here, multiple approaches are possible to implement, and the following are most common.

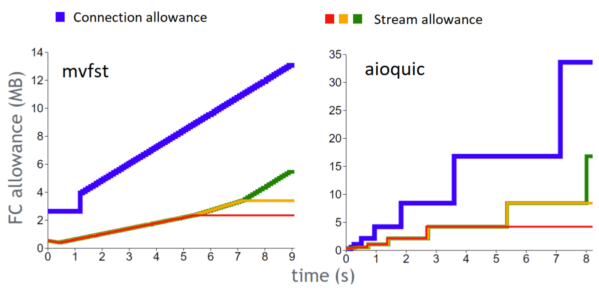

Figure 1: Flow control of mvfst and aioquic [3]

Static Allowance: With static allowance, the receive buffer has a fixed size, while the maximum allowance rate is increased linearly [3]. Typically, when the application has handled 50% of the received data in the buffer, the receive window is updated by adding the current buffer size. The downside of this method is that it can cause the sender to stop sending, when the updates of the receive window are delayed. This method is used by most of the analyzed implementations (lsquic, ngtcp2, quic-go, mvfst, picoquic), as it is easy to implement.

Growing Allowance: Growing allowance works similar to the static allowance method, but allows the receiver buffer to grow over time. This reduces the problem that this sender may stop sending, due to delayed receive window updates. Of the analyzed implementations, only aioquic used this method.

A detailed example of these approaches can be found in Figure 1 with mvfst (static allowance) and aioquic (growing allowance).

Regardless of the above mentioned methods, the most important aspect in regards to performance, is the size and the frequency of the receive window updates, as too small receive windows or too few updates can lead to a stalling sender. Flow control in QUIC remains an open issue, therefore, further study is required to identify the best possible approach.

3.3. Multiple Streams

TCP offers a single reliable in-order stream to transmit data. When transmitting independent resources, this is vulnerable to head-of-line blocking. This occurs when a single resource prevents other resources from being received, which happens when TCP loses a packet. The QUIC protocol defines multiple data streams to get around this problem [1]. In order to handle and multiplex these streams, a scheduler is required. Therefore, two approaches can be used to divide the bandwidth between the resources.

Sequential: When using a sequential scheduler, all data of one stream is send first before sending data of another stream. This is expected to work best for loading Web pages, as the application can prioritize which data should be sent first. However, this can lead to head-of-line blocking again e. g. when the data of a single stream is too big. Aioquic, ngtcp2 and picoquic are using this scheduler approach.

Round-Robin: When using a Round-Robin scheduler, the bandwidth is equally distributed between all resources.

However it is important to avoid sending data of multiple different streams inside one packet because this could lead to head-of-line blocking again. The typical approach is to send a few packets of data of the same stream before switching to another stream. The downside of this approach is that it can take longer to receive all data of a single stream. lsquic, quic-go and mvfst are using this approach.

We can see that stream multiplexing can have a significant impact on performance. Therefore, the QUIC standard additionally requires the implementations to have a prioritization system, which the application can use to prioritize streams [1]. With this system, the impact of the scheduler approach becomes less important, as higher priority streams will be sent first. However, this is only relevant if the application supports prioritization of resources. We think that the round-robin scheduler is a more general approach because it avoids head-of-line blocking regardless of the data size and it could also simulate a sequential scheduler when using the prioritization system.

3.4. Packet Size

The QUIC protocol requires a minimum UDP payload size of 1200 bytes, but to further improve throughput, a bigger packet size is required [1]. When using bigger packets, the chance of dropping a packet, due to an intermediate network node not supporting this size, highly increases. Therefore, it is recommended to either use Path MTU Discovery (PMTUD) or Data Packetization Layer Path MTU Discovery (DPLMTUD)[1], to find out the maximum packet size supported by all network nodes [1]. These methods works by sending a large packet to the destination. If the corresponding ICMP error message is received, we know that the MTU was too large and and intermediate network node dropped the packet. Therefore, we repeat the process by reducing the packet size until successful transmission. This feature is currently only supported by lsquic, quic-go and picoquic. The other implementations use a fixed packet size. Therefore, they are not allow to exceed the minimum UDP payload of 1200 bytes to ensure compatibility of intermediate network nodes.

3.5. Client Validation of 0-RTT

A new feature in the QUIC protocol is the 0-RTT connection establishment, where it is possible to send data before receiving any response from the server. This is made possible by reusing the preshared encryption keys of the session ticket, which were negotiated in the first connection. This feature is vulnerable to replay attacks and amplification attacks. Therefore, the QUIC protocol specifies that the server is not allowed to send back more than three times the data it received from the client until the address of the client is validated [1]. The validation can be done in two ways.

Approach 1: After the client requested data during the 0-RTT connection establishment, the server answers

1. https://blog.litespeedtech.com/2020/10/19/improve-performance-with-dplpmtud/

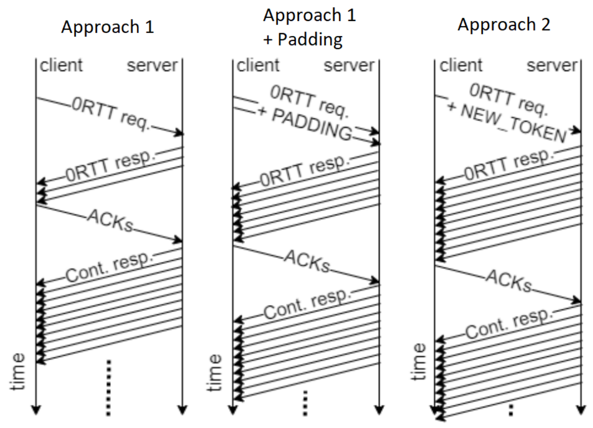

Figure 2: Client validation of 0-RTT [3]

directly within the 3x limit. It then waits until the acknowledgment of the client. When the acknowledgment was received the address can be considered validated and the server can send the rest of the data. The disadvantage of this method is that the server can only respond in the beginning with small packets inside the 3x limit. However, it is possible to increase this limit by adding some padding to the initial data request. This approach is currently used by aioquic and ngtcp2.

Approach 2: During the first connection establishment (1-RTT), the server sends an encrypted NEW_TOKEN frame to the client, which can be used by the client during the 0-RTT connection establishment to validate its address. The server then can ignore the 3x limit and directly send large amounts of data to the client, if the token matches. This is done by lsquic, quic-go, mvfst and picoquic.

A detailed diagram about these approaches can be found in Figure 2.

4. Performance

A. Yu and T. A. Benson measured the performance of different QUIC implementations and compared them to different TCP implementations in their paper "Dissecting Performance of Production QUIC" [4]. Compared to most other papers, they focused more on already deployed implementations, instead of testing it on a local setup. This approach in benchmarking resembles a more real world scenario. In their analysis, they also tried to diffentiate, if the results were due to the protocol specifications or due to the design of the implementation.

For their benchmarking, they use public available endpoints by Google, Facebook and Cloudflare on the server side. On the client side they choose to use cURL, Google Chrome, Facebook Proxygen and ngtcp2. All these implementations are using the HTTP/2 (H2) stack for TCP and HTTP/3 (H3) stack for QUIC. With the Network Link Conditioner, they simulate different network conditions. It is also worth mentioning, that they setup the flow control mechanism to not have any impact on the performance.

When transmitting a single resource, we and the authors excpect similar results between QUIC and TCP. For a small file size the QUIC implementations did outperform the TCP implementations. This is due to the improved handshake of the QUIC protocol. For larger files the impact of the improved handshake minimizes. Consequently, the performance between all the implementations are similar. However, when adding packet loss to the network, the Cloudflare H3 endpoint worsens compared to H2. The authors identified that this is due to different congestion control algorithms between H3 (CUBIC) and H2 (BBR). The other endpoint which stands out is Facebook. Their H3 endpoint performed significantly worse when adding extra delay. It was identified that this is due to a bug in the congestion control algorithm. We can see that the choice of the congestion control algorithm can have a huge impact on the performance as already outlined in chapter 3.1.

When transmitting multiple resources, we and the authors expect QUIC to perform better as TCP, due to QUIC's protocol design with the introduction of multiple data streams. However, the results were similar compared to transmitting a single resource. For small files, the H3 endpoints performed better, which could be traced back to QUIC's handshake design again, and for larger files the performance was similar. The only exception was the Cloudflare endpoint. Here, H3 also outperformed H2 for larger resources. The authors traced this issue back to different application configurations which favored the H3 implementation. The authors also analyzed the effect of the different scheduling approaches. Cloudflare is using a sequential scheduler, while Facebook and Google are using a round-robin scheduler. Here, the different scheduler did not have any effect on the performance due to the prioritization system by the H3 stack. This is also the same, which we concluded in section 3.3.

The authors concluded that most performance discrepancies are a result of the developers design or the operators configuration. These results can also be verfied by other papers [16], [17].

5. Conclusion and future work

In this work, we have discussed multiple design choices which needs to be considererd when implementing the QUIC protocol. We saw that there are multiple different approaches to implement congestion and flow control, multiple streams, packet size and client validation of 0-RTT. While not all aspects will have high impact on the performance and some might be application dependent, we concluded that further research is needed to find out which approach works best in practice.

We also analyzed the tests performed by A. Yu and T. A. Benson, where they compared the performance of different QUIC implementations with TCP over public available endpoints. We saw that in gerneral the QUIC implementations had the advantage when transmitting small resources due to the improved handshake design of the protocol. For larger files, the performance balances out because the impact of the handshake minimizes. However, we saw some discrepancies to this behavior. Most of these performance differences could be traced back to the developers design of the implementations or the configuration of the operators.

We feel that future analysis is needed to compare the performance of the different implementations. It is espacially important to focus on which design choices impacts the performance of the protocol.

References

[1] J. Iyengar and M. Thomson, "Rfc 9000," https://datatracker.ietf.org/doc/rfc9000/, May-2021, [Online; accessed 26-February-2022].

[2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Sweet, 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," August-2017.

[3] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity," August-2020.

[4] A. Yu and T. A. Benson, "Dissecting Performance of Production QUIC," April-2021.

[5] J. Laine, "aioquic," https://github.com/aiortc/aioquic, [Online; accessed 26-February-2022].

[6] L. Technologies, "Litespeed quic (lsquic) library," https://github.com/litespeedtech/lsquic, [Online; accessed 26-February-2022].

[7] T. Tsujikawa, "ngtcp2," https://github.com/ngtcp2/ngtcp2, [Online; accessed 26-February-2022].

[8] "A quic implementation in pure go," https://github.com/lucas-clemente/quic-go, [Online; accessed 26-February-2022].

[9] Facebook, "mvfst," https://github.com/facebookincubator/mvfst, [Online; accessed 26-February-2022].

[10] "picoquic," https://github.com/private-octopus/picoquic, [Online; accessed 26-February-2022].

[11] J. Iyengar and I. Swett, "Rfc 9002," https://datatracker.ietf.org/doc/rfc9002/, May-2021, [Online; accessed 26-February-2022].

[12] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida, "Rfc 6582," https://datatracker.ietf.org/doc/html/rfc6582, April-2021, [Online; accessed 26-February-2022].

[13] M. Geist and B. Jaeger, "Overview of TCP Congestion Control Algorithms," May-2019.

[14] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR Congestion-Based Congestion Control," Septemper-2016.

[15] I. S. I. U. of Southern California, "Rfc 793," https://datatracker.ietf.org/doc/html/rfc793, Septemper-1981, [Online; accessed 26-February-2022].

[16] S. Endres, J. Deutschmann, K.-S. Hielscher, and R. German, "Performance of QUIC Implementations Over Geostationary Satellite Links," Feburary-2022.

[17] M. Moulay, F. D. Munoz, and V. Mancuso, "On the Experimental Assessment of QUIC and Congestion Control Schemes in Cellular Networks," June-2021.