Comparison of Different QUIC Implementations

Salim Hertelli, Benedikt Jaeger*, Johannes Zirngibl*
*Chair of Network Architectures and Services, Department of Informatics
Technical University of Munich, Germany
Email: hertelli@in.tum.de, jaeger@net.in.tum.de, zirngibl@net.in.tum.de

Abstract—**QUIC is an encrypted and multiplexed transport protocol developed by Google and deployed on their servers in 2012. QUIC aims to replace the commonly used TCP/TLS stack. It was standardized on the 27th of May 2021 in the RFC 9000 [1], which defines the core and specifications of the protocol. QUIC is designed to be implemented in the user space. This allows different implementations to exist in multiple programming languages and with different features. This paper aims to give an overview of existing implementations and to compare them based on different metrics, such as the used programming language, supported versions of QUIC, handshake encryption method, and used congestion control algorithm.**

Index Terms—**quic, transport protocol, http/3, quic implementaions**

1. Introduction

QUIC is a connection-oriented, encrypted, and multiplexed transport protocol built on UDP as shown in Figure 1. It was developed and deployed by Google in 2012 as a replacement for the traditionally used TCP/TLS stack, commonly used in the HTTPS stack. QUIC aims

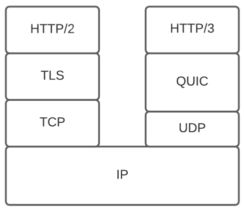

Figure 1: QUIC stack vs TCP/TLS stack (adapted from [2]).

to improve on the pre-existing protocols by enhancing security and reducing latency. It was standardized by the IETF[1] with the release of RFC 9000 in May 2021, which was complemented by three more documents, namely RFC 8999, 9001, and 9002.

As part of its design goal, QUIC is implemented in the user space. This property of QUIC leads to faster development and deployment cycles as it avoids the long process of pushing system-wide updates [2]. This also allowes for different congestion avoidance algorithms to be dynamically used, which makes it easier to perform experimentation using various congestion control algorithms, fix bugs, and deploy changes. Since then multiple implementations have emerged in different programming languages and paradigms, including functional programming languages such as Haskell.

In this paper, we list some of the available QUIC implementations. Differences between them will be analyzed based on various metrics and criteria, such as the status of the projects, how well-maintained they are, etc. Section 2 introduces some of the specifications of QUIC. In Section 3 we discuss the methodology we used to collect data about different projects. We then display the results in Section 4. In Section 5 we highlight other research that was conducted on QUIC. We then conclude the paper in Section 6.

2. Background

QUIC was started as an experimental protocol by Google back in 2012 to replace the existing TCP/TLS stack used for HTTP. By being developed to be deployed in the user space and not in the kernel of operating systems, it allows for faster development cycles. It does also allow for critical updates to be pushed and applied faster and gives developers much more room to experiment with new features and improvements [2].

In 2015 a draft of QUIC was submitted and a working group was created at the IETF with the goal of standardizing QUIC. This standardization came on May 27th, 2021 with the release of RFC 9000. It specifies the core of the QUIC protocol and serves as a certification for QUIC's reliability [1].

As detailed in the RFC 9000 [1], QUIC supports flow streams and network path migration among other features. In addition, QUIC allows multiple streams to be multiplexed, which prevents head-of-line blocking. This leads to a reduced latency overall compared to TCP, as the costs of using multiple TCP connections can be mitigated. Handshake delays are improved as well by removing unnecessary round trips. This allows exchanges to occur as early as possible or even immediately, leading to 0-RTT handshakes in some cases [2]. QUIC also improves congestion control [2]. RFC 9000 does not specify a particular algorithm to be used, this allows researchers to experiment and improve on it.

QUIC fully encrypts and authenticates handshakes. This holds for almost all the handshakes except some of the early ones. Encryption covers the majority of a QUIC package. The unencrypted parts are needed for

1. Internet Engineering Task Force

routing purposes such as connection ID and version number among others. In addition, the authentication process ensures that packets that have been tampered with can be discovered, which leads to a connection failure [2].

As mentioned previously, QUIC has many different implementations. Different implementations may support different versions of QUIC depending on the starting time of the project. At the time of writing, as it is just a couple of months after the standardization of QUIC, not all implementations support QUIC version 1. It is however prone to change with time.

3. Methodology

In this section, we discuss the process, tools, and methodology used to collect data about different QUIC implementations. As stated earlier, there exists various QUIC projects in many programming languages [3]. The choice of which implementation to select for this paper was then done based on different criteria including project age, the status of the implementation, and whether the project is being backed up by a big company. Chromium-quic was the first selected implementation due to the fact that Google was the company to start the development of QUIC and because it powers one of the most prominent browsers, namely Google Chrome. The next selected implementations are Quiche, Mvfst, and Msquic which are backed by Cloudflare, Facebook, and Microsoft respectively. Quic-go was then selected based on the debuting time of the project, which started in early 2016. The last two implementations are Aioquic and Haskell-quic, developed respectively in Python and Haskell. This was done to show diversity in the used programming languages for QUIC projects.

The next step was to collect data about the pre-selected projects. Project-specific properties were collected in an automated process. Many of the selected projects are hosted on GitHub. This allowed the usage of the GitHub API, which can be queried using scripts to collect the project creation date, number of pull requests, and the number of commits. It does also provide information about the main programming language of a project. However, this may lack a certain level of accuracy, considering the large number of commented lines in big coding projects. To collect exact information about the utilized programming languages, Cloc was used [4]. Cloc is an open-source software written in Perl used to count lines of codes in a directory. It does then present an overview of the results, including, but not limited to the used programming languages based on file extensions and how many lines of code each file contains. Moreover, it separates the commented lines from the actual code lines as well.

4. Evaluation

In this section, we display the collected results about the different QUIC implementations. The collected data is summarized in TABLE 1 and TABLE 2 and will be further discussed in this section.

4.1. Chromium-quic "Quiche"

Chromium-quic is a QUIC implementation and part of the Chromium projects, which are open-source projects developed by Google. The project is called Quiche as an acronym for "QUIC, HTTP/2, Etc". It is a production-ready implementation, written exclusively in C and C++. It powers parts of the "Google" search engine and the "YouTube" video playing service. The project is hosted on Google's servers [5] as well as on GitHub and is kept in sync. The code is well maintained and documented with an extensive wiki and many supporting documents, such as the RFCs. The project did go through a lot of QUIC versions, starting from Q403 until draft-29. It supports CUBIC for congestion control and uses QUIC-Crypto as well as TLS to encrypt the packets being transmitted [3].

4.2. Cloudflare "Quiche"

Quiche[2] is another implementation of the QUIC protocol written in Rust and hosted on GitHub [6]. It was developed by Cloudflare, a web security and infrastructure company, in order to enable HTTP/3 support on their servers. Documentation for the project does provide a guide on how to build and configure Quiche to receive, send and handle packages. The wiki also has a listing of the used structures, enum, and functions used in the implementation as well as short descriptions to help developers understand their functionality.

Quiche supports the usage of two different congestion control algorithms namely Reno and CUBIC as well as a high-level API in order to configure the used algorithm. CUBIC is deployed on Cloudflare's production environment. Later came the introduction of HyStart++ to improve congestion control. HyStart++ is a modification of the slow start phase in congestion control algorithms, which tries to improve the performance by reducing packet loss and preventing the overshooting of the ideal sending rate. This is done by introducing the Limited Slow Start phase (LSS). Before reaching the congestion threshold in the Slow Start phase, the congestion control algorithm switches to LSS. During the LSS phase, the congestion window grows slower than in the congestion avoidance phase. Upon reaching the congestion threshold, the algorithm switches then to the congestion avoidance phase [7].

4.3. Mvfst

Mvfst is an implementation developed by Facebook. It is mainly written in C/C++ and hosted on GitHub [8]. More than 75 percent of all the network traffic of Facebook is happening on QUIC and HTTP/3. This includes their social networking websites Facebook and Instagram. Mvfst makes use of Facebook's own TLS 1.3 implementation "Fizz" to ensure the security of packet exchanges. The implementation comes with a wiki that explains how to build, run and test Mvfst. It does also provide samples for both client and server side. The wiki does not mention, which congestion control algorithm Mvfast uses. However, the header files and the implementation for both NewReno and CUBIC are present in the source code [9].

Moreover, Facebook does experiment with artificial intelligence based congestion control algorithms. Mvfst-rl

2. Not to be confused with the Chromium QUIC implementation which is also called Quiche.

TABLE 1: Listing of different QUIC implementations and metrics about the status of the projects (September 2021)

Project	Language	License	Creation date	LoC	#pull requests	average/day	#commits	average/day
Quic-go	Go	MIT	06/04/2016	61119 Go	1843	0.92/day	678	1.82/day
Chromium-quic "Quiche"	C / C++	BSD-3-Clause	-	228 C 211574 C++ 40881 C/C++ headers	-	-/day	-	-/day
Cloudflare "Quiche"	Rust / C	BSD-2-Clause	29/09/2018	36920 Rust 1300 C	715	0.65/day	294	0.79/day
Aioquic	Python	BSD-3-Clause	05/02/2019	17337 Python	115	0.11/day	87	0.20/day
Mvfst	C / C++ Python / Rust	MIT	10/04/2018	75760 C++ 15927 C/C++ headers 6537 Python 1150 Rust	124	0.09/day	1,317	3.58/day
Msquic	C / C++	MIT	26/10/2019	56291 C 27155 C++ 39296 C/C++ headers	1,529	2.18/day	756	2.07/day
Haskell-quic	Haskell / C	BSD-3-Clause	11/01/2019	9741 Haskell 5503 C	10	0.01/day	481	1.30/day

TABLE 2: Different QUIC implementations and their used versions, handshake encryption methods and congestion control algorithms (September 2021)

Project	Versions	Roles	Handshake	Congestion Control
Quic-go	as the current draft	library, client, server	TLS 1.3 RFC	NewReno, CUBIC
Chromium-quic "Quiche"	Q043, Q046, Q050, T050 T051, draft-27, draft-29	library, client, server	QUIC Crypto, TLS	CUBIC
Cloudflare "Quiche"	draft-27, draft-28, draft-29	library, client, server	TLSv1.3 (RFC8446)	NewReno CUBIC + HyStart++
Aioquic	draft-29, version 1	library, client, server	TLS 1.3	NewReno
Mvfst	draft-29	library, client, server	TLS 1.3	NewReno, CUBIC Mvfst-rl (experimental)
Msquic	draft-29, version 1	client, server	TLS 1.3 RFC	CUBIC
Haskell-quic	draft-29	library, client, server	TLS 1.3	NewReno

[10] is a framework that uses asynchronous reinforcement learning training in order to improve congestion control in QUIC and is built on their own implementation. However, it is still an experimental feature and is yet not ready to be deployed in a production environment.

4.4. Msquic

Msquic is an IETF QUIC implementation written in C and C++. The project was started by Microsoft and is hosted on GitHub [11]. Currently, it supports both Draft-29 and version 1 of QUIC. Msquic uses TLS 1.3 to encrypt and authenticate all handshakes and packets. As for congestion control, it supports CUBIC. The project comes with extensive documentation to build, test and deploy Msquic. Microsoft does also provide daily benchmark results including single connection upload and download rates and the average number of requests completed per second. Msquic is a production-ready implementation. It powers Microsoft's HTTP/3 stack and is deployed in other products to handle QUIC connections [12].

4.5. Quic-go

Quic-go is a QUIC implementation in the Go programming language and is hosted on GitHub [13]. Currently, it does implement the IETF draft-29. The documentation in their GitHub repository indicates however that the support for draft-29 will eventually be replaced by a more recent standard. Quic-go does support both NewReno and CUBIC for congestion control and uses TLS 1.3 to encrypt handshakes and packages. The implementation comes with examples for client and server instances as well as instructions on how to run tests.

4.6. Aioquic

Aioquic is an IETF QUIC implementation in Python. It is an open-source project hosted on GitHub [14]. The implementation is built on Asyncio, which is a standard Python framework for asynchronous I/O. Aioquic is implemented to be conforming with the RFC 9000. It does support a minimal TLS 1.3 implementation for packages and handshake encryption and it uses NewReno for congestion control, as recommended by the RFC 9000, which features pseudocode for NewReno. Aioquic's wiki explains how to test the implementations on different operating systems including Windows, Linux, and MacOS. There are also different examples, which can be used to test different QUIC use cases.

4.7. Haskell-Quic

Haskell-quic is an implementation of IETF QUIC in Haskell and is hosted on GitHub [15]. The main difference of this implementation compared to the previous ones is that Haskell is a functional programming language. This makes supporting new features harder, as they are generally described in an imperative style pseudocode. The pseudocode needs then to be transfered to functional style in order to implement it in a functional language like Haskell. Kazuhiko Yamamoto, the main developer behind Haskell-quic, claims that this transformation is not a simple task to perform [16].

Haskell-quic is based on Haskell's lightweight threads and uses TLS to secure handshakes. The author does maintain a blog [17] to track the development of haskell-quic where he discusses important milestones and future features to be implemented. Haskell-quic uses NewReno for congestion control as recommended by the RFC 9000.

5. Related work

The IETF QUIC Working Group maintains a listing [3] in order to track known implementations. The listing also keeps track of the used programming languages, versions that are implemented, handshakes encryption methods, and, if available, some links to public servers that use or allow experimentation to be conducted with the corresponding implementation.

Marx et al. [18] compare different QUIC implementations and discuss their behavioral heterogeneity. They also discuss more behavioral aspects than those mentioned in this paper, including multiplexing scheduling and the 0 RTT approach used. In addition, they apply different methodologies by using the qvis and qlog tools.

Interoperability tests play an important role in internet protocols development, including QUIC. Interoperability tests check whether different independently developed implementations interact with each other as expected. Marten Seemann and Jana Lyengar describe it as a crucial tool to expose weakness and ambiguities in the specifications of QUIC. Moreover, the IETF recommends testing to be a part of the development process [19]. One interoperability testing framework is QUIC Interop Runner or QIR. QIR performs different tests between all pairs of their listed implementations. This includes testing the server as well as the client functionalities on different scenarios such as handshake loss and version negotiation among others [20].

6. Conclusion

In this paper, we discussed QUIC, a transport protocol intended as a replacement for the TCP/TLS stack and implemented in the user space. We have shown 7 different QUIC implementations and presented different properties ranging from the status of the project, documentation, supported QUIC versions, and used congestion control algorithms. We utilized an almost automated process to collect data about various projects using different APIs and project analyzing software.

Although all the mentioned implementations follow the IETF specifications, our results show a large diversity in the used programming languages and supported features. In particular, the used congestion control algorithms differ from one project to another as no particular algorithm was specified by the IETF. However, we can notice that CUBIC and NewReno are almost always used in production. Supported versions of QUIC are also not the same across different implementations. This can however be linked to the age of the project. As the QUIC standard was published just a few months before the time of writing of this paper, not all implementations are currently supporting version 1 of QUIC. However, it is already supported by some of them, including Aioquic and Msquic.

QUIC is still relatively new compared to TLS. However, it is remarkable to see that some implementations are already production-ready and being used by different mainstream services that are accessed by millions daily. This protocol as well as its development process will play an important role in defining norms for creating new internet protocols in the future as QUIC is getting more popular and is beginning to be universally adopted.

References

[1] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc9000.txt

[2] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. B. Krasic, C. Shi, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. C. Dorfman, J. Roskind, J. Kulik, P. G. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, and W.-T. Chang, "The QUIC Transport Protocol: Design and Internet-Scale Deployment," 2017.

[3] IETF QUIC Working Group, "QUIC Implementations," available at https://github.com/quicwg/base-drafts/wiki/Implementations, [Online. accessed September-2021].

[4] AlDanial, "Cloc," available at https://github.com/AlDanial/cloc, [Online. accessed September-2021].

[5] Google, "Quiche," available at https://quiche.googlesource.com/quiche/, [Online. accessed September-2021].

[6] Cloudflare, "Quiche," available at https://github.com/cloudflare/quiche, [Online. accessed September-2021].

[7] J. Choi, "CUBIC and HyStart++ Support in quiche," available at https://blog.cloudflare.com/cubic-and-hystart-support-in-quiche/, [Online. posted on 20-August-2020].

[8] Facebook, "mvfst," available at https://github.com/facebookincubator/mvfst, [Online. accessed September-2021].

[9] F. Engineering, "How Facebook is bringing QUIC to billions," available at https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billions/, [Online. posted on 21-October-2020].

[10] Facebook, "mvfst-rl," available at https://github.com/facebookresearch/mvfst-rl, [Online. accessed September-2021].

[11] Microsoft, "msquic," available at https://github.com/microsoft/msquic, [Online. accessed September-2021].

[12] D. Havey, "MsQuic is Open Source," available at https://techcommunity.microsoft.com/t5/networking-blog/msquic-is-open-source/ba-p/1345441, [Online. posted on 28-April-2020].

[13] L. Clemente, "Quic-go," available at https://github.com/lucas-clemente/quic-go, [Online. accessed September-2021].

[14] "Aioquic," available at https://github.com/aiortc/aioquic, [Online. accessed September-2021].

[15] K. Yamamoto, "Haskell-quic," available at https://github.com/kazu-yamamoto/quic, [Online. accessed September-2021].

[16] ——, "Developing QUIC Loss Detection and Congestion Control in Haskell," available at https://kazu-yamamoto.hatenablog.jp/entry/2020/09/15/121613, [Online. posted on 15-September-2020].

[17] ——, "The Current Plan for Haskell QUIC," available at https://kazu-yamamoto.hatenablog.jp/entry/2020/10/23/141648, [Online. posted on 21-October-2020].

[18] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same Standards, Different Decisions: A Study of QUIC and HTTP/3 Implementation Diversity," 08 2020.

[19] M. Seemann and J. Iyengar, "Automating QUIC Interoperability Testing," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 8–13. [Online]. Available: https://doi.org/10.1145/3405796.3405826

[20] M. Seemann, "QUIC Interop Runner," available at https://github.com/marten-seemann/quic-interop-runner, [Online. accessed November-2021].