MsQuic – A High-speed QUIC Implementation

Manuel Bünstorf, Benedikt Jaeger*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: m.buenstorf@tum.de, jaeger@net.in.tum.de

***Abstract*—The QUIC protocol is designed as a more flexible alternative for the TCP / TLS stack and is implemented in user-space. Thus several implementations exist, each with its own strengths and weaknesses. In this paper, we take a close look at MsQuic, a high-speed QUIC implementation by Microsoft. We give an overview of the library, showcase its strengths, and investigate in what projects it is already deployed. Also, we evaluate MsQuic regarding performance on different hardware architectures. Comparing the goodput of MsQuic with other QUIC implementations, we found that MsQuic outperforms them all, with the largest difference of goodput on older hardware.**

***Index Terms*—quic, msquic, goodput, performance, high-speed, transport, measurements**

1. Introduction

QUIC is a new transport protocol designed to replace the existing TCP / TLS protocol stack. It was originally developed by Google [1] and is now standardized by the Internet Engineering Task Force [2]. It improves performance and latency, by eliminating redundant handshakes, streamlining the handshake procedure. It is built on top of UDP and runs in user-space. Many different QUIC implementations exist as a result of being built in user-space, each having its strengths and weaknesses. They differ in their implementation language and in their design. This includes congestion and flow control, packet size, retransmission handling, and more. One of these implementations is MsQuic by Microsoft [3]. It is open source and designed to be a high-performance, general purpose and cross-platform implementation of the QUIC protocol. MsQuic is deployed in Windows [4] and other applications. In this paper, we take a look at QUIC, how it works on a basic level, and what improvements it makes over the existing TCP and TLS networking stack. We look at MsQuic regarding the team behind it, its goals and design as well as the process behind developing a QUIC library with a focus on performance. We showcase the high-level design of the library and shortly introduce its API. We also show results of our measurements on MsQuic and competing implementations, testing their claims of being optimized for maximal throughput.

The remaining part of the paper is structured as follows. Firstly, we introduce QUIC and explain important parts of its design. Then we showcase MsQuic, talking about its goals, high-level architecture and its development. Following this, we present our measurements regarding MsQuic and competing implementations. Finally, we summarize related work that gives more context to the performance measurements, before we conclude the paper.

2. Background

QUIC is intended as a next-generation transport protocol that powers the new HTTP / 3 [5]. It replaces TCP / TLS, taking care of flow control. It also handles stream multiplexing, which was previously handled by the HTTP layer. QUIC was originally developed by Google as a successor to SPDY, their first attempt at designing a next-generation transport protocol. Taking the lessons learned from SPDY, they started development on QUIC, which was standardized in 2021 by the IETF as RFC 9000 [2].

2.1. Rapid deployment

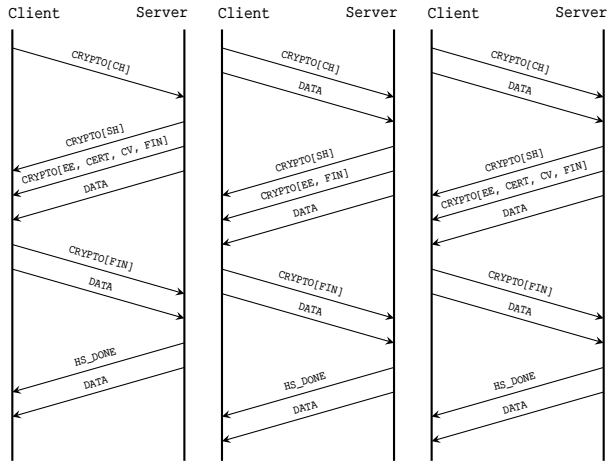
One of the main goals when designing QUIC was to make rapid deployment possible, which also fostered the general development process by Google. To achieve this they made two key choices for the design of the protocol [1]. The first one was to build QUIC in user-space rather than kernel-space, allowing deployment as part of applications like web browsers[1], in contrast to slow-moving kernels of operating systems. The other decision was to use the existing UDP protocol as a lightweight layer underneath QUIC, leveraging existing network infrastructure that already supports UDP traffic. That eliminated the need to make any changes to middleboxes for them to support the new protocol. These two decisions made it possible for Google to roll QUIC out to users as part of their products very quickly. It also gave them the opportunity to iterate on their designs in the real world, having the tools to A / B test changes with real-time feedback and monitoring.

2.2. Performance improvements

Another goal of QUIC was to reduce handshake latency and enhance performance over the existing TCP / TLS stack. First and foremost by reducing the amount of network round trips that are required for a TCP and a TLS handshake. It takes three round trips for a TCP / TLS handshake between a client and server before the client can send the actual request to the server. QUIC combines the handshakes for the transport layer

1. Google used their browser Chrome to roll out the QUIC protocol to their users.

 doi: 10.2313/NET-2023-11-1_01



(a) 1-RTT (b) 0-RTT (c) Rejected 0-RTT

Figure 1: QUIC handshake timelines

and encryption into one. With that alone, QUIC is able to alleviate the overhead needed for setting up a connection [1].

Establishing a connection between a client and server with QUIC is done in two different ways, as shown in Figure 1. An initial connection of a client that has not connected to the server before is depicted in Figure 1a. The client sends an incomplete `Client Hello` to the server, prompting a `Server Hello` from the server, together with cryptographic data like the certificate, which the client needs to establish an encrypted and trustable connection. At this point the server can already send encrypted `DATA` to the client. With the cryptographic data from the server, the client sends a complete `Client Hello` and encrypted `DATA` can be sent. The server responds with a `Handshake Done` message, indicating that a secure connection was established, and from there the connection to the client is established. This results in only one network round trip (1 RTT) of delay to establish a new connection, only a third of the overhead TCP/TLS has. After this initial connection, the client can cache the server configuration as well as the cryptographic data to use in later connections with the same server. As illustrated in Figure 1b, the client can avoid sending an incomplete `Client Hello` and is able to use the cached data to immediately send a complete `Client Hello` as well as `DATA`, allowing for a connection without any overhead. Lastly, the data cached by the client might become invalid after some time, for example because the server changed its configuration. When the client tries to initiate a new connection with an invalid encryption, the server will respond with the new cryptographic data. At this point the client will proceed with the regular 1-RTT procedure for connection establishment as seen in Figure 1c.

3. MsQuic

MsQuic is the open source implementation of the QUIC protocol by Microsoft [3]. Its initial release was on the 27th of November 2019. It was developed by the datapath and transports team at Microsoft, headed by Nicholas Banks [6]. He is the primary QUIC architect and developer and is responsible for the project. Later, the team was split into a dedicated datapath team and transports team, the latter of which now continues the development of the library. The code for MsQuic was open sourced on the 28th of April 2020 on GitHub and is licensed under a MIT License [3] that provides complete freedom over the code, allowing unconfined commercial use, modification and distribution. This, however, is without any liability or warranty.

Commercial software exists that already uses MsQuic. For example, it is already used as the kernel mode *msquic.sys* in Windows 11 and Windows Server 2022 and is used by the Windows kernel mode *http.sys* for the purpose of providing HTTP/3 capabilities [7]. The SMB protocol offers the usage of QUIC as an alternative to TCP, by using MsQuic under the hood [8]. The .NET Core ecosystem uses MsQuic to provide QUIC functionality as well [4]. The Microsoft Game Development Kit, which provides the possibility of using QUIC as a transport protocol to implement latency critical network code in games, also using MsQuic as the implementation of choice [9].

3.1. Goals

When Banks and his team started development on MsQuic their main focus was to build a general purpose QUIC implementation that can be used by other products from Microsoft to leverage QUIC as a next-generation transport protocol [4]. The two main components that wanted to use MsQuic right from the beginning were the HTTP kernel mode of Windows and the SMB protocol, both of which run in the Windows kernel. This meant that MsQuic needed to run in the kernel as well, in order to be used by both HTTP and SMB in Windows. A module inside of the Windows kernel has a few constraints, primarily regarding the implementation language. Kernel code for Windows has to be written in C or in a restricted form of C++. The other constraint is having to implement an asynchronous network IO model.

Although both HTTP and SMB were two main use cases for MsQuic from the very beginning, they had different requirements [4]. HTTP requires a high number of requests per second as well as low latency for connections. SMB on the other hand benefits from high throughput on a single connection. This meant MsQuic needed to be optimized for both cases, handling as many requests per second as possible while still enabling bulk transfer on single connections. A bit later into development .NET core team at Microsoft was interested in using their in-house QUIC implementation to provide QUIC functionality for the .NET ecosystem. Since .NET is a cross platform framework, this meant that the goals for MsQuic now also included providing cross platform support, for both Windows and Linux as well as Xbox.

3.2. Design

On a high level, MsQuic is split into two parts [4]. The core protocol that implements all of the platform independent protocol logic, as well as a platform specific abstraction layer that handles the interaction with constructs tied to the OS like sockets and threads. An

 doi: 10.2313/NET-2023-11-1_01

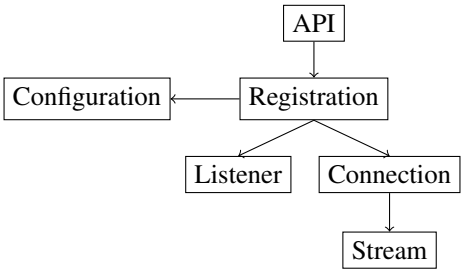


Figure 2: Object model of the API

asynchronous IO model is used for the data path, utilizing asynchronous calls into lower layers to send data and using upcalls back to the higher layer to complete these sends. To improve performance IO operations are batched between the Application / MsQuic layers and between the MsQuic / UDP layers. This batching works together with UDP send segmentation and receive coalescing [4].

The threading model of the library is comprised of two different types of threads, data path threads and core threads [4]. Data path threads handle UDP receive calls and basic QUIC validation, before queuing each packet to its associated connection object. The interaction of the data path threads with the UDP layer is platform dependent. Core threads process the queues of every connection object, performing the majority of processing. This division of the handlers for packet receives and the packet processing allows for a practically lock free execution of the code. It also makes horizontal and independent scalability possible, where both a high amount of requests and bulk throughput can be prioritized individually.

3.3. Prioritizing performance

One of the primary concerns when developing MsQuic is performance [3]. As a general purpose implementation, however, performance can mean different things. For HTTP, a high number of requests per second is crucial, while for file sharing high throughput is important. In order to provide performance in all scenarios, they have worked to standardize a measure for performance and created a process for testing the performance of their QUIC implementation in different scenarios [10]. These measures consist of single connection upload and download speeds, requests per second and handshakes per second. Testing for these scenarios is done automatically for every merge and pull request to the main branch. This provides real time feedback of changes in performance that can be tracked back to a certain commit. Thresholds are in place to automatically reject any pull requests that do not meet specified performance requirements. All of these metrics are publicly available on the MsQuic performance dashboard [11]. This commitment by the MsQuic team to keep track of performance changes on every commit has helped them a lot to achieve their goal of building a high-performance QUIC implementation.

3.4. API overview

The library exposes an API providing different objects encapsulating specific parts of MsQuic handling certain functionalities [12]. An overview of these objects is shown in Figure 2. The *Registration* object provides an execution context for the child objects; typically only one *Registration* per application should be created. A *Regestration* is associated with a *Configuration* object abstracting all of the settings available to change the behavior of the library. When building a server *Listener* objects are used to accept incoming connections from clients. The more *Listener* objects are created, the more simultaneous requests the server will be able to handle. A successfully accepted connection by a *Listener* will result in a new *Connection* object representing the connection between the server and a client. This *Connection* object can create or accept arbitrarily many *Stream* objects, each one representing an individual QUIC stream that is used for data transmission.

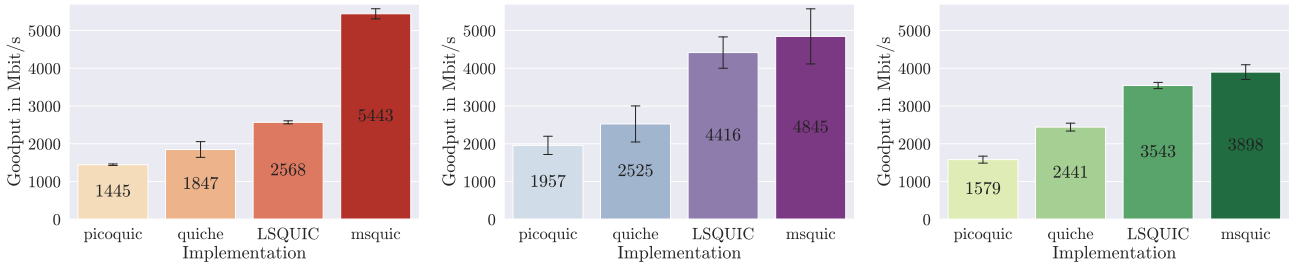
The API uses callbacks to indicate that an event has happened. The functions that are registered to these callbacks are not run in separate threads to MsQuic and as such should keep the execution time to a minimum, to not block execution of the library.

4. Evaluation

We compare the performance of MsQuic to other QUIC implementations and chose the following three. Quiche, which is the QUIC implementation from Cloudflare written in Rust. LSQUIC, an implementation from LiteSpeed, which is written in C and lastly picoquic, that has the goal of being a minimal but fully functional implementation of the QUIC protocol and is also written in C. We chose these three implementations because they prioritize performance and most importantly are all written in a systems programming language, making them good targets for performance and speed comparisons with MsQuic.

4.1. Setup

To conduct our measurements we used the test framework by Jaeger et al. [13], which allowed us to execute the tests on real hardware inside of a testbed. With this framework the measurement procedure was the following. Two seperate hosts are selected, one acting as a client and one as a server. The server hosts a webserver, exposing a 1073 MB large file and the client attempts to download it. Measuring the time this data-transfer takes, we calculate the average transmission speed. After the transfer is complete we make sure that the file was transmitted properly and did not get altered during transfer. This is called goodput instead of throughput because instead of measuring the raw amount of data transmitted, it takes into account what portion of the data is actually useful. It gets impacted negatively by retransmissions of packets and does not count packet headers as useful data and because of that it is a better representation of real world usage. We conducted these measurements ten times for each implementation, to reduce random deviations in the measured goodput. We performed this procedure on three different host pairs, with differing CPUs, to examine the how the performance of the QUIC implementations changes with different hardware. Each of the first two hostpairs had an Intel Xeon E5-1650 v3 CPU with 6 cores and 12 threads, running at 3.5Ghz. This is an older CPU



(a) Intel Xeon E5-1650 v3 (b) Intel Xeon Gold 6312U (c) AMD EPYC 7543

Figure 3: Goodput of the four QUIC implementations on different CPU architectures

from around 2014, giving insight into the performance on an aged Haswell architecture. The other two hostpairs we tested on had fairly recent CPUs: Intel Xeon Gold 6312U with 24 cores and 48 threads running at 2.4Ghz and the second hostpair used AMD EPYC 7543 CPUs with 32 cores and 64 threads, running at 3.7Ghz. The connection between each of the hostpairs was made with a 10GBase-T Cat6a network cable with full duplex mode.

4.2. Results

Our results are visualized in Figure 3, with the black lines at the top of the bars indicating the standard deviation of the measured samples. Interestingly MsQuic performed best on the older hardware, as seen in Figure 3a, achieving a mean goodput of 5443 $\mathrm{Mbit\,s^{-1}}$ and significantly outperforming all other implementations. On the newer Intel CPU all implementations achieve a higher goodput, except for MsQuic, which loses around 598 $\mathrm{Mbit\,s^{-1}}$ of mean goodput as depicted in Figure 3b. To be noted as well is the larger deviation of the measurements, indicating that the speed of the connection was unstable. This is not the case for the AMD host pair as illustrated in Figure 3c. None of the implementations achieve as high goodputs as measured on the new Intel CPU, but the achieved speeds are much more stable. MsQuic achieved the lowest goodput on this host pair, with a mean of only 3898 $\mathrm{Mbit\,s^{-1}}$.

From our resulst it seems like MsQuic achieves higher goodputs on older hardware, outperforming all the other implementations by a large margin. MsQuic still surpasses all the other implementations on the newer hardware, however with a much smaller margin. This behaviour might be caused by differences of the CPU architectures, but more measurements and research needs to be done to reach a conclusion.

5. Related work

This paper takes a close look at MsQuic, measuring its performance and comparing it to other QUIC implementations. A number of similar works exist that investigate the improvements the QUIC protocol makes [14]–[16]. Carlucci et al. check if QUIC can be deployed safely and compare QUIC to HTTP and SPDY [15]. The work of Cook et al. discovered specific use cases and conditions where QUIC is of high interest [14]. Yu et al. measured the performance of QUIC in production environments, locating the largest bottlenecks of deployed QUIC applications [16]. Seemann and Iyengar introduce the QuicInteropRunner, a framework to test interoperability and performance between different QUIC implementations [17].

Other similar works propose concrete techniques to improve the performance of QUIC implementations [18]–[20]. Yang et al. tackle the problem of QUIC using up to 3.5 times the CPU cycles of optimized TCP and TLS implementations [18]. They examined QUIC implementations to discover the computationally most intensive parts and using their findings to define an architecture for offloading these calculations to NICs. Tyunyayev et al. introduce picoquic-dpdk, a modification of picoquic that uses the DPDK library to bypass the Linux kernel networking stack, reducing the amount of slow context switches [19]. A different approach to circumventing large amounts of context switching is presented by Wang et al. [20]. They developed an implementation of QUIC that runs in kernel-space and used it to more accurately compare TCP and QUIC.

6. Conclusion

The last few years have been a success story for QUIC. It provides significant improvements over the existing TCP / TLS stack. Deployment of this new technology was fast and user adoption was quick. All because QUIC is built on top of UDP, making it effortless to deploy in existing network infrastructure and because it runs in user-space, making it not depend on operating systems to implement it. Between the many implementations of the QUIC protocol MsQuic stands out with its great performance, not only in regards to raw goodput but also the ability to handle a large number of requests per second. The team building MsQuic at Microsoft, has shown their dedication to building a high performance QUIC implementation by putting great efforts into automatic performance measurements and testing. Ensuring that every change to the code base is inline with their performance goals. On top of that MsQuic is open source under an MIT License, has cross platform support and good documentation and is under active development backed by a major corporation. We were able to validate their claims of high performance. MsQuic has double the goodput of LSQUIC, the second fastest QUIC implementation we tested. All of these points make MsQuic a good choice for any project that needs support for the QUIC protocol.

References

[1] 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

[2] 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

[3] Microsoft, "MsQuic," 2023, last accessed 24 March 2023. [Online]. Available: https://github.com/microsoft/msquic

[4] N. Banks, "MsQuic - QUIC Performance Talk," 2021, last accessed 26 March 2023. [Online]. Available: https://www.youtube.com/watch?v=Icskyw17Dgw

[5] M. Bishop, "HTTP/3," RFC 9114, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114

[6] "Nicholas Banks LinkedIn," last accessed 13 May 2023. [Online]. Available: https://www.linkedin.com/in/nicholas-banks-a3977520/

[7] Microsoft, "Platform Support of MsQuic," last accessed 8 May 2023. [Online]. Available: https://github.com/microsoft/msquic/blob/9f74f69d0c16fadb62a332246daabac704bc7db0/docs/Platforms.md

[8] ——, "SMB over QUIC," last accessed 8 May 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-over-quic

[9] ——, "MsQuic for Microsoft Game Development Kit," last accessed 8 May 2023. [Online]. Available: https://learn.microsoft.com/en-us/gaming/gdk/_content/gc/networking/overviews/game-mesh/msquic-intro-networking

[10] N. Banks, "QUIC Performance," 2021, last accessed 26 March 2023. [Online]. Available: https://datatracker.ietf.org/doc/draft-banks-quic-performance

[11] ——, "MsQuic Performance Dashboard," last accessed 26 March 2023. [Online]. Available: https://microsoft.github.io/msquic/

[12] Microsoft, "MsQuic API," 2023, last accessed 24 March 2023. [Online]. Available: https://github.com/microsoft/msquic/blob/main/docs/API.md

[13] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, "QUIC on the highway: Evaluating performance on High-Rate links," in *International Federation for Information Processing (IFIP) Networking 2023 Conference (IFIP Networking 2023)*, Barcelona, Spain, Jun. 2023.

[14] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui, "Quic: Better for what and for whom?" in *2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6.

[15] G. Carlucci, L. De Cicco, and S. Mascolo, "HTTP over UDP: An Experimental Investigation of QUIC," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 609–614. [Online]. Available: https://doi.org/10.1145/2695664.2695706

[16] A. Yu and T. A. Benson, "Dissecting Performance of Production QUIC," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1157–1168. [Online]. Available: https://doi.org/10.1145/3442381.3450103

[17] 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

[18] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, "Making QUIC Quicker With NIC Offload," 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. 21–27. [Online]. Available: https://doi.org/10.1145/3405796.3405827

[19] N. Tyunyayev, M. Piraux, O. Bonaventure, and T. Barbette, "A High-Speed QUIC Implementation," in *Proceedings of the 3rd International CoNEXT Student Workshop*, ser. CoNEXT-SW '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 20–22. [Online]. Available: https://doi.org/10.1145/3565477.3569154

[20] P. Wang, C. Bianco, J. Riihijärvi, and M. Petrova, "Implementation and Performance Evaluation of the QUIC Protocol in Linux Kernel," in *Proceedings of the 21st ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, ser. MSWIM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 227–234. [Online]. Available: https://doi.org/10.1145/3242102.3242106