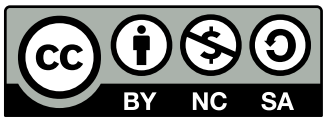
Aalto University
School of Electrical
Engineering

Master's Programme in Computer, Communication and Information Sciences

Performance of Server Message Block implementations over QUIC

David Enberg

Master's Thesis
2025



Aalto University
School of Electrical
Engineering

Author David Enberg

Title Performance of Server Message Block implementations over QUIC

Degree programme Computer, Communication and Information Sciences

Major Communications Engineering

Supervisor PhD Pasi Sarolahti

Advisor Bastian Shajit (MSc)

Collaborative partner Tuxera Oy

Date 28 November 2025 **Number of pages** 24 **Language** English

Abstract

Keywords For keywords choose, concepts that are, central to your, thesis

Aalto University
School of Electrical
Engineering

Författare David Enberg

Titel Arbetets titel

Utbildningsprogram Electronik och electroteknik

Huvudämne Communications Engineering

Övervakare Prof. Pirjo Professori

Handledare TkD Alan Advisor, DI Elsa Expert

Samarbetspartner Company or institute name in Swedish (if relevant)

Datum 28 November 2025 **Sidantal** 24 **Språk** engelska

Sammandrag

Nyckelord Nyckelord på svenska, temperatur

Preface

Otaniemi, 30 June 2025

Eddie E. Engineer

Contents

Abbreviations

ACK	acknowledgment
HOL	Head-of-line
IP	Internet Protocol
ISN	Initial Sequence Number
NAT	Network Address Translator
OS	Operating System
RFC	Request For Comment
RTT	Round-Tripe Time
SMB	Server Message Block
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

1 Introduction

1.1 Research questions and objectives

1.2 Thesis structure

2 Background

This section of the thesis will give an overview of the two most common transport protocols, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). This section also covers the Server Message Block (SMB) protocol, outlining the file-sharing protocol.

2.1 Internet transport protocols

2.1.1 Transmission Control Protocol

The TCP, as outlined in Request For Comment (RFC) 793 is a foundational internet transport protocol. It was originally published in September 1981, focusing primarily on solving military communication challenges. It is intended to be a highly reliable transport protocol between hosts in a packet switched network. The TCP is connection-oriented, providing reliable, end-to-end, bi-directional communication between a pair of processes, in the form of a continuous stream of bytes. The TCP protocol is designed to fit into a layered hierarchy of protocols, slotting in on top of the internet protocol (IP)[1]. IP handles the addressing and routing of datagrams between the hosts, while TCP aims to ensure that information is delivered correctly, in order, and without duplications, without any reliability guarantees needed from the underlying protocol, which may lose, fragment or reorder the datagrams[2].

TCP ensures reliable communication by using a system of sequence numbers and acknowledgments (ACKs). Each transmitted byte of data is assigned a sequence number, and the peer is required to send an ACK to acknowledge that the data was received. On the receiver side the sequence numbers are used to reconstruct the data, ensuring that the data is received in order. If the sender does not received an ACK within a timeout period, the missing segment will be retransmitted. A checksum is included with each segment, ensuring that the datagram was not corrupted during transport. If data corruption is detected, the receiver will discard the damaged segment, and rely on the retransmission mechanic to recover. The TCP uses a receive window for flow control, allowing the receiver to decide the amount of data that the sender may send before waiting for further ACKs. The reliability and flow control aspects of the TCP demands that the TCP store some information about the transmission. The data stored about the data stream, sockets, sequence numbers and windows sizes, is referred to as a connection. The network address and port tuple is referred to as a socket, and a pair of sockets is used in identifying the connection. Using this mechanic to uniquely identify connections, allows for multiple processes to simultaneously communicate using the TCP[2].

The TCP header, figure 1, encodes the functionality of the TCP. It follows the IP header in a datagram. The header is usually 20 bytes long, but can be extended using options. It begins with the source and destination port, which together with the source and destination addresses, are used to identify the connection. The next two fields in the header are the sequence and acknowledgement numbers. The sequence number is the sequence number of the first data byte in the data segment. If the SYN

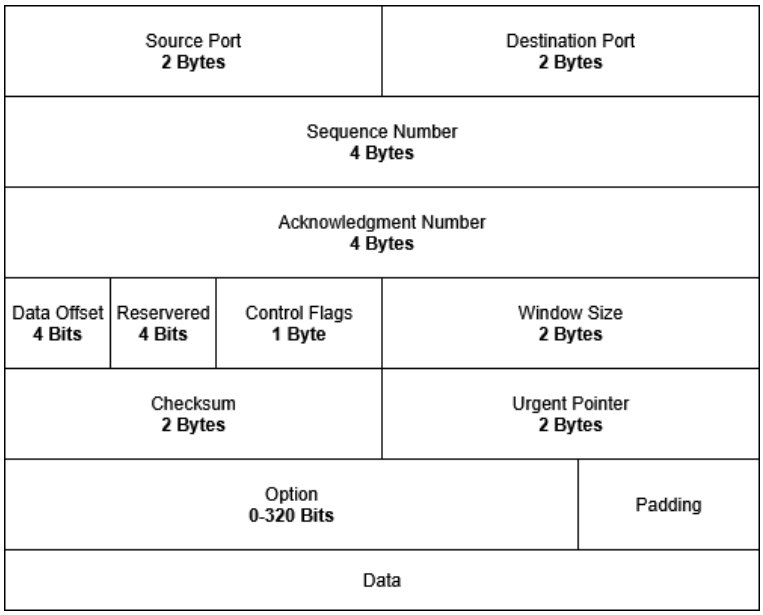


Figure 1: The TCP header

flag is set the sequence number is referring to the initial sequence number (ISN). The acknowledgement refers to the next sequence number the receiver is expecting to receive, at the same time acknowledging that all sequence numbers up to this point was received. Next is the data offset field, indicating where the data begins. The reserved field following this must be 0. After this is the 1 byte flags field

- **URG** Urgent pointer field is set
- **ACK** acknowledgement field is set
- **PSH** Push function, requesting that buffered data is sent immediately to the receiver
- **RST** Reset the connection
- **SYN** Synchronize sequence numbers
- **FIN** Sender is done sending data

The 2 byte window field specifies the number of bytes that may be in-flight at any one time. This is the specified size of the sliding window that is used for flow control purposes. Following the window field is a 2 byte checksum field, used for detecting corruption of the TCP-header, data payload as well as a pseudo IP header, containing information about the source and destination addresses, as well as the protocol number and tcp packet length. In case the URG bit is set in the flags field, the 2 byte urgent pointer header field indicates where the urgent data ends. Finally the options field contains extension to the normal TCP header, containing among other, options for maximum segment size and multipath TCP[3].

To ensure reliable delivery of TCP segments, each segment is assigned a sequence number. This allows the receiver to reconstruct segments delivered out of order, and additionally detect missing segments. The receiver sends acknowledgments, containing the next expected sequence number, to signal to the sender that the data was successfully received. The sequence number is a 32 bit number, with the initial sequence number (ISN) selected randomly at the time when the TCP connection is established. This ensures that sequence numbers from stale connections have a low probability of overlapping with any active connection[2].

The TCP connection is established via a three-way handshake. The client sends a TCP packet with the SYN bit set in the flags field, this packet contains the clients ISN. The server responds with a packet with the SYN and ACK bit set, acknowledging the clients sequence number as well as providing the servers ISN. Finally the client responds with an ACK, acknowledging the servers ISN. Following this the client and server are synchronized, and communication may begin. A peer may terminate its side of the connection by sending a FIN packet, signaling to the other endpoint that one side has closed its side of the communication. The closed endpoint may continue receiving data until the other endpoint also closes it side[2].

2.1.2 User Datagram Protocol

The UDP, which was defined by RFC 768, is designed to enable programs to transmit self-contained messages, know as datagrams, over a packet-switched network. The UDP is designed to run on top of the IP[1], using IP addresses and port numbers for addressing. The UDP is by design connectionless, providing no guarantees for datagram delivery, duplicate datagrams or in-order delivery. In exchange, the UDP aims to minimize the overhead present in the protocol. As UDP is connectionless there is no need to establish a connection via a handshake, instead datagrams can be transmitted directly, and they should be designed in such a way that they can stand on their own. The UDP header, as seen in figure 2 is only 8 bytes long, consisting of the source and destination port, as well as the length of the datagram and a checksum to verify the received datagram[4]. Even though UDP has a checksum field its use varies depending on the implementation. Some implementations may discard the datagram, or alternatively pass it along to the application with a warning, as UDP provides no way to recover from broken datagrams[5]. The minimal UDP header (8

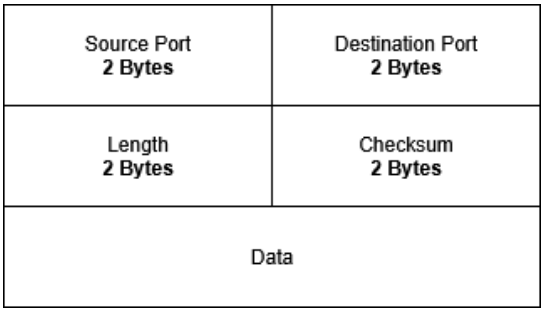


Figure 2: The UDP header

bytes) combined with the lack of a handshake makes UDP a protocol with the bare minimum needed for datagram transfer. Generally, UDP is used for applications where low latency is a requirement and some amount of packet loss is deemed acceptable. It is then up to the application layer to handle missing, reordered or duplicate datagrams.

2.2 The Server Message Block protocol

3 QUIC

The Internet's underlying infrastructure is in a state of constant evolution, driven by a demand for decreased latency, increased throughput requirements and a need for improved security. For many years now, the TCP has been the de-facto solution for reliable and secure, when combined with Transport Layer Security (TLS), communications. However, TCP was developed in a time where security and latency where not considerations, at least not in the same way as in todays landscape. Over the years there have been efforts to enhance TCP, such as multipath TCP[3] and combining TCP and TLS in HTTPS[6] to improve security. This section of the thesis will cover QUIC, a transport protocol developed to overcome the limitations and improve the performance as compared to the TCP[7].

The importance of the QUIC protocol is not to be underestimated. It represents a substantial change to the internet's transport layer, the first one in over two decades. QUIC was initially developed by Google, and then later standardized in RFC 9000[8]. QUIC is designed to address the issues experienced by TCP, with a focus on optimizing for web traffic. The main issues being the Head-of-line (HOL) blocking, but also aiming to improve on other aspects, such as integrating the TLS handshake into the transport handshake. A decision that was made for QUIC specifically was to move the protocol out of the kernel space and into the user space, allowing for rapid development and innovation[7].

This chapter of the thesis will outline the limitations of TCP that led to the development of QUIC, give an overview of the architecture and logic that drives QUIC.

3.1 The motivation for a new transport protocol

The TCP is a cornerstone of modern internet infrastructure. It has been a reliable work horse for more than 40 years, ensuring communications between users and hosts since its inception. However, as the design of TCP is largely colored by the landscape when it was created, much of the improvements that have been made to TCP, such as security, has had to be built on top of TCP, as these were not considerations at the time of the TCP's inception. Todays internet landscape, with real-time content, hyper mobile users and increased demands on latency, but also privacy and security, have exposed some of the limitation imposed by the TCP stack. This section will outlined the key issues that prompted the development of a new, modern protocol: QUIC.

3.1.1 Head-of-Line Blocking

The TCP guarantees that all frames will be delivered, reliably in-order. Seeing as the TCP works as a single byte-stream the creates a phenomenon know as Head-of-Line (HOL) blocking, where any lost packet will block the delivery of all subsequent packets until the missing packet has been retransmitted. This can potentially amplify issues in the network, increasing delays, decreasing throughput and worsening the user experience. To combat this limitation, modern network protocols, such as HTTP/2[9], have introduced measures to combat the issue. HTTP/2 introduced multiplexing of

multiple requests over one connection, allowing multiple application-level streams, for example for different resources such as images or javascript, to be multiplexed over a single TCP connection. This means that HTTP/2 managed to mitigate application-level HOL blocking, which was an issues in earlier versions of HTTP, it still potentially suffered from transport level HOL blocking, as the multiplexed streams was still sent over a single TCP connection. As a result a single lost packet in the TCP stream still caused all other unrelated streams over the same connection to stall, even if their packets were successfully delivered, until the offending packet could be retransmitted. This in practice means that much of the improvements made by HTTP/2 in this regard was negated by the issues of TCP, particularly in lossy environments[10].

3.1.2 Handshake Delay

A limitation of the TCP stack is the delay caused by the TCP handshake. As discussed in earlier chapters, establishing a TCP connection is done via a three-way handshake (SYN, SYN-ACK, ACK). This handshake incurs one Round-Trip Time (RTT) of delay. In addition, most application use TLS for security, and historically the TLS 1.2 handshake and setup adds two additional RTTs of delay. While network bandwidth is ever increasing, much of the communication done on the internet consist of short dialogues, that are significantly impacted by the additional overhead brought on by the TCP plus TLS handshake[7].

Some of the latency brought on by the TLS handshake is addressed by TLS 1.3, adding support for 1-RTT and 0-RTT handshakes, at the cost of perfect forward secrecy[11]. Even with these enhancements, the TCP plus TLS handshake takes a minimum of 1,5 RTTs, due to the separation of connection and security handshake.

3.1.3 Protocol Ossification

A big hurdle in deploying new protocols and extensions to existing ones on the internet, is the protocol ossification of existing protocols on the internet. There exists a heap of middleboxes, such as Network Address Translators (NATs) and firewalls that are part of the network. These devices may be overly conservative, dropping or modifying packets that do not conform to their assumption. This is already an issue for TCP enhancements, and entirely new protocols have no chance of reaching their destination, without explicitly adding support in all necessary middleboxes. To get around this, protocol designer have to design their protocols from the ground up to be middlebox proof, such as QUIC encapsulating its protocol inside UDP as an anti-ossification measure[12].

A related issue of rolling out enhancements to existing protocols is that the network stack tends to be part of the Operating System (OS) kernel. The networking stack is tightly coupled to the OS, requiring OS updates or upgrades to implement changes to existing protocols. With todays upgrade frequency it can take years to roll out simple changes to the networking stack. QUIC moves the deployment of the protocol to the user space, improving the speed of development and deployment, and opening up the space for multiple actors to create their own implementations of the protocol[7].

3.2 Background and evolution

As discussed in Section 3.1, the combinations of TCP, TLS and HTTP/2 are plagued by issues that are difficult to circumvent without major overhauls or extensions to the individual protocols, which due to protocol ossification is increasingly difficult. With this in mind, a new protocol was being created, aiming to solve the issues of HOL blocking, improved latency and circumventing protocol ossification. The answer was the protocol that would later be standardized into QUIC, early on know as gQUIC. QUIC began development back in 2012, by Jim Roskind, an engineer at google. Initially the motivation for developing a new transport protocol was to improve support for the now deprecated SPDY protocol[13]. QUIC was designed to run over UDP, by encapsulating the protocol frames into UDP datagrams, and encrypting the contents, the protocol could effectively sidestep the issues of middlebox interference, allowing for rapid deployment without any necessary modifications to existing infrastructure. To combat the issue of HOL blocking, QUIC implements transport level multiplexing of data streams, allowing multiple independent streams to exist over one connection. Packet loss in any of the data streams would not affect any of the other, blocking only itself while waiting for retransmission. QUIC uses a combined connection and cryptographic handshake, minimizing the latency of establishing a new connection. While TCP uses IP-port tuples to identify connections, this does not allow for mobility of the end user. If the IP or port of the user changes during the lifetime of the connection the connection is dropped, and has to be reestablished. To combat this QUIC uses Connection IDs to identify connections, allowing the connection to resume when there is some change in network. QUIC was widely deployed on Google's front end servers, and by 2017 it was already estimated that QUIC represented 7% of global internet traffic[7].

QUIC was submitted to the IETF for consideration in 2016, and a working group was created for the purposes of standardizing QUIC. The goals of the working group was to create a general purpose transport protocol that contained the benefits of gQUIC. The custom cryptographic handshake was replaced with TLS 1.3, the packet header was reworked into two types, a long and a short header, that was mostly encrypted to prevent middlebox interference. Loss detection and congestion control mechanics were updated, flow control semantics were separated into per stream and per connection limits and version negotiation was introduced to enable future compatibility of the protocol[8]. In the end the protocol was standardized in a number of RFCs. RFC 9000[8] defines QUICs core transport mechanics, RFC 9001[14] defines the use of TLS 1.3 and RFC 9002[15] describes the loss detection and congestion control algorithms used by the protocol. In addition, RFC 8999[16] defines some version-independent properties of QUIC, aligning QUICK packets, headers and versioning between different versions of the protocol. Following the standardization the adoption of QUIC has been quick. Chromium, and by extensions all chromium based browsers has supported QUIC since before it was standardized[17]. Both Firefox[18] and Safari[19] added support for QUIC soon after the standards were published. QUIC has shown the viability and potential of deploying network protocols to the user space, enabling rapid adoption without OS kernel updates.

3.3 Architectural Overview

The QUIC protocol is designed to be a general-purpose, secure and multiplexed transport protocol, working on top of UDP. In comparison to TCP, which usually is part of the OS kernel, QUIC is typically implemented in the user space, enabling quick iteration and deployment of protocol enhancements. This section describes QUIC's position in the networking stack, the different architectural parts and the basic elements of the protocol's operation.

QUIC packets are directly encapsulate inside UDP datagrams. This design has both practical and implementation advantages. When deploying QUIC, the fact that the wire image of QUIC packets are identical to that of UDP datagrams, means that they pass seamlessly through middleboxes and firewalls, without suffering the adverse effects of protocol ossification as is often the case in TCP. As discussed in Section 2.1.2, the UDP is a barebones protocol, with the minimum overhead needed to transmit datagrams. This works to the advantage of the designer when building a protocol on top of UDP, as the designer the freedom they need to implemnt their own semantics, withouth risking interference from the underlying protocol. UDP provides the basic datagram services that are then enhanced by the QUIC protocol, ensuring a secure, reliable and performant protocol[7].

From an architectural perspective, QUIC incorporates three main layers, a transport layer, a security layer and an application interface. From the bottom up, UDP provides minimal mechanism for transmitting datagrams, withouth any guarantees. On top of this QUIC implements its own transport layer, handling the multiplexing of datastreams, ensuring reliable and in-order delivery of data as well as connection management mechanics[8]. QUIC security layer is fully integrated into the protocol, using TLS 1.3 for encryption of wire traffic, as well as authentication and authorization of peers, ensuring secure communications between the endpoints. The security layer also protects most of the packet headers, leaving only the necessary info for routing and version control visible on the wire[14]. The final layer exposes a standardized application interface that can support virtually any application-layer protocol, the most prominent one being HTTP/3[9].

One of the main innovation made by the QUIC protocol is the concept of transport-level, mutiplexed and independent data streams. Any QUIC conneciton may contain one or multiple data streams, seen entirely as their own independant object. This helps mitigate the HOL blocking issues, as if the application is using multiple streams, when loss occurs, only the stream on which the loss occured will be blocked. While the lossy stream is blocking and waiting for retransmission, the other streams can continue sending as normal. QUIC uses per connection limits for the number of streams and per stream flow control limits[8].

Connection management and and identification in QUIC differs in the IP-port tuple combination that is tranditionally used in TCP. QUIC connection are identified by a connection ID, which are independtly selected by the endpoints. The purpose of the connection IDs is to allow the conneciton to survive changes in the network, for example when a mobile users moves from a local network to cellular. The migration is done transparently and securly, allowing the application to continue operations

withouth interruption[8].

4 The Server Message Block protocol

4.1 Information about the SMB protocol

5 Implementing QUIC as transport for SMB server

5.1 MsQuic architecture and API

5.2 Fusion SMB server QUIC transport layer design

6 Performance and interoperability benchmarking

6.1 Test environment

6.1.1 Hardware environment

6.1.2 SMB over QUIC implementations analyzed

Windows SMB client/server

Fusion SMB server

6.2 Test scenarios

6.2.1 interoperability tests

6.2.2 Becnhmarking workloads

6.3 Results

7 Conclusions

7.1 Discussion

7.2 Future work

References

[1] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791.

[2] *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793.

[3] A. Ford et al. *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. Mar. 2020. DOI: 10.17487/RFC8684. URL: https://www.rfc-editor.org/info/rfc8684.

[4] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

[5] J. Kurose and K. Ross. *Computer networking: A top-down approach, global edition*. en. 8th ed. London, England: Pearson Education, June 2021.

[6] E. Rescorla. *HTTP Over TLS*. RFC 2818. May 2000. DOI: 10.17487/RFC2818. URL: https://www.rfc-editor.org/info/rfc2818.

[7] A. Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196. ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. URL: https://doi.org/10.1145/3098822.3098842.

[8] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. May 2021. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000.

[9] M. Thomson and C. Benfield. *HTTP/2*. RFC 9113. June 2022. DOI: 10.17487/RFC9113. URL: https://www.rfc-editor.org/info/rfc9113.

[10] H. de Saxcé, I. Oprescu, and Y. Chen. "Is HTTP/2 really faster than HTTP/1.1?" In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2015, pp. 293–299. DOI: 10.1109/INFCOMW.2015.7179400.

[11] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

[12] K. Edeline and B. Donnet. "A Bottom-Up Investigation of the Transport-Layer Ossification". In: *2019 Network Traffic Measurement and Analysis Conference (TMA)*. 2019, pp. 169–176. DOI: 10.23919/TMA.2019.8784690.

[13] J. Roskind. *QUIC: Design Document and Specification Rationale — docs.google.com*. https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit?usp=sharing. [Accessed 14-08-2025].

[14] M. Thomson and S. Turner. *Using TLS to Secure QUIC*. RFC 9001. May 2021. DOI: 10.17487/RFC9001. URL: https://www.rfc-editor.org/info/rfc9001.

[15] J. Iyengar and I. Swett. *QUIC Loss Detection and Congestion Control*. RFC 9002. May 2021. DOI: 10.17487/RFC9002. URL: https://www.rfc-editor.org/info/rfc9002.

[16] M. Thomson. *Version-Independent Properties of QUIC*. RFC 8999. May 2021. DOI: 10.17487/RFC8999. URL: https://www.rfc-editor.org/info/rfc8999.

[17] *QUIC, a multiplexed transport over UDP — chromium.org*. https://www.chromium.org/quic/. [Accessed 14-08-2025].

[18] *How to enable HTTP/3 support in Firefox*. https://www.ghacks.net/2020/07/01/how-to-enable-http-3-support-in-firefox/. [Accessed 14-08-2025].

[19] *Examining HTTP/3 usage one year on — blog.cloudflare.com*. https://blog.cloudflare.com/http3-usage-one-year-on/. [Accessed 14-08-2025].