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	29
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**	29	**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

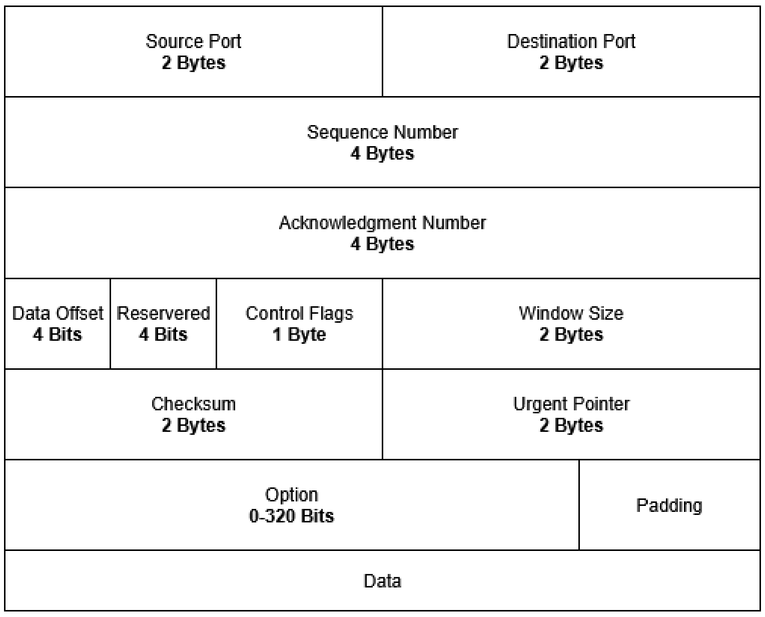

Figure 1: The TCP header

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

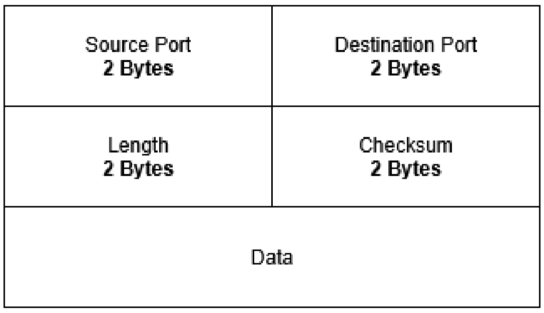

Figure 2: The UDP header

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].

3.4 Packet and Frame Structure

QUIC packets are contained inside UDP datagrams, and together with the encryption and integrity protection provided this gives a robust protection against protocol ossification. As compared to TCP, where segmentation and reassembly of packets are transparently done as part of the transport protocol and not accessible to the application, QUIC defines a large set of different frames types for different, distinct roles in conneciton establishments, data transfer and protocol logic management. Depending on the packet type QUIC uses either a long header or short header.

QUIC packets are divided into two general categories, those that use the long header and those that use the short header. The long header is used during the establishment phase of the connection, being used in the Version Negotation, Initial, 0-RTT, Handshake and Retry packets. The long header, as seen in Figure 3, contains the necessary data for these function. The first bit of the long header is set to 1 to indicate that this is a long header. The fixed bit following this is always set to 1, except for Version Negotation packets. The Long Packet type indicates the type of packet, and the 4 type specific bits are dependant on the packet type. Following this, the version ID field indicates the specific QUIC version this packet is using, and finally the destination and souce ID length and IDs contain the identifiers for the source and destination of this packet. Following the header is, if relevant, the specific packet type payload. In addition, the Initial, 0-RTT and Handshake packet types includes a packet number length and packet number field. In comparison, the short header is more compact, reducing per-packet overhead. The Header Form bit (set to 0 in the short header case) and the Fixed bit is the same as for the long header. The Spin

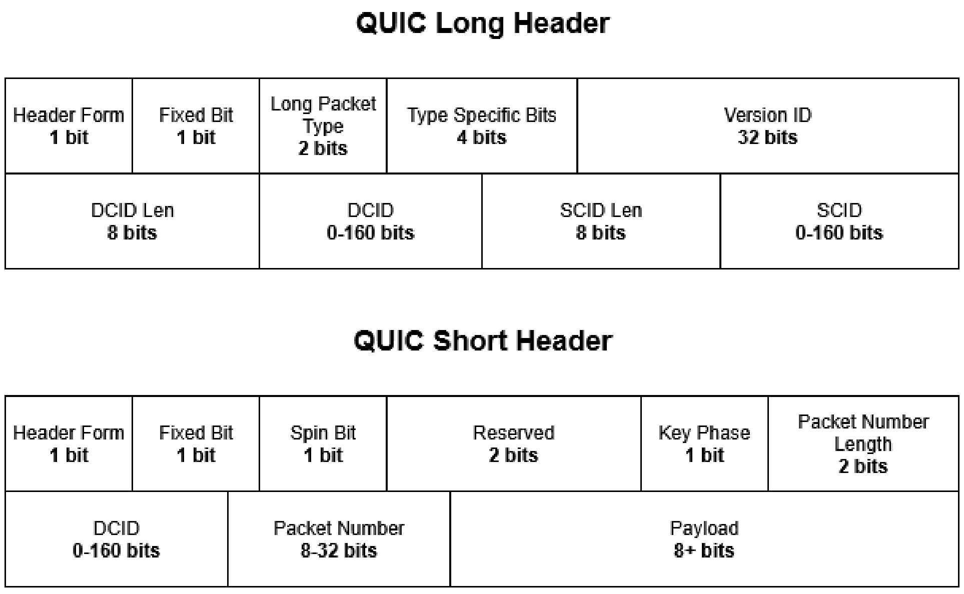

Figure 3: The QUIC header formats

Table 1: Table of QUIC long header types

Packet Type	**QUIC v1**	**QUIC v2**
Initial	0x00	0b01
0-RTT	0x01	0b10
Handshake	0x02	0b11
Retry	0x03	0b00

Bit that follows may be used to make on path estimations of the RTT. Following the two reserverd bits is the key phase bit, used to keep track of the keys that are used to protect the packet. The packet number length is the length of the packet number field minus one. Lastly is the destination ID again indicating the recipient, the packet number used for protocols semantics and finally the packet payload[8].

The long header packet type numbers differs between QUIC version 1 and version 2, as is outline in Table 1. Intial packets are used to initiate the connection and transmit the initial TLS hanshake. The Handshake packet carries the subsequent cryptographic messages, after the intial exchange. In case the connection attempt is a resumption, the client may use the 0-RTT packe type to transmit encrypted application data as part of the hanshake, basing the encryption on previously established keys. The Retry packet type may be used by the server to force the client to validate its address[8].

In QUIC there is only one short-header packet type, know as the 1-RTT packet. It is used for the bulk of communication, taking advantage of the more compact short header to reduce overhead on the connection, which is useful for high-throughput applications. The header is in part protected by header protection, preventing interference by middleboxes on the wire.

QUIC uses the concept of frames to enable protocol funcitonality. The payload of QUIC packets consists of one or more frames, and the frames can be of different types, enabling the transmission of application data and control data in the same packet. Frames allow the QUIC protocol to signal transport events between the peers. Arguably the most important frame type for data transmission is the STREAM frame. The stream frame carries stream data associated with a specific stream. The STREAM frame contains a stream identifier, byte offset and flags to track the state of the stream. The STREAM frame implicitly creates a new stream in case the identifier is previously unknown. The ACK frame is then used to acknowledge received packets. The ACK frame contains one or more ranges of packet numbers that have been received, with possible gap values indicating missing packets. This higher granularity enables the sender to only retransmit the missing packets, lowering retransmission overhead in reordering scenarios. Other important frame types include MAX_DATA and MAX_STREAM_DATA for flow control on the connection and stream level, PING frames to probe the peer and ensure that the connection staus alive, CRYPTO frams for carrying TLS hanshake data and CONNECTION_CLOSE to terminate the connection. These are the most important frame types, with an extensive list being available in RFC 9000 chapter 19 Frame Types and Formats. Together, these frames allow for great control of transport functionality, allowing for explicit control of the semantics[8].

3.5 Streams

The QUIC protocol is designed around the concept of streams. Streams are an abstraction of a byte-streams, that are used to transmit data between applications. Streams may be either bidirectional or unidirectional, that is, from the perspective of one peer, the streams may be read-only, write-only or read and write. Any single connection may contain one or more streams, with the streams being multiplexed and entirely indedependent from one another. Streams are design to be lightweight, with minimal overhead. A stream can open, closed and transmit data in a single frame, or alternatively the streams can persist for longer lengths of time, up to the lifetime of the connection. The number of streams and the amount of data that may be sent over a single stream is constrained by the flow control. The different streams are identified via a unique identifier inside a connection, the stream ID, which is a 62-bit integer. The reasoning for using a 62-bit integer is that the two remaining bits are used to distinguish between unidirectional and bidirectional streams, as well as identify the initiator of the stream, which may be either the client or server[8].

The lifetime of a stream is explicitly managed by the peers. The stream is first created by sending a stream frame, containing the stream ID of the stream that is requested to be opened. Data transfer id done via STREAM frames, that carry the application data that is to be transmitted. Once data transfer is complete, and the application is done with the stream, it may be closed by sending a stream frame with the FIN bit set, or alternatively send a RESET_STREAM to abruptly end the stream, signaling that no further data will be sent. On the receiver end the receiver can read data from the stream, and may end the stream by sending a STOP_SENDING frame, signaling that no more data will be accepted on the stream[8].

The desing of QUIC streams is such that it directly addresses TCP's HoL blocking issue. In TCP, the in-order delivery guarantee is implement on an entire connection, resulting in the whole connection stalling in case of a lost packet until it is retranmsitted and received. In comparison, QUIC building it's streams on top of UDP, have implemented an in-order guarantee per stream. That is, any lost packet only blocks the specific stream that packet belonged to, allowing the other streams that may exists on the same connection to keep transmitting while the erronous stream retransmits. This feature also works in the favor for control data, as control frames is transmitted separately from the stream frames, they are not affected by loss on any of the data streams[8]. An example of this advantage in action is in the HTTP/3 protocol. Here each client request is mapped to its own bidirectional QUIC stream, and unidirectional streams are used for control and push streams. HTTP/3 implementations should support at least 100 concurrent streams taking advantage of the multiplexing afforded by QUIC[20]. When a client visits a web page, it may open tens of streams simultaneously, to fetch HTML, CSS, JavaScript, images and font. If a packet containing part of an image is lost, QUIC enables the HTML and CSS streams to continue delivering, improving render times in lossy environments.

The lighweight desing of QUIC streams makes the suitable for short request-response communications in other domains as well. DNS over QUIC takes advantage of the stream semantics by creating a new QUIC stream for each request. Beside preventing HoL blocking, DNS over QUIC takes advantage of QUIC's stream semantics by allowing each DNS query to use a separate stream for communication. This allows the protocol to cheaply process each request-response, forgoing the overhead of setting up a connection, with its full handshake, instead creating a lightweight stream for the specific communication. The multiplexed streams also allow the server to process and respond to the queries out-of-order, allowing multiple, simoultaneous, outstanding queries at any one time.

3.6 Flow and Congestion Control

To prevent fast senders or malicious actors from overwelming the receive buffer, the QUIC protocol implements flow control on two levels, a per-stream flow limitation and a per-conneciton limitation. The receiver controls the amount of data that can be sent on any one QUIC stream at a time, as well as over the connection as a whole. The receiver also limits the number of concurrent streams that the sender may open, preventing the overwelming of the receiver with a large number of unique streams. The limits are set during the hanshake process, but they may be updated by sending MAX_STREAM_DATA and MAX_DATA frames to signal that the flow control limits have been increase. A receiver may not reduce the flow control limits during the connection. If the sender is faster than the receiver it may reach a state where it is unable to transmit data due to either one of the limit. In this case the sender should send a STREAM_DATA_BLOCK or DATA_BLOCK frame to signal to the receiver that there is outstanding application data waiting to be sent, but it is currently unable due to the flow control limits. Similarly the limit on the number of streams that may be opened is set during the connectiion initialization phase. It may the layer be increased by sending a MAX_STREAMS frame, and similarly to the data flow control limits, the streams limits may only be increased, not decreased. The sender can signal to the receiver that it would like to open more streams by sending a STREAMS_BLOCKED frame[8]. This granular control allows the protocol implementations to adapt the flow control according to the environment, enabling improved performance especially in dynamic environments.

The QUIC protocol uses three different packet number spaces, separate for initial, hanshake and application data spaces. This is as an effect of the fact that the corresponding packets use different levels of encryption, ensuring that acknowledgements sent in one packet number space does not cause retransmissions of packets of a different encryption level. The packet numbers of QUIC are monotonically increasing, signaling the order in which the packets were sent. That is, when retransmission occurs the retransmitted packet uses the next available packet number instead of the same packet number. To reconstruct reordered packets QUIC instead uses a stream offset field inside the stream frames, to keep track of the correct order of data. Loss detection in the QUIC protocol is then based on these packet numbers. Loss can either be detected vi acknowledgement-based detection, if a packet that is considered in-flight and is unacknowledged, but a packet sent after the potential lost packet has been acknowledged, and enough time has passed since the packet was sent. Alternatively, a Probe Timeout (PTO) causes one or two datagrams to be sent out, expecting a response to test the reachability of the peer[15].

The QUIC protocol provides a set of generical signals that are design to support multiple different sender-side congestion algorithms. A congestion algorithm similar to TCP NewReno[21] is outlined in RFC 9002. However, a sender is free to choose any one algorithm they feel suits there use case, such as CUBIC, given that they conform to the congestion control algorithms oulined in RFC 8085[22]. One option is to use QUIC packets which contain only ACK frames, as these do not count towards the flow control limits, but the loss of these packets can be detected by the QUIC algorithm[15].

The algorithm oulined in RFC 9002 begins in slow start, setting the initial congestion window to 10 times the maximum datagram size. When in slow start the sender increases the congestion number by the number of acknowledged bytes, until packet loss is detected. This means that the congestion window can double every RTT as long as all inflight packets are acknowledged. When packet loss is detected the sender enters recovery period. During the recovery period the congestion windows is reduced to half its currently size, and this reduction may be done instantly when entering the recovery period or gradually using some other mechanism. Any additional detect loss during this period does not affect the congestion window size. The recovery period ends once the congestion window has been reduced to the new size, and the sender has sent a packet and that packet has been acknowledged. From here the sender enters congestion avoidance. During the congestion avoidance period, the sender may at maximum increase the congestion window by one maximum datagram size for each complete congestion window that has been acknowledged. In case loss is detected the sender goes back into recovery mode. In case persisent congestion is detected, that is the continuous loss of all sent packets, the sender reenters slow start mode, and the process starts over[15].

In the QUIC algorithm there may be a situation where reordering of packets causes the receiver to receiver packets to which it does not have the necessary keys to decrypt, such as handshake or 0-RTT packets arriving before the initial packets. Loss of these packets may be ignored, unless an earlier packet from the same packet space has been acknowledged, in which case the loss must not be ignored. When a probe packet is sent it should not be blocked by the congestion cnotrller, even if they might exceed the current congestion, and these packet should also be accounted for as being in-flight. The sender should pace the sending of its packets, as large bursts of packets may induce congestion or packet loss. A lack of application data or pacing on the part of the sender may cause the congestion window to be underutilized. In this case the congestion window should not be increased[15].

3.7 Connections

The QUIC protocol is a stateful, connection-oriented protocol where the QUIC connection functions as the shared state between peers. Connections are created via a handshake, during which both the transport and security hanshake is performed, with the security handshake being based in TLS 1.3[14]. The handshake establishes the connection, and the parameters for the connection are declared. The parameters are individually decided by the peers, and the opposite peer needs to comply with the conditions set. It is possible via 0-RTT data transfer to transfer application data as part of the initial message, before the server has responded, and the server may already start sending data before the final hanshake message from the client. This allows the protocol to exchange some security guarantess for improved latency when establishing a connection[8].

3.7.1 Connections and Identifiers

From a connection standpoint one of the innovations made by the QUIC protocol, as compared to TCP, which uses port-IP tuples to identify connections, is the Introduction of connection IDs. The point of connection IDs is to make the protocol more resilient to changes in the lower level protocols, such as IP addresses changing when a client moves from one network to another. In case of connection migration from one network to another the connection ID used is also changed at the same time. This is to prevent passive listener from tracking a peer from one network to another, which could be done via correlating connection IDs[8].

In a QUIC connection each peer decides the connection ID for the other peer. Initially the connection IDs are issued during the handshake phase, and the connection ID is associated with a sequence nubmer, starting from 0. A peer may use NEW_CONNECTION_ID frames to assign a new connection ID to its peer, at the same time increasing the sequence number by one. A peer can and should issue multiple connection IDs to its other peers, and should accept packets originating from any of these connection IDs. This give the peer a sufficient number of unused connection IDs that are available for later use during the connection. Packets from the assigned connection IDs should be accepted until they are explicitly retired, either using a RETIRE_CONNECTION_ID frame or by sending a NEW_CONNECTION_ID fram with the Retire Prior To value increased, which forces the peer to retire the related connection IDs. Sending a RETIRE_CONNECTION_ID frame implicitly requests the peer to issue a new connection ID to replace the one that has just been removed. A connection ID must not be retired withouth alerting the peer, and the number of connection IDs that have been locally retired, that is the connection ID is retired and a RETIRE_CONNECTION_ID fram has been sent, but no acknowledgement has yet been received, should be kept to a limited number[8].

When a QUIC packet is received it is attempted to match this packet to an existing connection. If the recepient is a server it is also possible that the incoming packet is part of a new connection. If it is not possible to associate the packet with any connectino ID of any existing connection, and the packet is not part of initializing a new connection, then a Stateless Reset may be sent in response. If a packet is received, but it is not consistant with the connection, the packets are discarded. Such a scenario could be incorrect versioning or inability to decrypt the packet[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].

[20] M. Bishop. *HTTP/3*. RFC 9114. June 2022. DOI: 10.17487/RFC9114. URL: https://www.rfc-editor.org/info/rfc9114.

[21] A. Gurtov et al. *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. Apr. 2012. DOI: 10.17487/RFC6582. URL: https://www.rfc-editor.org/info/rfc6582.

[22] L. Eggert, G. Fairhurst, and G. Shepherd. *UDP Usage Guidelines*. RFC 8085. Mar. 2017. DOI: 10.17487/RFC8085. URL: https://www.rfc-editor.org/info/rfc8085.