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 24 November 2025 **Number of pages** 51 **Language** English

Abstract

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

Aalto University
School of Electrical
Engineering

Författare David Enberg

Titel Utvärdering av SMB över QUIC lösningar

Utbildningsprogram Datakommunikationsteknik

Huvudämne Communications Engineering

Övervakare PhD Pasi Sarolahti

Handledare Bastian Shajit (MSc)

Samarbetspartner Tuxera Ab

Datum 24 November 2025 **Sidantal** 51 **Språk** engelska

Sammandrag

Nyckelord tbd

Preface

Tölö, 24 November 2025

David C. Enberg

Contents

Abbreviations

ACK	acknowledgment
ALPN	Application Layer Protocol Negotiation
DFS	Distributed File System
DTLS	Datagram Transport Layer Security
HOL	Head-of-line
HTTP	HyperText Transfer Protocol
IP	Internet Protocol
ISN	Initial Sequence Number
NAT	Network Address Translator
OS	Operating System
RDMA	Remote Direct Memory Access
RFC	Request For Comment
RTT	Round-Tripe Time
SCTP	Stream Control Transmission Protocol
SMB	Server Message Block
SSL	Secure Socket Layer
SST	Structured Stream Transport
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

1 Introduction

In modern communication networks, reliable, secure and high-performance internet transport protocols has become a cornerstone of communication over the internet, being essential for applications ranging from web browsing and multimedia sharing to enterprise level file sharing. The Transmission Control Protocol (TCP)[1] has served as the solution of choice for reliable communication for over 4 decades. Side-by-side with TCP, the User Datagram Protocol (UDP)[2] has been used for application with high requirements on latency, with no guarantees of reliability. As the application demands move towards higher throughput, lower latency and increased security demands, the inherent limitations of TCP has become apparent.

One demanding use of internet transports is remote file access, present at both the enterprise and consumer level. The Server Message Block (SMB)[3], is a widely adopted protocol for sharing files over a network, with implementations available in Windows, Linux and embedded systems. Traditionally the SMB protocol has used TCP to ensure reliable delivery[3]. However, the increasing demand of modern use cases, with mobile users and more security demanding applications, is increasingly highlighting the limitations of TCP.

Recently, a new transport protocol, QUIC, has been developed to address the issues and shortcomings of TCP. QUIC was initially developed by Google[4], and later standardized by the IETF in RFC 9000[5]. QUIC implements transport level multiplexing, encryption and authentication via TLS 1.3, and improved connection setup latency as parts of its design[5, 6]. QUIC has already been widely deployed as a foundation for HTTP/3[7], supporting large-scale web applications. But QUIC was not only developed as transport protocol for web applications, also finding use as a general-purpose transport protocol[5]. One of these is to serve as a transport protocol for the SMB protocol.

Although TCP has served as the reliable transport protocol for SMB during the last decades, it has several well-known drawbacks in modern networking environments. First, TCP suffers from head-of-line (HOL) blocking, which can significantly impact performance in lossy environments. Second, TCP combined with TLS requires additional messages during the connection setup, the handshake, negatively affecting the latency of connections, especially affecting short communications. Finally, TCP suffers heavily from protocol ossification, different middleboxes, firewalls and operating system (OS) implementations makes changes difficult and slow to propagate[4].

Although the improvements brought on by QUIC has been widely studied in the context of web traffic[8–10], no studies have been done on the performance of SMB using the QUIC transport protocol. While the support for alternative transport protocols in the SMB protocol, mainly RDMA and QUIC, has been defined and implemented by Microsoft[3], formal efforts into researching and comparing the performance of these alternative transport is non-existent. There is also no current implementation of SMB over QUIC for Linux based system, with currently the only available implementation being in Windows machines. SMB suffers in the context of network reachability, as following a number of exploits, many internet service providers choose to block port 445, which is the port used by SMB, as many exploits over the internet target this specific port, resulting in normal SMB traffic being blocked as well[11].

A possible solution to address the limitation present in TCP is the use of QUIC as an alternative transport protocol. Via combining QUIC's multiplexed transport, and SMB semantics for remote file access, this approach could mitigate the adverse effects of HOL blocking, improve latency of the connection and improve deployment to users space applications. Additionally, since QUIC is design with mobile users in mind, it could improve the reliability of the SMB protocol in diverse mobile environments, especially for mobile users. By moving from the traditional TCP port 445 to UDP port 443, SMB over QUIC in essence would mimic normal web traffic, circumventing the blocking of SMB traffic that is commonly done for TCP.

The objective of this thesis is to design, implement and benchmark a QUIC transport layer for an SMB server. The aim is to develop a working prototype that can be used to test performance, interoperability and compare against standard SMB over TCP solutions. In order to develop this solution, the thesis will implement a QUIC transport layer. This will be accomplished by using the MsQuic library[12]. This QUIC transport layer will the be integrated into Fusion SMB, an SMB server developed by Tuxera[13]. The thesis will perform a series of performance benchmarks and interoperability tests, focusing on throughput and compatibility.

This thesis is limited to supporting a QUIC transport layer using MsQuic. Other potential QUIC libraries are beyond the scope of this thesis, as there is no unified interface between the libraries, resulting in the work done to get one library working is not transferable to another library. The performance evaluations are limited to different throughput tests, including several different workloads, representative of real world enterprise and customer use cases. Other tests that will not be considered is latency and benchmarking connection creation, as the only available client is the Microsoft Windows SMB over QUIC client, which is limited in the number of connections. Questions such as large scale deployments, multi-node setups and integration into cloud architecture remains outside the scope of this thesis.

1.1 Thesis structure

The rest of this thesis is structured as follows. Chapter 2 provides background on internet transport protocols and network storage protocols. Chapter 3 introduces QUIC, the motivations for its creations and it architecture. Chapter 4 gives and overview of the SMB protocol. Chapter 5 describes the design and implementation of a QUIC transport layer for the Fusion SMB server. Chapter 6 presents the test environment, workloads and the resulting performance numbers. Finally, Chapter 7 discusses the findings and possible future work.

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)[14]. 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[1].

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[1].

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

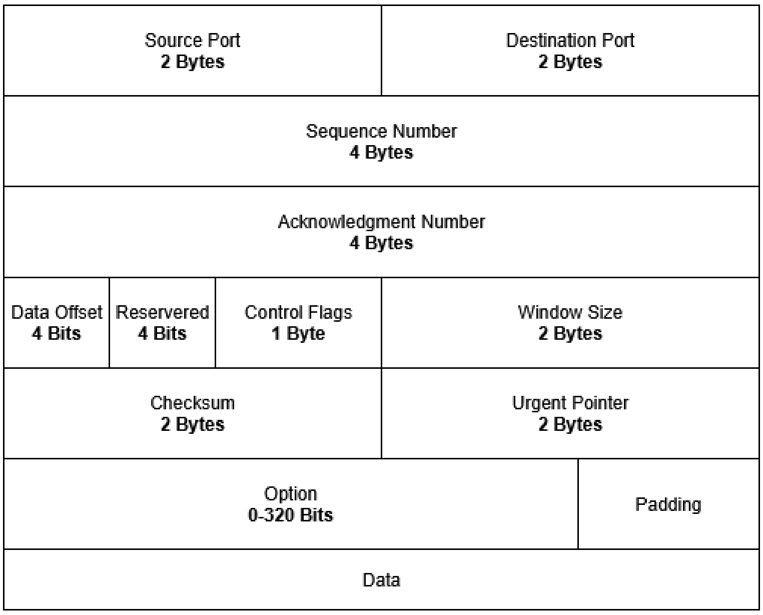


Figure 1: The TCP header

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

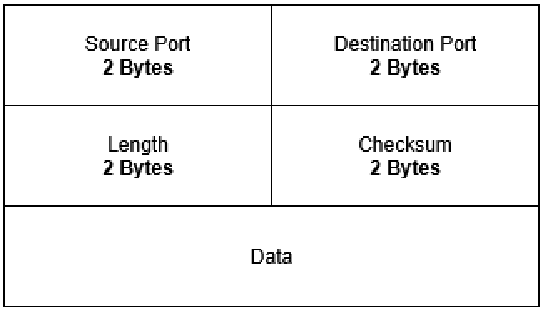
The 2 byte window field specifies the number of bytes that may be in-flight at any one time. This is the specified size of the sliding window that is used for flow control purposes. Following the window field is a 2 byte checksum field, used for detecting corruption of the TCP-header, data payload as 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[15].

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[1].

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[1].

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[14], 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[2]. 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[16]. The minimal UDP header (8



Source Port **2 Bytes**	Destination Port **2 Bytes**
Length **2 Bytes**	Checksum **2 Bytes**
Data	

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 Streams

A data stream is defined as a set of digital signals that is used to transmit information[17]. In practice, when talking about internet transport protocols, a data stream refers to a ordered, series of bytes that is used to transmit information. For TCP this holds true, even though TCP splits the transmitted data into packets, from the application point of view the information is a stream of bytes. In comparison, UDP does not offer this same abstraction, as each datagram exists independently of one another, but a series of these datagrams could be seen as a type of data stream, albeit an unordered one. Having only one stream per connection can lead to issues in the transport layer, such as HOL-blocking, which will be discussed in Section 3.1.1. The solution of opening multiple connections comes with drawbacks, of high overhead and resource consumption for the peers, with each connection taking up a separate socket on the client machine. As a result, protocols have emerged that aim to extend the data stream capabilities of TCP and UDP, and in the process solve some of the issues present in the legacy transport protocols.

2.2.1 SST

The Structured Stream Transport (SST) was introduced in [18] with the aim of extending TCP semantics, by introducing a hierarchical structure, allowing child streams to be created on top of an existing streams. These streams can be created without having to set up new connections, making them much more lightweight than TCP applications. The SST supports multiplexed streams, allowing the streams to exists in parallel, alleviating the issues of HOL blocking as each stream uses independent flow control. The SST also supports independent stream prioritization as well as reliable and best effort (unreliable) delivery[18].

There were three primary reasons for creating a new protocols, the SST. First, while TCP is better suited for long connection and large data transfers, UDP excels in short communication where the overhead of setting up connection via TCP's three way handshake provides significant overhead. Secondly, some protocols provides multiplexing on top of TCP, but suffers from HOL blocking as loss will block all data transfers until retransmission. Finally, some protocols such as HTTP/1.0 has bad utilization of network resources, as they are design to run on top of TCP and open a new TCP connection for each and every resource that is loaded. This was somewhat remedied in HTTP/1.1 with the serializing of requests over one connection, but at the time of the SST's creation it had yet to be widely implemented[18].

As can be seen in Figure 3, the SST architecture consists of three parts, the Stream, Channel and Negotiation Protocols. The Channel Protocol is responsible for the delivery channel providing the best-effort delivery service, congestion control and

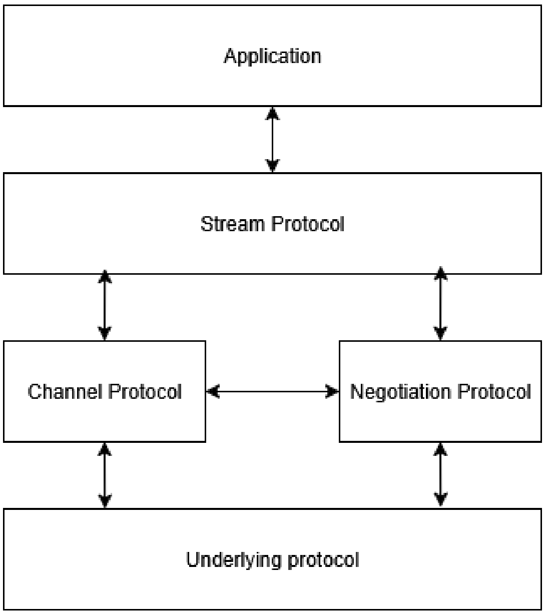


Figure 3: The SST architecture

packet sequencing. The Negotiation protocol is responsible for setting up the channel, and negotiating protocol details. Most interesting is the Stream Protocol, building on top of the other protocols to implement the Stream structure that SST is based upon. SST implements the concept of child streams. Child streams are created on top of a parent streams, but are otherwise equivalent to the main stream in all but name. Child streams allow multiple streams to exists on top of one connection, and they are design to be lightweight. This mitigates the issue of HOL blocking between stream, as loss on one stream still enables the other streams to keep transmitting while the erroneous stream independently recovers. SST also implements a second type of stream, while the normal stream is semantically similar to TCP, the ephemeral stream is closer to UDP. Ephemeral streams allows the application to have short communications as part of the larger connection, without any additional overhead[18].

2.2.2 SCTP

The Stream Control Transmission Protocol (SCTP), was first defined in RFC 4960[19], and which was then later replaced by RFC 9260[19]. SCTP was originally implemented to carry Public Switched Telephone Network (PSTN) messages, but was later standardizes to a general purpose protocol. SCTP is a connection based protocol, with connections referred to as association, and is designed to be ran on top of IP. As a contrast to TCP, which is heavily focused on transporting a byte stream, SCTP focuses on delivering messages between application. A message is sent on one end in one operation, and the whole message is received on the other end, in one operation. A series of these messages then form the stream inside SCTP. SCTP natively supports multiplexing of these streams, addressing TCP's issue of HOL blocking. This is a significant improvement for PSTN messagingn, where control messages must not be delayed by data loss in other streams[20].

In SCTP user messages are split into SCTP DATA chunks, which are bundled together with SCTP Control chunks together with a common header to form the SCTP packet. While there may be multiple chunks inside one SCTP packet, each chunk only contains the data of one message. Multiple sequences, streams, of these chunks may be sent in parallel over one association, enabling multiplexed streams. SCTP support both ordered and unordered message streams, allowing messages to potentially be delivered out-of-order, which may be preferable for certain kinds of data, such as control data[20].

SCTP supports multihoming, allowing the protocol to simultaneously bind to multiple different addresses and ports. This improves resilience, allowing the traffic to migrated to a backup path in case the primary network path is experiencing issues. The migration allows continued communication without breaking the association[20].

2.3 HTTP and its evolutions

Table 1: Comparison of HTTP Versions

Version	**HTTP/1.1**	**SPDY**	**HTTP/2**	**HTTP/3**
Transport	TCP	TCP	TCP	QUIC
Multiplexing	Pipelining	Over TCP	Over TCP	QUIC multiplexing
Compression	No	SPDY specific	HPACK	QPACK
Server Push	No	Yes	Yes	Yes

The HyperText Transfer Protocol (HTTP), is a protocol stemming from the early days of the World Wide Web, defining a protocol that would allow a server and a client to talk to each other. HTTP allows a client to request different resource from a web server, may that be javascript, css, images, multimedia files or something else. The high-level idea of HTTP is that a client request a resource from the server, and the server, if possible and permitted, responds with the requested resource. HTTP defines the formats of these messages, which are written in American Standard Code for Information Interchange (ASCII), making the message human readable directly on the wire, without having to interpret them. There are a set of HTTP request methods defined for the message, such as the GET method for requesting a resource, and POST for posting data to the server[16].

A comparison of HTTP version can be seen in Table 1. The first version of HTTP, HTTP/1.0 was defined in RFC 1945[21]. HTTP/1.0 used non-persistent connection, and had no concept of multiplexing. It was a fairly basic protocol, working on a request-response principle, with each request opening its own TCP connection. HTTP/1.1 defines three request methods. GET to request data from the server, POST to post data to the server and HEAD, which is similar to GET, but request that only the HTTP header be returned in the response, not the architectural body of the response[21]. What is notable for HTTP/1.0 is that it actually did not suffer from HOL-blocking of TCP, like most of its successors. Because each request opened a new TCP connection, there was not concept of HOL blocking as the requests were entirely independent of one another. This was however not a very efficient resource usage, as perform multiple three-way TCP handshakes to load a simple web page added significant overhead.

HTTP/1.0 was iterated upon and became HTTP/1.1, which was initially defined in RFC 2616[22]. HTTP/1.1 improved upon HTTP/1.0, adding support for several new features. First, persistent connections allow multiple requests to be sent over one connection, cutting down on the number of connections that need to be opened to load a set of resources. Second, pipelining allows multiple request to be made, without having to wait for the previous request to return. This is not quite yet true multiplexing, as the responses need to be returned in the same order as the request were made, causing HOL blocking of subsequent requests in case of loss[22]. Third, with the Introduction of HTTP over TLS, HTTPS, it was now possible to set up more secure, encrypted communications, in stead of transmitting data in cleartext[23].

2.3.1 SPDY

The SPDY protocol was designed by Google to improve latency of web pages, as compared to HTTP/1.1. The SPDY protocol was designed work as an extension to HTTP/1.1, slotting into the session layer and largely defining how HTTP traffic is sent over the wire, with somed additional improvements to the HTTP protocol. In practice SPDY introduced three improvements, multiplexed streams, header compression and server push capabilities[24].

SPDY introduced multiplexing of HTTP streams, allowing for an unlimited number of multiplexed streams over one TCP connection. As compared to HTTP/1.1's pipelining, the true multiplexing of SPDY allows the server to responds to requests in any order, instead of being forced to follow the request order. This allowed for a second improvement, where a client could attach a priority to the requests, signaling to the server which resources should be prioritized, especially in congested channels. To limit the number of bytes that have to be sent over the wire SPDY implemented HTTP header compressions, sending the header in a compressed format instead of ASCII. Additionally SPDY added server push and server hint features. Server push allows the server to send a resource to the client, without the client having request the resource first. Server hint is similar to server push, but instead of sending the resource, the server sends a hint to the client with a suggestion that the client should request a specific resource[24].

2.3.2 HTTP/2

HTTP/2 is standardized in RFC 9113[25], is the evolution from HTTP/1.1. It implements some of the ideas from SPDY, with the original specification for HTTP/2, RFC 7540[26], acknowledging the contributions made by the SPDY team. The HTTP/2 protocol uses the same semantics as HTTP/1.1, but like SPDY improves on the transport of the HTTP data between endpoints. HTTP/2 implements streams and stream multiplexing, stream prioritization, header compression and server push capabilities. Likey SPDY, HTTP/2 support multiplex, independently mutiplexed streams, allowing multiple parallel requests over one TCP connection. Stream priority allows the client to signal which requests are more important to be filled, and in case no priority is given, the server may choose to determine priority based on other information[25]. For header compression, HTTP/2 implements a dedicated compression algorithm called HPACK, defined in RFC 7541[27]. HPACK not only implements compression, but the algorithm also helps in reducing redundant header fields, by preventing the same static header fields from being sent over and over again[27]. HTTP/2 supports server push capabilities, allowing the server to send unprompted resource to a client, usually by estimating the requests that the client likely will make in the future, in that way improving latency[7].

Even though HTTP/2 implements stream multiplexing, it still suffers from HOL blocking simply due to the fact that HTTP/2 streams are created on top of a single TCP connection. This means that the requests wont block each other, but in case of packet loss, the TCP HOL blocking will cause all requests to stall while waiting for [25]. This highlights the fact that despite improvements in the HTTP protocol, the HOL blocking issue may not be solved as long as the protocol is running on top of TCP.

2.3.3 HTTP/3

HTTP/3, as defined in RFC 9114[7], is as of writing the final evolution of the HTTP protocol. It was specifically designed to be ran on top of QUIC, taking advantage of the improvments offered by the QUIC transport protocol. Like HTTP/2, the HTTP semantics stays largely the same, but improvements are made to take advantage of the new transport. HTTP/3 includes support for transport level multiplexing, transport level encryption and connection mgiration. The most significant advantage offer by HTTP/3 is the improvements to transport level HOL-blocking. Because HTTP/3 takes advantage of native QUIC streams for its multiplexing, which are entirely independent on the transport level, even packet loss wont affect unrelated streams. This allows HTTP/3 to take full advantage of multiplexing requests to load resources, without bottlenecking from the transport layer. QUIC offers an improved conenction setup time as compared to HTTP/2, as HTTP/2 uses TCP + TLS, requiring multiple RTTs to establish a secure connection. HTTP/3 utilizes the fact that QUIC has the TLS hanshake integrated into the QUIC transport handshake, potentially reducing connection setup time to 1 RTT. In some cases it is even possible to use 0-RTT data transfer to instantaneously transfer data wihtout waiting for the conneciton to be set up. Connection migration of QUIC allows connection to stay active for mobile users, even in the face of network changes[7]. As with HTTP/2, HTTP/3 uses header compression, but HPACK has been replace by QPACK, which improves on HPACK by better taking advantage of QUIC's features, improving HOL blocking issues with only a slighty degradation in compression rate[28].

2.4 TLS

The Transport Layer Security (TLS) protocol is an evolution from the Secure Socket Layer (SSL)[29] protocol. TLS aims to secure communication between two endpoints.

The TLS protocol provides three properties to the communication. First authentication, with the server side of the communication always being authenticated, and the client side optionally being authenticated. Second confidentiality, TLS ensures that data sent between the endpoints is encrypted and ensured, ensuring that only the intended recipient is able to decrypt and read the data. Finally integrity, data transmitted through a TLS tunnel cannot be tampered with, wihouth alerting the recipient[30].

The TLS protocol consist of two components, First, the hanshake protocol is used to establish the connection, being responsible for authentication, negotiation and creating the key material used for encryption. Second, the record protocol is responsible for encrypting and transmitting the data. The name comes from the fact that the data to be transmitted is split into a series of records, all of which are individually encrypted and tranmitted. The TLS protocol is designed to be a general purpose security protocol, as it only requires to be ran on a reliable transport, and in principle any application protocol may be ran on top of TLS[30].

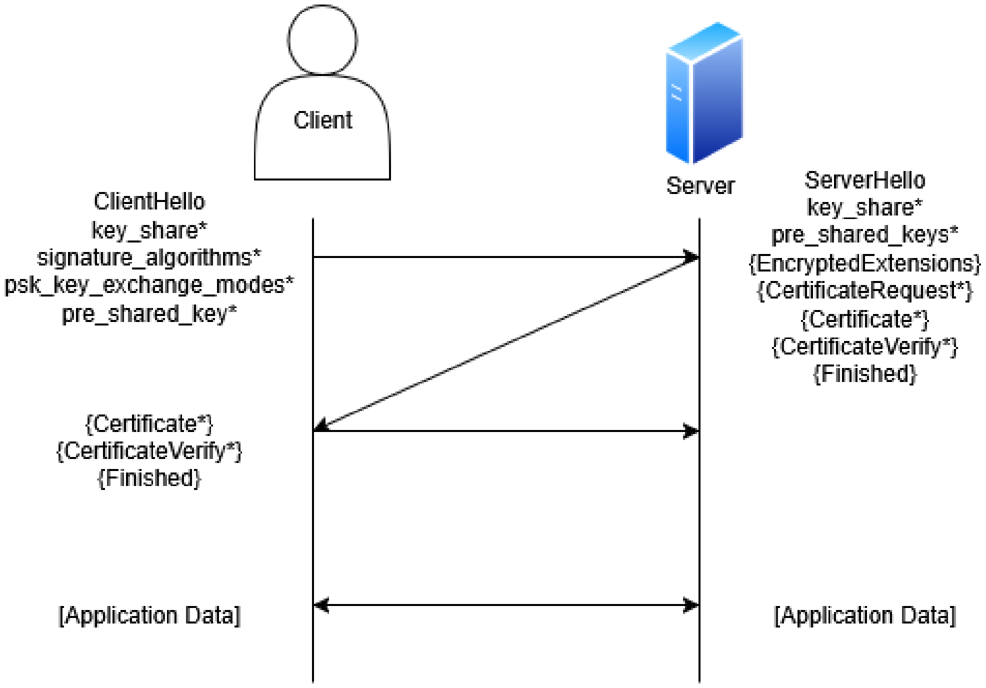


Figure 4: The TLS 1.3 Handshake

The TLS hanshake is the most important part when establishing a TLS session. In TLS 1.2 and earlier the hanshake required 2 RTTs before application data could be sent securely, but since TLS 1.3 the hanshake has been streamlined into 1-RTT, with the possibility of 0-RTT data transfer, at the cost of perfect forward secrecy. The TLS 1.3 hanshake is outlined in Figure 4 where the asterisk indicated optional or alternative parts of the message, curly brackes indicate handshake secret protected and square brackets indicate full application protection of data. The handshake begins with a ClientHello message, containing a nonce, the protocol version and either a pair of Diffie-Hellman key share or a set of pre-shared key labels. The server responds with a ServHello, containing key shares, certificate if it used in authentication of the server and possible extensions, such as CertificateVerify request if the server requests the client to be authenticated via a certificate as well. The certificate and CertificateRequest as well as Extensions are protected with keys derived during the handshake process. These are different from the secrets used to protect application data and does not guarantee perfect forward secrecy. The combination of ClientHello and ServerHello is used to create the shared keys. Finally the client responds, acknowledging that the process is finished[30].

Normally the hanshake adds additional overhead to connection setup. With TLS 1.2 and lower adding an additional 2+ RTTs of latency, and TLS 1.3 adding 1-RTT of latency. However the QUIC protocol integrates the TLS 1.3 hanshake into the transport layer protocol, allowing both hanshake to be made in parallel, negating the latency impact, but more on this in Chapter 3. Once the hanshake is finnished and the TLS session setup, the record protocol takes over. The record layer splits the application data into appropriate chunks, which are then encrypted and integrity protected. And transmits these via the underlying, reliable transport.

The first versions of TLS, TLS 1.0 and 1.1, defined in RFC 2246[31] and RFC 4346[32], improved upon SSL but were deprecated in 2021 via RFC 8996[33], due to the fact that these algorithms rely on weak cryptographic algorithms. The TLS 1.2 protocl improved upon its predecessor by introducing more secure cryptographic algorithms and authenticated encryption[34], and removing backwards compatibility with SSL via RFC 6176[35]. Finally, as mentioned earlier in this section, TLS 1.3 is the latest version of the TLS protocol. It removed support for legacy cryptographic algortihms, added the 1-RTT hanshake and 0-RTT data transfer, among other cryptographic improvements[30].

2.4.1 DTLS

As mentioned in Section 2.4, the TLS protocol is designed to run on top of a reliable transport. However, it is quite common for application to run on top of UDP, which as mentioned in Section 2.1.2, makes no guarantees for reliability. To solve this problem and provide protection for datagram based applications, Datagram Transport Layer Security (DTLS) is designed to overcome the challenges of securing an unreliable transport. There are four main reasons the TLS cannot be used directly on top of UDP. First, TLS uses an implicit sequence number as a nonce, meaning that TLS records being delivered out of order causes the decryption process to fail. Second, the TLS hanshake is rigid, the right messages has to arrive in the right order for the hanshake to succeed. Third, handshake messages may be larger than one datagram, causing fragmentation. Finally, UDP is susceptible to Denial-of-Service (DoS)[36].

To handle packet loss, DTLS uses a retransmission timer to enable retransmission of lost packets, such as part of the handshake. Reordering is mitigated by assigning an explicit packet number, enbaling the receiving party to process the received messages in the correct order. DTLS adds support for fragmentation of handshake message, the sending part can split the handshake message into multiple parts that all fit inside a datagram, and the receiving party can the reassemble the hanshake message from the fragments. To combat DoS attacks, DTLS implements a HelloRetryRequest message, which contains a cookie, that then has to be included in the ClientHello message to be accepted by the server. Optionally, DTLS includes replay detection and protection[36].

DTLS is designed to be as close to TLS as possible in capabilities and funcitonalities. DTLS follows TLS versioning, with the latest, DTLS 1.3, being defined in RFC 9147[36]. Like TLS 1.3, DTLS 1.3 implements the improved 1-RTT handshake, enforces stronger cryptographic algorithms than earlier versions and adds support for 0-RTT data transfer, however 0-RTT data transfer requires that replay protection is enabled on the connection[36].

2.5 DFS

Distributed File Systems (DFS) allow clients to communicate with servers, enabling the clients to manipulate the files as if they were stored on their local machine. DFS's abstract away the complexities of distributed data access, exposing a set of file operations to the clients, allowing them to among other create, delete, read and write files. The DFS ensures consistency, reliability and access between different clients and access points[37]. Currently, the most widely adopted distributed file systems is the Server Message Block (SMB)[3] and the Network File System (NFS)[38]. Both protocols provide file system access over a network, but they differ significantly in design, environment and features.

2.5.1 Server Message Block

The Server Message Block (SMB) protocol is a stateful protocol, and is a protocol designed for sharing files printers and other resources over a network. It was originally created in the early 1980s, and later adopted and extended by Microsoft[39]. The protocol has undergone several iterations since that time. While the underlying concepts of SMB has stayed the same, later revisions, that is SMB version 2.0 and forwards, differs heavily from earlier version SMB 2.0 introduced different packet formats, and reduced the chattiness of the protocol, among other improvements. SMB 3.0 introduced encrypted trafic, multichannel support and integration with RDMA in the form of SMB Direct. Since SMB 3.1.1 the protocol has added support for using the QUIC protocol as a transport[3].

The main distinguishing characteristics of the SMB protocol is its stateful design. The server maintains information about the state of a session, such as file handles, locks and leases over the lifetime of a session. This allows the SMB protocol to guarantee consistency and helps in coordinating multiple, concurrent clients. As the development of the SMB protocol has largely been a result of Microsoft's effort, the protocol is deeply integrated into Windows authentication and authorization, supporting NTLM and Kerberos authentication, and integrates as part of Active Directory Domain services[3].

2.5.2 Network File System

The NFS protocol was introduced by Sun Microsystems in 1984, aiming to be a protocol for enabling remote access to filesystem between different UNIX machines. NFS was designed to be a simple and open filesystem, making it easy to port to different machines.In contrast to the SMB protocol, the NFS protocol was designed to be a stateless protocol. This simplified processing, as each request contained all necessary information to complete the request, and in case the request failed the client could simply try again. Recovery from error states was simple as no sessions had to be reestablished. The sateless design came with drawbacks, for example two concurrent clients trying to write the same file could cause the write to be interleaved, causing unintended outcomes, as the writes would simply be processed in the order they arrived[40].

Over time the NFS protocol has evolved, with the latest version NFSv4.2 being defined in RFC 7862[41]. NFSv4 introduced a stateful protocol, and introduced features such as file locking, multipath and compound operations, alongside performance improvements and improved authentication support for stronger security[38]. NFSv4.1 introduced NFS sessions and parallel access to data, as well as improving support for clustered solutions[42], while NFSv4.2 introduces support for server side operations, and sparse files[41].

Table 2: Comparison of SMB and NFS

Protocol	**SMB**	**NFS**
Environment	Windows	UNIX
Resources	Files, directories, printers, network resources	Files and directories
Sessions	Stateful	Stateless v2/v3, Stateful v4

Table 2 gives an overview of the major differences of SMB and NFS. SMB provides many different features, a natively stateful protocol for sharing all kinds of resources, and a solution that is very Windows centric, with UNIX based system requiring a third party solution for support, such as Fusion File Share or Samba. In comparison, NFS is a lightweight, easy to setup protocol that is purely focused on file and directory sharing, and which is heavily integrated into many different UNIX based operating systems.

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[15] and combining TCP and TLS in HTTPS[23] 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[4].

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[5]. 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[4].

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[25], 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[43].

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[4].

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[30]. 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[44].

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[4].

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[45]. 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[4].

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[5]. In the end the protocol was standardized in a number of RFCs. RFC 9000[5] defines QUICs core transport mechanics, RFC 9001[6] defines the use of TLS 1.3 and RFC 9002[46] describes the loss detection and congestion control algorithms used by the protocol. In addition, RFC 8999[47] 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[48]. Both Firefox[49] and Safari[50] 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[4].

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[5]. 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[6]. The final layer exposes a standardized application interface that can support virtually any application-layer protocol, the most prominent one being HTTP/3[25].

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[5].

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[5].

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 5, 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

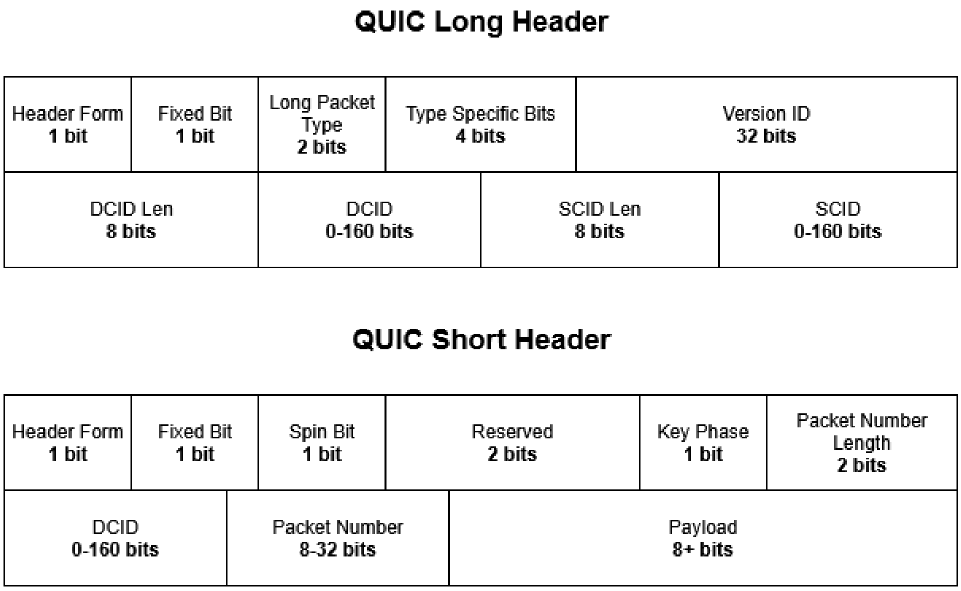


Figure 5: The QUIC header formats

Table 3: 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[5].

The long header packet type numbers differs between QUIC version 1 and version 2, as is outlined in Table 3. 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[5].

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[5].

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[5].

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[5].

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[5]. 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[7]. 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[5]. 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[46].

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[51] 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[52]. 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[46].

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[46].

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[46].

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[6]. 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[5].

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[5].

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[5].

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[5].

3.7.2 Connection Lifetime

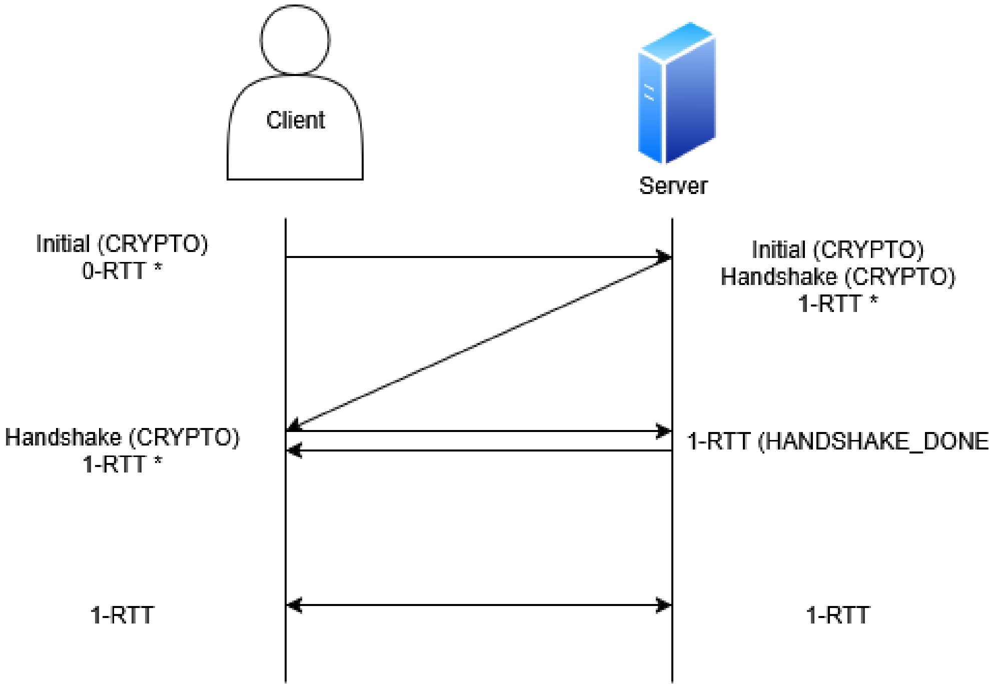


Figure 6: The QUIC Handshake

On of the main innovations made by the QUIC protocol is the integration of the TLS 1.3 cryptographic handshake into the transport handshake, improving the latency of connection setup and encrypting the connection from the beginning. As can be seen from Figure 6, in an 1-RTT hanshake, the connection is inniated by the client sending an Initial packet, which contains a TLS ClientHello message, which is contained inside a CRYPTO frame. The server responds with an Initial Packet containing the TLS ServerHello message, and a Handshake Packet containing EncryptedExtensions, as well as its certificate and possibly the CeritifcateVerify message in case the client should be authenticated as well. The client then responds with a Handshake message containing the TLS Finished message, which signals the completion of the hanshake, and application data can now flow securely[5, 6].

It is possible to transmit data before the cryptographic handshake has been complete, albeit with some drawbacks. In Figure 6 the opportunities for early data transmisison has been marked with an asterisk. Firstly, 0-RTT data transfer is possible, in case the client is previously known to the server, and has received a TLS resumption ticket, data may be sent during the Initial Packet from the client. 0-RTT data transfer comes with the caveat that it is susceptible to replay attacks. It is also possible for the server to derive the encryption keys, early and transmit data alongside its initial hanshake message. This mechanic may be refered to as "0.5-RTT", but the data is carried inside a 1-RTT packet. This has the drawback of transmitting data before the client has authenticated, which means that the server should avoid transmitting sensitive data during this transfer. Similarly, once the client has completed the cryptographic hanshake it may already start transmitting data alongside the TLS Finished message[5, 6].

There are three ways in which a QUIC connection may be terminated. First via an idle timeout. The idle timeout is configured in the transport parameters, via the max_idle_timeout. If both endpoints announces a max_idle_timeout, the lower of the two will be used. The idle timeout is reset when receiving a packet from the peer, or alternatively sending a packet to the peer. This means that and endpoint may defer the idle timeout, by sending a packet to the peer if it expects that data is still outstanding. If the idle timeout runs out, the connection will be silently closed, and the state discarded. Second, an endpoint may explicitly shut down a connection by sending a CONNECTION_CLOSE frame. This also immediately closes all open streams, putting the peer that sent the CONNECTION_CLOSE frame into the closing state, and the peer that received the CONNECTION_CLOSE into the draining state. This is to enable the peers to cleanly shut down their connection, and properly handle packets that arrive out of order. Finally, a connection may be shut down via a stateless reset. This is used in case the peer does not have access to the state of the connection, and as a result is unable to process incoming packets. This sends a Stateless Reset Token, which is associated with a connection ID, causing the peer to immediately reset the connection.

3.7.3 Connection Migration

The QUIC protocol adds support for connection migration. In legacy protocols such as TCP, the connection is identified by the touple pair of source and destination IP addresses and ports. If something happens that causes any of this information to change, for example changing from a local network to a mobile, the connection will be closed and the state discarded. As discussed in Section 3.7.1 one of the motivation for adding connection IDs was to improve the resilience of connections, allowing the connections to survive changes in network. The connection IDs are selected by the endpoints, and allows the packets to be correctly routed, even if the IP address or port of either endpoints changes for any reason[5].

As changes in network port or address may be involuntary, which might happen as a result from NAT rebinding or network change, there may be a situation where either endpoint detecs a change to its peer's address or port. The endpoint must then perform path validation. Path validation is done via sending a PATH_CHALLENGE frame, containing a random or hard to guess payload. The peer receives the PATH_CHALLENGE, and responds with a PATH_RESPONSE frame, echoing the payload from the PATH_CHALLENGE frame. In this way the path is validated one way. The peer may include its own PATH_CHALLENGE frame alongside the PATH_RESPONDE, to validate reachability the other way[5].

Conneciton migration may also be initialized explicitly, by sending packets from a new local address. In this case both peers must ensure reachability on the new path, by using path validation as described earlier in this chapter. When an endpoint receives packets from a new address, it must validate the path and subsequently send packets to the new address, using a new connection ID, to prevent on wire tracking of the connection migration. In practice, on the receiving end the process is the same for explicit and implicit migration. A server will not initialize a connection migration, this is exclusively done by the client. What a server may do, is a preferred address. This allows servers to accept incoming connection on one address, and then quickly transfer the connection to a separeate, prefered address, performing path validation against the client from the new address[5].

4 The Server Message Block protocol

The main purpose of the SMB protocol is to share files and directories over a network. The SMB protocol is a stateful protocol, where clients initatiate connections, an authenticated session is created and requests are sent over the connection, allowing the client to performan file operations, but also additionally access other resources such as printers. There currently exists 2 major versions of the SMB protocol, the original SMB protocol, sometimes refered to as the CIFS protocols, which later evolved into the SMB 2 Protocol, that encapsulates SMB Versions 2 and 3[53]. reference to the SMB protocol in this and following section will refer to the SMB 2 protocol, and the legacy SMB protocol will be referenced as SMB 1 for clairy, however it will not be covered by in depth by this thesis. This section of the thesis will give a technical overview of the SMB protocol.

Table 4: Comparison of SMB protocol dialects

Dialect	Introduced in	Key Features
SMB 1.0	2000	Basic file and printer sharing
SMB 2.0	2006	Reduced chattiness, new packet format, support of symbolic links
SMB 2.1	2009	Opportunistic locks, minor performance enhancements
SMB 3.0	2012	Encryption of traffic, multi-channel support, SMB Direct (RDMA), transparent failover, Directory Leasing
SMB 3.0.2	2013	Unbuffered read and write
SMB 3.1.1	2016	Support QUIC as a transport, RDMA encryption, Improved cryotography support

Within the larger versions of the SMB protocol there exists minor versions, referred to as dialects. Within the SMB 2 protocol the following dialects exists, SMB 2, SMB 2.1, SMB 3.0, SMB 3.0.2 and SMB 3.1.1. The basic funcionality of the SMB protocol was already defined in SMB 1, allowing clients to connect to shares and perform file operations. As can be seen in the overview given by Table 4, the protocol has been continously improved by then, adding support for, among other fatures, encryption and alternative transports[53].

The SMB protocol places itself in the application layer, it does not stand on its own but relies on other protocols for some of its functionality. To enable authentication of clients, the SMB protocol uses the Simple and Protected GSS-API Negotiation (SPNEGO), as defined in RFC 4178[54]. The SPNEGO protocol exposes a common interface that may be used by for authentication with the help of other authentication protocols. In the case of the SMB protocol the SPNEGO protocols rely on Kerberos

Authentication, as defined in RFC 4120[55], or the New Teachnology LAN Manager (NTLM) protocol, as defined in [MS-NLMP][56].

The SMB protocol is designed to be ran on top of a reliable transport. historically this was NetBIOS over TCP, later standardized to the more common TCP. As outlined earlier in this section, later versions of the SMB protocol supports alternative transport. The SMB dialects >= 3.0 supports use of RDMA via SMB direct, for improved performance. The latest version of the SMB protocol, 3.1.1 added support for the QUIC protocol, which comes with increased security guarantees and potentially improved performance for mobile applications[53].

4.1 SMB message structure

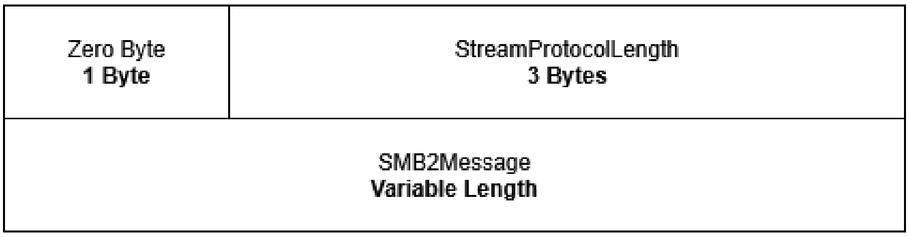
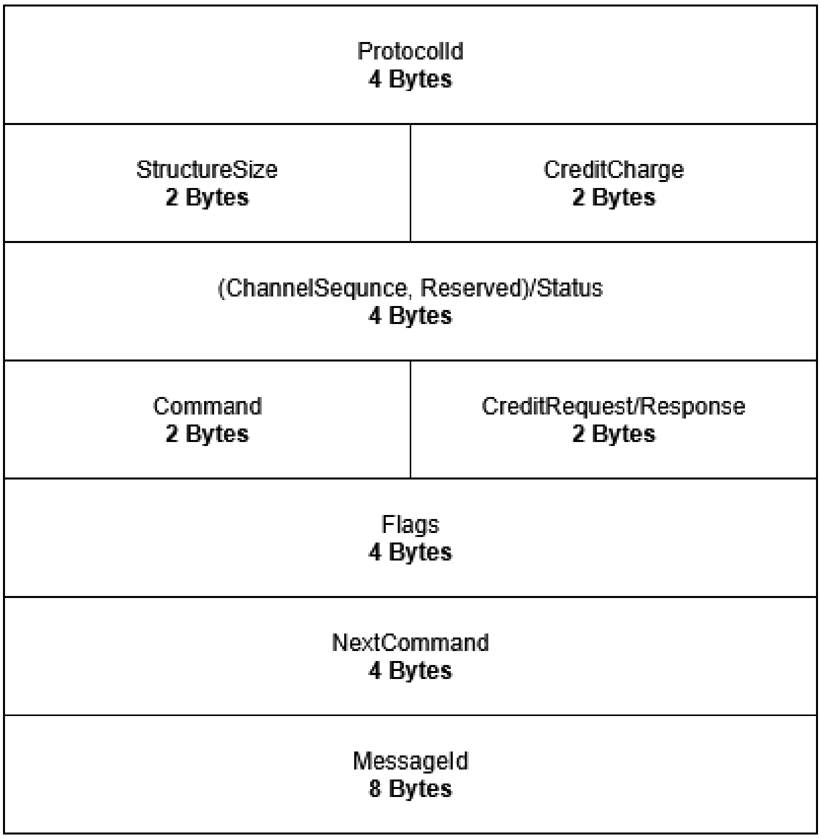


Figure 7: The SMB Direct TCP header

The SMB protocol uses the Direct TCP packet header, as shown in Figure 7. It begins with a zero byte, followed by a 3 Byte StreamProtocolLength field which indicates the length of the SMB2 message. Following this is the actual SMB2 message. The first part of the SMB2 Message contains the SMB2 packet header. There are 2 versions of the header, an aynchronouns and synchronous header, for asynchronous and synchronous requests respectively. The first 32 bytes of the SMB2 packet header is common for both request types, as may be seen from Figure 8. The SMB2 Transport Header begins with the 4 byte Protocol Id field, which is always set to 0x424D53FE, or when converted to ASCII and in network byte order, 0xFE, S, M, B. The following field is the StructureSize, which is set to 64, which is the total size of the header. Next is the CreditCharge, this indicates how many credits the request consumes, more on the SMB credit system later in this chapter. Following this is a field that is interpreted differently for the client and the server, and depending on dialect. On the client side, if the dialect is at least SMB 3.0, this field indicates Channel, for dialects < 3.0, this field is reserved. From the Server side this field indicates the status that in response to any request. The 2 byte command field is used to indicate the command of the request or response, Table 5 contains a list of all possible SMB2 commands. Following this is the 2 byte credit request and response field, which contains information about requested credits. The flags field contains flags for determining how the packet should be processed. In the SMB protocol it is possible to concatenate multiple requests into one packet. This is then indicate vai the NextCommand field, that indicates the offset to the next request. Finally, the MessageId is the unique identifier that identifies the specific request and its corresponding response. In addition, the asynchronous



ProtocolId
4 Bytes

StructureSize 2 Bytes	CreditCharge 2 Bytes

(ChannelSequnce, Reserved)/Status
4 Bytes

Command 2 Bytes	CreditRequest/Response 2 Bytes

Flags
4 Bytes

NextCommand
4 Bytes

MessageId
8 Bytes

Figure 8: The SMB Direct TCP header

header contains an AsyncId for identifieng the asynchronous, operation, a SessionId that identifies the session, and in case the message is signed, the signature. The synchronous header also contains the signature and messageId, but instead of the AsyncId the synchronous header contains the TreeId, more on this later. Following the SMB2 packet header is the SMB2 Protocol Message, which has a variable legnth depending on the command code and if it is a request or a response[53].

4.2 SMB connection lifetime

The SMB connections beginns by establishing a connect via the transport protocol of choice, as discussen in Section 4, this may be TCP, RDMA or QUIC. Once the connections is established the first message sent from the client is the SMB2 NEGOTIATE message. The client sends an array of supported dialects, and in response the server selects the greates common dialect that is supported by both. In the SMB2 NEGOTIATE response the server also indicates its capabilities, such as transport encryption in the case of QUIC transport. Once the dialect and capabilities has been negotiated, the next step in the process is authenticating the user. This is done via the SMB2 SESSION_SETUP message. As dicussed in Scetion 4, this is done via the SPNEGO protocol, and the underlying protocol used for authentication is either NTLM or Kerberos. For kerberos authentication the client authenticates against a domain

Table 5: SMB Commands

Name	Value
SMB2 NEGOTIATE	0x0000
SMB2 SESSION_SETUP	0x0001
SMB2 LOGOFF	0x0002
SMB2 TREE_CONNECT	0x0003
SMB2 TREE_DISCONNECT	0x0004
SMB2 CREATE	0x0005
SMB2 CLOSE	0x0006
SMB2 FLUSH	0x0007
SMB2 READ	0x0008
SMB2 WRITE	0x0009
SMB2 LOCK	0x000A
SMB2 IOCTL	0x000B
SMB2 CANCEL	0x000C
SMB2 ECHO	0x000D
SMB2 QUERY_DIRECTORY	0x000E
SMB2 CHANGE_NOTIFY	0x000F
SMB2 QUERY_INFO	0x0010
SMB2 SET_INFO	0x0011
SMB2 OPLOCK_BREAK	0x0012
SMB2 SERVER_TO_CLIENT_NOTIFICATION (ASYNC only)	0x0013

controller, which issues a ticket to the client that can be used for authentication. In case Kerberos authentication is not available, the client may authenticate via NTLM. This works by the server sending a challenge to the client, and using the client password as a shared secret, the response to the challenge is calculated, and sent back to the server, which verifies the response and authenticates the client[53].

Once the connection is established and the client is authenticated, it may now connect to a share exported by the server. This is done via the SMB2 TREE_CONNECT request. The client requests to connect to a specific share, identified with the path name in the form \\server\share. The server tries to find the requested share, and if successful and the client has appropriate access rights, allocates a specific tree connect object to store information abouth the session, and this objects is identified via a TreeId that is sent back to the client, alongside the share type, disk, pipe or printer. In addition, the server returns information about the share capabilities, suchs a continous availability or scaleout capabilities[53].

Having connected to the share the client is now able to perform the main meat of the transaction, that is file operations. The client can use SMB2 CREATE to create or open files, SMB2 CLOSE to close or delete files, SMB2 WRITE and READ to write and read to a file correspondingly, or SMB2 QUERY_DIRECTORY to list the available files. Once a file is opened it is identified by an opaque file handle, called an open, returned by the server. Subsequent file operations are then done via this handle, and the server and client store the associated information inside a table. As the SMB protocol is stateful, all the requests are associated with a certain session, and the requests are handled in accordance with the clients capabilities and authorization. To limit the number of outstanding requests any client has at any one time, the SMB protocol uses what is refered to as a credit system. Every client has a certain number of credits, that are consumed upon sending any request. The number of credits is equal to the size of the request, or the expected size of the response, whichever is larger, divided by 65536 plus 1. It is up to the client to request credits as they are consumed, and the server should grant credits when requested, to avoid a situation where the client reaches 0 credits, as without any credits the client cannot request any more credits and is then stuck[53].

When the client is finished with a share it is time to disconnect and log off. The disconnection is performed via a SMB2 TREE_DISCONNECT message, using the TreeId provided earlier when connection was setup to the share. If the client is done with its operations, it can log off from the server via the SMB2 LOGOFF message, allowing the client to cut all connections to the server. This section only covers the most relevant process and possibilities of using the SMB protocol to access a shared resource. For further reference the Microsoft Technical documentationcite[53] outlines all possible scenarios, and is available for futher reading into the technical specifics of the SMB protocol messages and functionalities.

4.3 SMB over alternative transports

The SMB protocol currently supports 4 transports, TCP, RDMA, QUIC and the deprecated TCP over NetBIOS. The standard transport in most scenarios is TCP, but the protocol is moving towards the use of alternative transports to improve performance and security. The SMB protocol supports RDMA via SMB Direct. The goal of RDMA is to allow remote direct memory access, that is reading from and writing to a remote memory buffer, withouth having to copy the data. This copy saves CPU cycles, and in that ways requires less processing power and improves throughput by using specific network adapters with support for RDMA[57]. Likewise, the goal of SMB Direct is to improve performance by utilizing RDMA and multichannel. SMB multichannel then allows the client to open multiple connections to the server, improving performance by transfering data in parallel. Encryption is supported since SMB 3.1.1 by encrypting the payloads before they are sent over the wire[58].

From the perspective of this thesis, the more interesting alternative transport is SMB over QUIC. As discussed in Section 3, the QUIC protocol brings many advantages over legacy TCP transports. QUIC enables transport-level encryption that leverages TLS 1.3 for actually secure cryptography, multiple logical streams to combat HOL blocking, moving the traffic to port 443 to avoid the common port blocking of SMB port 445 and connections migration. The biggest improvement to the SMB protocol is the inclusion of transport level ecnryption. While SMB version >= 3 supports per share based encryption, the cryptographic analysis is outside the scope of this thesis, but benefits of TLS 1.3, such as perfect forward secrecy is not part of standard SMB encryption[59]. SMB over QUIC supports negotiating transport level encryption in the transport capabilities of the negotiate protocol request. This allows the client and server to skip SMB level encryption, even though it is enabled on a share, allowing the communication to entirely rely on QUIC for secure communications[53]. Additionally SMB over QUIC supports a concept that is called client access control. As outlined in earlier sections, in TLS 1.3, during the handshake, there is the possibility for the server to request that the client provides its own certificate for authentication purposes. The server can then use that certificate to determine if the client should be able to connect to the share[60].

5 Implementing QUIC as transport for SMB server

The QUIC protocol, as outlined in Chapter 3, currently has no kernel level implementation in Linux, instead relying on user space libraries for support. This was a concious design decision for the protocol, allowing for different implementations and easier iterations and switching betweeen different stacks. There exists many different implementations of QUIC, such as lsquic developed by LiteSpeed, quiche developed by cloudflare and MsQuic developed by microsoft[61]. There is also an active effort to develop a version of QUIC for the Linux kernel, driven by Xin Long[62]. For the purposes of this thesis the MsQuic library was chosen as the library that is used to implement the QUIC transport layer. The reasoning behind this choice was twofold. Firstly, the MsQuic library shows promising performance numbers when compared to other libraries[63]. Secondly, as the only SMB over QUIC client available is the Microsoft client, it is advantageous to use the same implementation both in the server and the client, to ensure maximum compatibility. This section of the thesis will outline the architecture and API of the MsQuic library, as well as describe the design of a QUIC transport layer using the MsQuic library, and the process of integrating the QUIC transport layer into the Fusion SMB server.

5.1 MsQuic architecture and API

The MsQuic library is a cross-platform implementation of the QUIC transport protocol, implemented as a shared library written in C, with additional support for development in C++, Rust and C#. It is open source and licensed under the MIT-license. The library utilizes the Windows Secure Channel (Schannel) suite for cryptography implementations on Windows, and for Linux based machines the library uses a custom fork of the OpenSSL library to provide support for QUIC specific cryptography[12]. The library uses an asynchronous processing model, where the caller create callback functions and registers those with different MsQuic objects, such as listeners and connections. This callback handlers are then invoked to handle events that occur, such as receiving a new connection request or receiving data on a stream[64].

5.1.1 High level architecture

The MsQuic library abstracts away the concept of networking sockets, instead opting for a object based model, which can be seen in Figure 9. The MsQuic library model includes 5 main types of objects, the Registration, Configuration, Listener, Connection and Stream objects. The hierarchy of these objects can be seen in Figure 9. All function calls are done in relation to one of these objects with the exception of the call to intialize the library, which returns a function table of MsQuic functions, and the corresponding call to close the library[64].

The registration object represents the execution context for the transport, being responsible for the connection logic and creating the threads that are used for processing. All other objects are created under the registration, and each registration is completely independent from one another, with resources and scope not being shared between

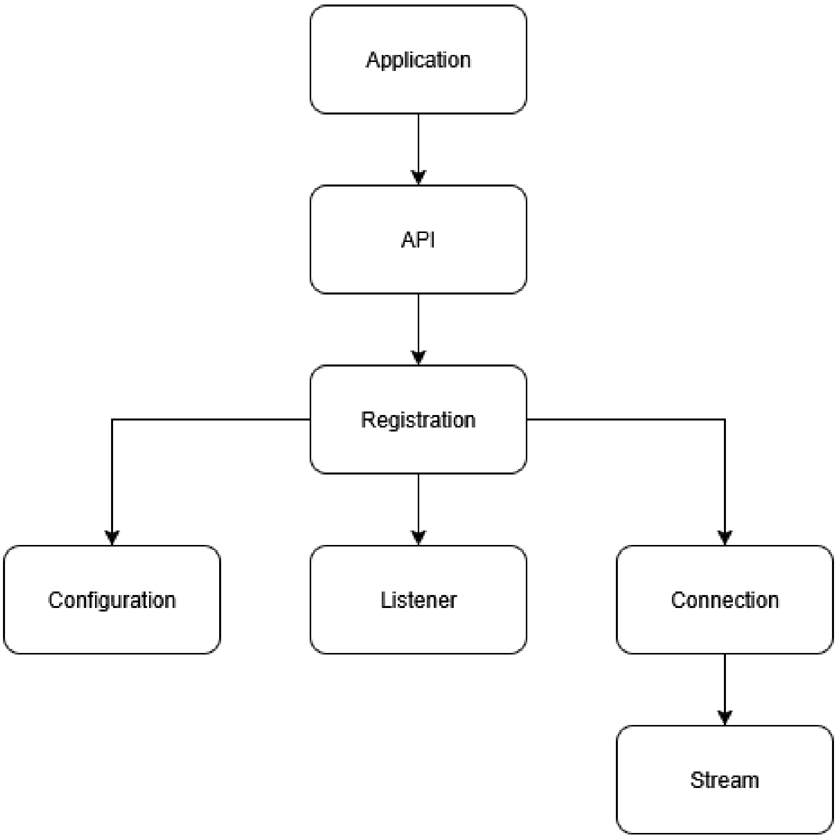


Figure 9: High Level MsQuic Object Model

different registrations. In practice this means that generally one application uses only one registrations[64].

The configuration object stores the settings that are used in configuring the MsQuic library. These include QUIC transport settings, as well as security configurations. Everything from intial receiver windows, idle timeout timers to the congestion algorithm used may be configured. The comprehensive list of configuration parameters may be found in the MsQuic documentation. In addition, each of the MsQuic objects have specific parameters that may be configured, and the certificate as well as its parameters is separately configured, for example configuring requesting a certificate from the client and allowed cipher suites. The configuration object also stores a list of Application Layer Protocol Negotiation (ALPN) IDs that the server supports[64]. For reference SMB over QUIC simply uses the ALPN ID "smb".

Following this is the listener object, which is responsible for listening for and accepting incoming connections. The listener object uses a callback function that is responsible for handling incoming connections. The callback function decides if the incoming connections is accepted, and if it is a connection object is created in response[64].

The Connection object represents the QUIC connection between two peers. On the client side this connection is explicitly created via a function call, while on the server side it is created in the listenere callback. The connection object contains information about the connection, and it is expected to handle events related to connection setup, stream creation and connection teardown, either as a result of an error or from one of the peers closing the connection[64].

The most important object type is the stream object, as it is responsible for the data transfer between the peers. As dicussed in Section 3, there can be a theoritical unlimited number of streams on top of a connection, but in practice the maximum number is configured in the MsQuic library, with separate limits for unidirectional and bidirectional streams. New streams can be created either by the client or the server, and with the other party being notified of the event. When sending data the MsQuic library takes temporary ownership of the buffer, copying the data internally and queuing the data to be sent. This allows the MsQuic library to "keep the pipe full", aiming to not let the stream sit unecesarily idle. It is also possible to disable this internal buffering, instead making the caller responsible for queuing enough sends to prevent idleing the connection[64].

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

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

[3] Microsoft. *[MS-SMB2]: Server Message Block (SMB) Protocol Versions 2 and 3*. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-smb2/5606ad47-5ee0-437a-817e-70c366052962 (visited on Aug. 26, 2025).

[4] 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.

[5] 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.

[6] 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.

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

[8] S. Cook et al. "QUIC: Better for what and for whom?" In: *2017 IEEE International Conference on Communications (ICC)*. 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7997281.

[9] T. Shreedhar et al. "Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads". In: *IEEE Transactions on Network and Service Management* 19.2 (2022), pp. 1366–1381. DOI: 10.1109/TNSM.2021.3134562.

[10] K. Nepomuceno et al. "QUIC and TCP: A Performance Evaluation". In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. 2018, pp. 00045–00051. DOI: 10.1109/ISCC.2018.8538687.

[11] Broadband Internet Technical Advisory Group (BITAG). *Port Blocking*. Tech. rep. Technical report documenting ISP port-blocking practices including TCP/139 and TCP/445. BITAG, Feb. 2015. URL: https://www.bitag.org/documents/Port-Blocking.pdf.

[12] Microsoft. *MsQuic*. URL: https://github.com/microsoft/msquic (visited on Aug. 26, 2025).

[13] Tuxera. *Tuxera Fusion SMB*. URL: https://www.tuxera.com/products/tuxera-fusion-smb/ (visited on Aug. 26, 2025).

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

[15] 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.

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

[17] Margaret Rouse. *Data Stream*. URL: https://www.techopedia.com/definition/6757/data-stream (visited on Aug. 28, 2025).

[18] B. Ford. "Structured streams: a new transport abstraction". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: Association for Computing Machinery, 2007, pp. 361–372. ISBN: 9781595937131. DOI: 10.1145/1282380.1282421. URL: https://doi.org/10.1145/1282380.1282421.

[19] R. R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Sept. 2007. DOI: 10.17487/RFC4960. URL: https://www.rfc-editor.org/info/rfc4960.

[20] R. R. Stewart, M. Tüxen, and karen Nielsen. *Stream Control Transmission Protocol*. RFC 9260. June 2022. DOI: 10.17487/RFC9260. URL: https://www.rfc-editor.org/info/rfc9260.

[21] H. Nielsen, R. T. Fielding, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945. May 1996. DOI: 10.17487/RFC1945. URL: https://www.rfc-editor.org/info/rfc1945.

[22] H. Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. June 1999. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616.

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

[24] Google. *SPDY: An experimental protocol for a faster web*. URL: https://www.chromium.org/spdy/spdy-whitepaper/ (visited on Aug. 29, 2025).

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

[26] M. Belshe, R. Peon, and M. Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. May 2015. DOI: 10.17487/RFC7540. URL: https://www.rfc-editor.org/info/rfc7540.

[27] R. Peon and H. Ruellan. *HPACK: Header Compression for HTTP/2*. RFC 7541. May 2015. DOI: 10.17487/RFC7541. URL: https://www.rfc-editor.org/info/rfc7541.

[28] C. '. Krasic, M. Bishop, and A. Frindell. *QPACK: Field Compression for HTTP/3*. RFC 9204. June 2022. DOI: 10.17487/RFC9204. URL: https://www.rfc-editor.org/info/rfc9204.

[29] A. O. Freier, P. Karlton, and P. C. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101. Aug. 2011. DOI: 10.17487/RFC6101. URL: https://www.rfc-editor.org/info/rfc6101.

[30] 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.

[31] C. Allen and T. Dierks. *The TLS Protocol Version 1.0*. RFC 2246. Jan. 1999. DOI: 10.17487/RFC2246. URL: https://www.rfc-editor.org/info/rfc2246.

[32] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346. Apr. 2006. DOI: 10.17487/RFC4346. URL: https://www.rfc-editor.org/info/rfc4346.

[33] K. Moriarty and S. Farrell. *Deprecating TLS 1.0 and TLS 1.1*. RFC 8996. Mar. 2021. DOI: 10.17487/RFC8996. URL: https://www.rfc-editor.org/info/rfc8996.

[34] E. Rescorla and T. Dierks. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. Aug. 2008. DOI: 10.17487/RFC5246. URL: https://www.rfc-editor.org/info/rfc5246.

[35] T. Polk and S. Turner. *Prohibiting Secure Sockets Layer (SSL) Version 2.0*. RFC 6176. Mar. 2011. DOI: 10.17487/RFC6176. URL: https://www.rfc-editor.org/info/rfc6176.

[36] E. Rescorla, H. Tschofenig, and N. Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. Apr. 2022. DOI: 10.17487/RFC9147. URL: https://www.rfc-editor.org/info/rfc9147.

[37] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. 10th ed. John Wiley & Sons, 2021, pp. 757–769.

[38] T. Haynes and D. Noveck. *Network File System (NFS) Version 4 Protocol*. RFC 7530. Mar. 2015. DOI: 10.17487/RFC7530. URL: https://www.rfc-editor.org/info/rfc7530.

[39] Andrew Tridgell. *Myths About Samba*. URL: https://www.samba.org/samba/docs/myths_about_samba.html?ref=cybersecuritycrumbs.cryptopathe.me (visited on Sept. 2, 2025).

[40] R. Sandberg et al. "Design and implementation of the Sun network filesystem". In: *Proceedings of the summer 1985 USENIX conference*. 1985, pp. 119–130.

[41] T. Haynes. *Network File System (NFS) Version 4 Minor Version 2 Protocol*. RFC 7862. Nov. 2016. DOI: 10.17487/RFC7862. URL: https://www.rfc-editor.org/info/rfc7862.

[42] D. Noveck and C. Lever. *Network File System (NFS) Version 4 Minor Version 1 Protocol*. RFC 8881. Aug. 2020. DOI: 10.17487/RFC8881. URL: https://www.rfc-editor.org/info/rfc8881.

[43] 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.

[44] 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.

[45] 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].

[46] 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.

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

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

[49] *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].

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

[51] 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.

[52] 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.

[53] *Server Message Block (SMB) Protocol Versions 2 and 3*. Tech. rep. Release: July 28, 2025. Version v83.0. Microsoft Corporation, July 2025. URL: https://winprotocoldoc.z19.web.core.windows.net/MS-SMB2/%5bMS-SMB2%5d.pdf.

[54] P. J. Leach et al. *The Simple and Protected Generic Security Service Application Program Interface (GSS-API) Negotiation Mechanism*. RFC 4178. Oct. 2005. DOI: 10.17487/RFC4178. URL: https://www.rfc-editor.org/info/rfc4178.

[55] D. C. Neuman et al. *The Kerberos Network Authentication Service (V5)*. RFC 4120. July 2005. DOI: 10.17487/RFC4120. URL: https://www.rfc-editor.org/info/rfc4120.

[56] *NT LAN Manager (NTLM) Authentication Protocol*. Tech. rep. Release: April 23, 2024. Version v36.0. Microsoft Corporation, Apr. 2024. URL: https://winprotocoldoc.z19.web.core.windows.net/MS-NLMP/%5bMS-NLMP%5d.pdf.

[57] R. J. Recio et al. *A Remote Direct Memory Access Protocol Specification*. RFC 5040. Oct. 2007. DOI: 10.17487/RFC5040. URL: https://www.rfc-editor.org/info/rfc5040.

[58] Microsoft. *SMB Direct*. URL: https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-direct (visited on Sept. 5, 2025).

[59] Microsoft. *SMB over QUIC*. URL: https://learn.microsoft.com/en-us/windows-server/storage/file-server/smb-over-quic (visited on Sept. 5, 2025).

[60] Microsoft. *SMB over QUIC*. URL: https://learn.microsoft.com/en-us/windows-server/storage/file-server/configure-smb-over-quic-client-access-control (visited on Sept. 5, 2025).

[61] QUIC Working Group. *Implementations and tools*. URL: https://github.com/quicwg/quicwg.github.io/blob/main/implementations.md (visited on Sept. 8, 2025).

[62] Xin Long. *QUIC in Linux Kernel*. URL: https://github.com/lxin/quic (visited on Sept. 8, 2025).

[63] M. Kempf et al. "QUIC on the Fast Lane: Extending Performance Evaluations on High-rate Links". In: *Computer Communications* 223 (2024), pp. 90–100. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2024.04.038. URL: https://www.sciencedirect.com/science/article/pii/S014036642400166X.

[64] Microsoft. *MsQuic*. URL: https://microsoft.github.io/msquic/msquicdocs/index.html (visited on Sept. 8, 2025).