Performance analysis of modern QUIC implementations

Calle Halme

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 26.4.2021

Supervisor

PhD Pasi Sarolahti

Advisor

PhD Pasi Sarolahti

Aalto University
School of Electrical Engineering

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Calle Halme	
Title Performance analysis of modern QUIC implementations	
Degree programme Computer, Communication and Information Sciences	
Major Communications Engineering	**Code of major** ELEC3029
Supervisor PhD Pasi Sarolahti	
Advisor PhD Pasi Sarolahti	
Date 26.4.2021 **Number of pages** 70	**Language** English

Abstract

Originally entering development in 2012 by a Google engineer, QUIC is a secure general-purpose transport layer protocol currently developed by the IETF, finalized and awaiting the release of its first non-draft RFC. While QUIC is deployed on top of UDP, it is connection-oriented, provides reliable packet delivery, and congestion control. QUIC also integrates TLS for additional security. QUIC data transfer is based on datastreams, transporting packets consisting of various types of predefined frames. This structure provides multiple advantages over TCP, such as stream multiplexing and no head-of-line blocking.

In addition to background information, and an overview of the QUIC protocol, this thesis includes a performance analysis on three QUIC implementations, compared to a TCP implementation. The testing was done with a custom Python-based test framework developed for this thesis, using Mininet as a network emulator. The test scenarios consist of client applications downloading files of varying sizes from a server in increasingly poor network conditions. The results show that even in scenarios that don't fully utilize QUIC's more advantageous features, QUIC can match TCP in connection duration in certain circumstances, and even surpass it.

Keywords AIOHTTP, aioquic, LSQUIC, ngtcp2, performance, QUIC, TCP, UDP

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Calle Halme	
Työn nimi Modernien QUIC toteutusten suorituskykyanalyysi	
Koulutusohjelma Computer, Communication and Information Sciences	
Pääaine Communications Engineering	**Pääaineen koodi** ELEC3029
Työn valvoja FT Pasi Sarolahti	
Työn ohjaaja FT Pasi Sarolahti	
Päivämäärä 26.4.2021	**Sivumäärä** 70 **Kieli** Englanti

Tiivistelmä

QUIC, jonka kehitystyö alkoi vuonna 2012 Googlen toimesta, on turvattu kuljetuskerroksen tiedonsiirtoprotokolla. QUIC:n kehitystyö on siirtynyt IETF:lle, ja protokolla odottaa tällä hetkellä ensimmäistä virallista RFC versiotaan. Vaikka QUIC on toteutettu UDP protokollan päälle, on se silti yhteysorientoitunut, luotettava, sekä tarjoaa ruuhkanhallintaa. QUIC integroi myös TLS protokollan saavuttaakseen korkeamman tietoturvatason. QUIC:n tiedonsiirto perustuu pakettivirtoihin, ja paketteihin jotka koostuvat erilaisista ennalta määritetyistä kehyksistä. Tämä rakenne tarjoaa useita etuja TCP:hen nähden; useita tietovirtoja voidaan multipleksata yhteen yhteyteen, eikä pakettien hukkuminen aiheuta QUIC:ssä kaikkien muiden pakettien käsittelyn pysähtymistä.

Taustatietojen, sekä QUIC protokollan yleiskatsauksen jälkeen tässä diplomityössä esitetään kolmen QUIC toteutuksen, sekä vastaavan TCP toteutuksen suorituskykyanalyysi. Analyysin vaatimat testit on toteutettu diplomityötä varten kehitetyssä Pythoniin pohjautuvassa testiympäristössä, hyödyntäen Mininet verkkoemulaattoria. Testiskenaarioissa asiakasohjelma lataa eri kokoisia tiedostoja palvelimelta muuttuvassa verkkoympäristössä. Tulosten perusteella voidaan todeta, että vaikka testiskenaariot eivät hyödynnä kaikkia QUIC:n hyödyllisimpiä ominaisuuksia, QUIC saavuttaa silti TCP:tä vastaavia, sekä jopa parempia tuloksia tietyissä tilanteissa.

Avainsanat AIOHTTP, aioquic, LSQUIC, ngtcp2, QUIC, suorituskyky, TCP, UDP

Contents

Abstract **3**

Abstract (in Finnish) **4**

Contents **5**

Abbreviations **7**

1 Introduction **8**
- 1.1 Research objectives and questions . . . 9
- 1.2 Structure of the thesis . . . 10

2 Background **11**
- 2.1 Internet transport protocols . . . 11
  - 2.1.1 User Datagram Protocol . . . 11
  - 2.1.2 Transmission Control Protocol . . . 12
  - 2.1.3 Congestion control . . . 13
- 2.2 Enhancements on streams . . . 15
  - 2.2.1 Structured Streams Transport . . . 15
  - 2.2.2 SCTP . . . 17
- 2.3 Security . . . 18
  - 2.3.1 IPsec . . . 18
  - 2.3.2 TLS . . . 19
  - 2.3.3 DTLS . . . 20
- 2.4 Web and HTTP . . . 21
  - 2.4.1 SPDY . . . 22
  - 2.4.2 HTTP/2 . . . 23
  - 2.4.3 HTTP/3 . . . 24

3 QUIC **25**
- 3.1 Origins . . . 25
- 3.2 Packets and frames . . . 26
  - 3.2.1 Formats . . . 28
  - 3.2.2 Packetization, reliability and datagram size . . . 28
- 3.3 Streams . . . 29
- 3.4 Control and recovery . . . 31
  - 3.4.1 Flow Control . . . 31
  - 3.4.2 Congestion Control . . . 32
  - 3.4.3 Recovery . . . 33
- 3.5 Connections . . . 34
  - 3.5.1 Identifiers and establishment . . . 35
  - 3.5.2 Migration . . . 38
  - 3.5.3 Management . . . 38
  - 3.5.4 Closure . . . 39

3.6 Security 40

4 QUIC implementations **43**
4.1 Overview 43
4.2 Implementations chosen for analysis 46
4.2.1 aioquic 46
4.2.2 LSQUIC 47
4.2.3 ngtcp2 47

5 Research methods **49**
5.1 Test environment 49
5.2 Test framework 50
5.3 Test scenarios 51
5.3.1 Scenario 1. Ideal network conditions 52
5.3.2 Scenario 2. Nonideal network conditions 52
5.3.3 Scenario 3. Multiple co-existing implementations 53
5.3.4 Scenario parameters 53

6 Results **54**
6.1 Scenario 1. Ideal network conditions 54
6.2 Scenario 2. Nonideal network conditions 57
6.2.1 Delay: 50ms, Packet loss: 2% 58
6.2.2 Delay: 200ms, Packet loss: 0% 60
6.2.3 Delay: 200ms, Packet loss: 5% 61
6.3 Scenario 3. Multiple co-existing implementations 63

7 Conclusions **65**
7.1 Research summary 65
7.2 Limitations of the study 66
7.3 Suggestions for further research 66

References **68**

Abbreviations

AEAD	Authenticated Encryption with Associated Data
AH	Authentication Header
ALPN	Application-Layer Protocol Negotiation
API	Application Programming Interface
BBR	Bottleneck Bandwidth and Round-trip
DTLS	Datagram Transport Layer Security
ECN	Explicit Congestion Notification
ESP	Encapsulating Security Payload
FEC	Forward Error Correction
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPsec	Internet Procotol Security
IoT	Internet of Things
MAC	Message Authentication Code
NAT	Network Address Translation
OSI	Open System Interconnection (model)
PKI	Public Key Infrastructure
RFC	Request For Comments
RTT	Round-Trip Time
SA	Security Association
SACK	Selective Acknowledgements
SCTP	Stream Control Transmission Protocol
SSL	Secure Sockets Layer
SST	Structured Stream Transport
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol
URI	Universal Resource Identifier
URL	Uniform Resource Locator

1 Introduction

The OSI-model (*Open System Interconnection model*) is a high level communication protocol characterization model, with the aim of providing proper interoperability and protocol standards irregardless of specific technical details and architecture relating to any one protocol. The model was first introduced in the late 1970s, and has long since been used in the design process and implementation of various communication protocols. Even the TCP/IP model, the Internet's core set of communication protocols, can be loosely mapped to parts of the OSI model despite the differing layering structures. [1]

An integral part of the OSI model, the transport layer is responsible for connecting any two network nodes wishing to communicate with each other over the Internet, providing them with a logical bidirectional connection. The nodes themselves see each other as directly connected, as the transport layer abstracts any network infrastructure between them to a simple connection, enabling the nodes to communicate indifferently to the number of middleboxes connecting them. The information that is to be transmitted is packaged into segments or datagrams, and sent further down the OSI-stack. Upon arrival to its peer, this process is reversed and the receiving node has its data. [1] There are various protocols to facilitate this process, however TCP (*Transmission Control Protocol*) [2] and UDP (*User Datagram Protocol*) [3] are historically the most commonly used ones [4]. A more in-depth overview of these two protocols is provided in Section 2.1.

While both of these protocols have been successfully used for decades now, they have their limitations due to being initially developed so long ago. However, as they are already well-established and ubiquitous protocols in modern network infrastructure, any additions or alterations are fundamentally limited in their scope due to backwards compatibility issues. This networking phenomenon is known as ossification, and it refers to the presence of a high amount of inflexible middleboxes within the network infrastructure [5]. Additionally, especially in the case of TCP as the more widely used protocol [4], any changes have to compete for the limited 40 bytes of space allotted in the options field of the header, meaning only the most critical additions can achieve more widespread use. And even then, there always exists the possibility of running into a middlebox that only supports the most basic functionalities.

Due to the inherent limitations of the TCP protocol, combined with the ever-increasing needs of modern networking hardware and software, new protocols have been proposed as alternatives over time - QUIC is one such protocol. Originally created by Jim Roskind at Google in 2012 [6], QUIC has evolved significantly in many ways throughout the last eight years. One major evolution happened a few years after the initial reveal, as a version of the protocol was adopted by the IETF (*Internet Engineering Task Force*) for future development as a potential replacement for TCP; an Internet Draft was submitted in 2015, and a working group established in 2016.[1,2] While the design considerations of the protocol are in the hands of the

[1]https://mailarchive.ietf.org/arch/msg/i-d-announce/zSk53ClZRO6eSH4s5a7bQ5Jiuns/
[2]https://datatracker.ietf.org/wg/quic/about/

IETF now, Google does still maintain its own implementation of the protocol, which is a part of their Chromium Projects.[3] While QUIC initially stood for Quick UDP Internet Connections, the evolved form being worked on by the IETF is not actually an acronym of anything. Thus, any further mention of QUIC will refer to the IETF implementation, as opposed to the original one.

The main intention of QUIC is to be a faster, more robust version of TCP. This is achieved by multiple design decisions, including the use of UDP as the base for the protocol. This is done because it allows QUIC to implement its own packet loss recovery system independently for each stream, alleviating blocking issues caused by error-heavy connections. The protocol latency is also improved by heavy focus on handshake optimization, as well as TLS integration into the protocol itself - this also provides the benefit of encryption by default. QUIC also provides connections with their own unique identifiers, which facilitates faster connection re-establishment. While the protocol is still under development, it would seem that the core functionality has been mostly finalized. The current QUIC RFC (*Request For Comments*) draft as of writing this thesis is 34. [6] The QUIC protocol is further discussed in Section 3.

As with any protocol with potential to become the future TCP, many implementations have been developed by parties most interested and involved in QUIC development and testing. These implementations[4] vary in both design philosophy and programming language, but do their best to abide by the protocol standards set by the IETF. All major QUIC implementations will be briefly covered in Section 4, followed by further emphasis on the implementations chosen for further study. Having multiple implementations with very different styles is very useful for testing, as the network infrastructure that QUIC is to be integrated with is also incredibly diverse and heterogeneous. It also enables more robust interoperability testing to determine whether the protocol is defined well enough that room for error based on interpretation is minimal.

1.1 Research objectives and questions

Since some of the abovementioned implementations may eventually reach very widespread use, it is prudent to do performance testing to assure they overperform the protocol they are to replace - TCP. It is also interesting to see how these protocols perform in regards to each other: how much difference does the programming language of choice, for example, actually make. Therefore, the main questions this thesis aims to answer are as follows:

1. Does QUIC provide a noticeable performance benefit over TCP? If so, how much does this vary in different network conditions, if at all?
2. Does the protocol performance vary between different QUIC implementations?
3. If so, how can these differences be explained? Additionally, are there any patterns present in the data that can be used to further analyze said differences

[3]https://www.chromium.org/

[4]https://github.com/quicwg/base-drafts/wiki/Implementations

between implementations?

To this end, three different QUIC implementations were chosen for further testing for this thesis, alongside a corresponding TCP implementation as a control setup. The implementations were chosen based on criteria detailed in Section 4.2, with the critical requirements including adherence to the latest QUIC draft(s) at the time of testing, as well as having comparable testing environments in the form of client/server implementations that support HTTP (*Hypertext Transfer Protocol*). The chosen implementations feature both the Python programming language, as well as C, which have traditionally been noted as having major differences in regards to performance, especially as the program sizes increase.[5]

The implementations were tested in an emulated Mininet[6] environment, consisting of at least one client and one server, connected via a traffic controlled link with varying properties. The resulting traffic was captured from the server's side of the network using tshark, alongside potential logging provided by the implementations. The testing environment, framework, and scenarios are further explained in Section 5.

The results show that the QUIC implementations can match, and even outperform TCP in the various scenarios. The QUIC implementations performed more evenly with TCP in scenarios with better network conditions, and increasingly better as the conditions degraded, though TCP was not always the slowest performer. The QUIC implementations manage faster handshake times than TCP with a few exceptions, which is more relevant performance-wise in the shorter connections. The QUIC implementations also send more packets and more server overhead than TCP, and none of them seem to be able to perform fairly in parallel with TCP.

1.2 Structure of the thesis

The structure of this thesis will proceed in a logical order, starting from an overview of background information in Section 2, relating to the underlying technology relevant to QUIC. Section 3 covers the QUIC protocol itself in detail, followed by Section 4 on currently known implementations, as well as a deeper look on the implementations chosen for further analysis and the selection criteria. Following this, the research methods and the test environment developed for the thesis is discussed in Section 5, alongside the test scenarios that were used for producing the results. An overview of the most relevant results can be found in Section 6. Finally, Section 7 will focus on conclusions and some meta-discussion on the thesis itself.

[5]https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gcc.html
[6]http://mininet.org/

2 Background

This section of the thesis will cover some key background information regarding the transport layer and its protocols, along a slightly deeper look into congesting control. The concept of streams is also covered, including their use in improving data transfer in the transport layer. Finally, the section contains various security protocols, and an overview of HTTP.

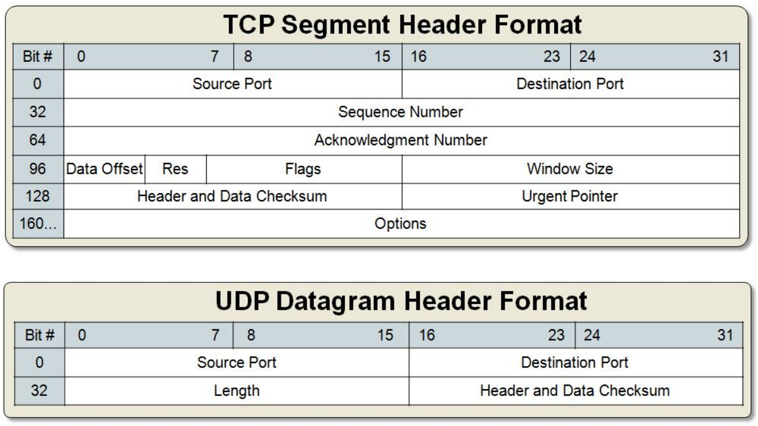
2.1 Internet transport protocols

The transport layer resides in the OSI model as an abstraction for the communication that occurs between different application processes. For the sake of comparison, the link layer, for example, handles data transfer between actual network links like routers, and the network layer handles data transfer between devices connected to the network, such as PCs. The transport layer abstraction is useful, because it allows the processes to ignore many inconvenient metrics of the actual connection, like the distance between the processes or the network infrastructure that connects them, as these are handled by other protocols. This is achieved by the transport layer building its packets, called segments or datagrams, based on its own rules and then encapsulating them in IP (*Internet Protocol*) packets for further transport. The transport layer contains a multitude of different protocols that excel in different situations, with TCP and UDP being the most prevalent. The main purpose of this layer is to createa logical end-to-end connection between two network endpoints. It is noteworthy, however, that the network layer is responsible for routing the packets within the actual network infrastructure itself. The transport layer expands the functionality of IP, as well as potentially fixes some of the issues it experiences. To that end, the transport layer can provide such services as: flow and congestion control, error correction, connection-oriented transport, reliable transport, packet ordering and multiplexing. [1]

2.1.1 User Datagram Protocol

UDP is a communications protocol providing connectionless and unreliable data transfer between any number of endpoints, as opposed to something like TCP. While this may immediately seem inferior, it does have its benefits. Instead of establishing static connections between endpoints, UDP functions solely on IP addresses and port numbers for data delivery. The lack of connection establishment means less connection overhead, and lower latency. UDP connections can also be either uni- or bidirectional, and support multicast. As can be seen in Figure 1, UDP headers are also significantly less complex than their TCP counterparts, containing only port numbers, data length and a checksum. This helps to further reduce network overhead. [3]

The simplicity of the protocol does have its drawbacks, as the unreliability in this case means that UDP does not give any guarantees of packet delivery, be it the order or even the fact that any given packet is delivered successfully. It also does not implement any congestion control methods by default. This, combined with the smaller overhead and latency, results in UDP being used in implementations that are speed-reliant and error-resistant, such as real-time voice communications. [1]



TCP Segment Header Format

Bit #	0	7	8	15	16	23	24	31
0	Source Port				Destination Port			
32	Sequence Number							
64	Acknowledgment Number							
96	Data Offset	Res	Flags		Window Size			
128	Header and Data Checksum				Urgent Pointer			
160...	Options							

UDP Datagram Header Format

Bit #	0	7	8	15	16	23	24	31
0	Source Port				Destination Port			
32	Length				Header and Data Checksum			

Figure 1: TCP and UDP headers.[7]

2.1.2 Transmission Control Protocol

TCP is a communication protocol that provides the endpoints connection-oriented, reliable, and bidirectional data transfer. When a TCP connection is established, it is done via a three-way-handshake, initiated by the client. This handshake consists of the client and server exchanging segments containing SYN (*Synchronize*) flags, as well as their respective sequence numbers used throughout the ensuing connection. After the handshake is completed, data can be exchanged between the endpoints assuming no other preamble is needed due to other protocols, like TLS (*Transport Layer Security*) for example. [2]

Being connection-oriented means that there is a single data "pipe" per any one TCP connection, whereas reliability means that the protocol guarantees the delivery of each packet, as well as their order. This is obviously beneficial for processes that operate under strict accuracy constraints in regards to data, however ensuring the delivery of ordered packets can cause large delays when packets go missing on the way. TCP achieves reliability with the usage of cumulative packet acknowledgements. This means that each received packet has to be acknowledged by the receiver. If packets are received out of order, the receiving node will have to buffer or drop the packets, and retransmit the acknowledgement of the latest correct packet. When a sender receives three sequential acknowledgements for the same packet, it interprets that a packet after this has been lost, and sends said packet again. The problem with this method is the fact that packets in the pipe after the lost packet can't be properly processed until the lost packet has been succesfully delivered. This results in a number of packets being stuck in receiver buffers, or even being dropped in

[7]https://skminhaj.files.wordpress.com/2016/02/92926-tcp_udp_headers.jpg

some cases, which can introduce long delays when sending large amounts of data - this problem is also known as head-of-line blocking. TCP does have a timer that can trigger retransmission of packets, but the duplicate acknowledgement method is usually faster. Data reliability can additionally be verified with a checksum field in the TCP packet header, which can be used to measure data integrity, although it is not infallible as it is only 32 bits. [1, 2]

In addition to reliable data transfer, TCP also includes congestion control algorithms and methods. These are not necessarily exclusive to TCP, and as such are futher discussed in the next section. In case of TCP specifically, the packet headers include a window size parameter, which represents available buffer size of the endpoint. The protocol also has a congestion window attribute, which is updated based on packet loss, and used to control the transfer speed of data so as to accomodate more congested network conditions. The congestion window starts out small, but is rapidly incremented until a certain threshold is reached or packet loss occurs. If the threshold is reached, the protocol enters congestion avoidance, meaning the window is incremented slowly until packet loss is detected. At this point, the window can either be halved or reset. Additionally, TCP headers contain a flag for detecting congestion, the ECN (*Explicit Congestion Notification*) flag, but this isn't yet widely used. [1, 2]

It is also noteworthy that TCP has a sizable header compared to its common counterpart, UDP, however it is not unusual for protocol headers to be even larger - QUIC is one such example. As shown in Figure 1, the TCP header contains source and destination port information, sequence and acknowledgement numbers, as well as multiple other fields related to managing the connection and data. One of the more important of these additional fields is the options field, as it can be used to add extra functionality to the protocol. This doesn't come without its limitations however, as the options field is limited in size to 40 bytes, and is highly contested between multiple protocols. Solutions of varying quality have been proposed for this problem, but none have been widely adopted so far. Another problem caused by the options field is ossification, as certain middleboxes running old or simplistic software may elect to simply drop TCP packets that include unknown parameters in the options field.

2.1.3 Congestion control

The concept of congestion control refers to actions taken in order to regulate network traffic in order to avoid high levels of congestion, or too much data in flight within a network at any given time. While modern network infrastructure has gotten quite fast and reliable, every network has its limits, which can be measured with metrics such as bandwidth and buffer sizes. When too many nodes in a given network send data at rates the network cannot support, the network will become increasingly congested, leading to a variety of problems. These problems include, but are not limited to, the following:

1. Increased queuing delays, as data rates exceed the rate at which routers or middleboxes can process data, or the rate at which links can transfer data.

2. Packet loss, as the buffers of various network nodes fill up and they are forced to drop any new packets.

3. The need for retransmissions, as packets are either lost or extremely delayed. This will also further increase congestion and exacerbate the other problems.

4. Wasted processing effort by routers in the case that packets are sent multiple times even if they are not lost, just delayed enough to trigger retransmission timers or other such mechanisms.

5. Wasted effort by all nodes along a multihop path in forwarding a packet, if said packet is dropped before it reaches its destination. [1]

To avoid these problems and ensure timely and fair data transfer, the network needs to be monitored and appropiately throttled when necessary. To this end, two different high-level types of congestion control can be identified: end-to-end and network-assisted. In the first case, the network layers do not assist the transport layer in detection, nor handling of congestion - the transport layer must instead infer congestion based on metrics available to it, such as delay and lost packets. In network-assisted congestion control, however, the routers will provide congestion information to the transport layer, which can occur in the form of simple flags that indicate congestion, or more in-depth metrics that can be used to measure the level of congestion as well. This feedback is communicated either via dedicated packets, or edited flags on the data being transmitted. [1]

While these problems and their solutions are more clear cut, network congestion can manifest more abstract issues as well, such as fairness. In the context of networks, fairness refers to the equal distribution of network resources across applications and users. A congestion scenario including different protocols and algorithms with irregular rules regarding fairness may lead to unfair utilization of network resources. While certain fairness measures exist in current congestion control schemes, they generally only work in cooperative, TCP-friendly environments. Certain protocols, whether with malicious intent or not, may use methods such as opening multiple separate connections to transmit data, not throttling their send rates sufficiently in case of congestion, or using outdated methods for detecting congestion altogether, which may lead into them monopolizing the bandwidth at the cost of other network users. One way of achieving fairness in networks is to manage the queues of routers and switches, equalizing flows by keeping the more aggressive ones in check. While these types of methods have their benefits, they also come with downsides: they can have a high impact on the network architecture, and may not treat all flows equally if they use unsupported metrics for detecting congestion, like delay or increases in RTT (*Round-Trip Time*). They can also be too slow to react, and are not robust and scalable enough for high-speed networks with thousands of flows. [7, 8]

As congestion control is more of a general concept than a specific feature, common protocols that provide congestions control, such as TCP, have a wide variety of different algorithms with different sets of features. Some of the more popular ones over the years include: New Reno, SACK (*Selective Acknowledgements*), CUBIC and BBR

(*Bottleneck Bandwidth and Round-trip*). New Reno is a successor for Reno, providing some functionality extensions to the fast retransmit and fast recovery algorithms, the intention of which is to avoid unnecessary timeouts and congestion window reductions [9]. SACK is a protocol intended to help TCP avoid unnecessary retransmissions by extending TCP's ability to keep track of lost packets by introducing SACK packets, which are sent by the data receiver and include received packet information [10]. CUBIC is a TCP standard extension, altering the window increase function of congestion control to a cubic one instead of linear function. This change provides enhanced scalability and stability in faster networks with longer links. CUBIC is the default congestion control algorithm for the Linux operating system [11]. BBR is a more recent algorithm developed by Google, with the intention of detecting congestion with metrics other than simply packet loss, and improved performance in poor networks. It also does not need to be implemented client-side to function. [12]

2.2 Enhancements on streams

In its most general definition, a data stream is a sequence of elements that are made available over time [13]. For message-oriented protocols like UDP, a sequence of messages can be seen as a data stream. In contrast, for a protocol like TCP which transports data in an ordered stream of bytes, the end-to-end connection itself between two network nodes can be seen as a data stream. Therefore, a TCP connection can only ever contain a single stream. Unfortunately, due to the need for reliability, opening and closing such connections requires the execution of pre-determined procedures, that will cause a set amount of delay. Additional delays are also present in the connection itself, as the rigid packet ordering and reliable delivery of TCP can cause problems during situations where data is lost. Since TCP is largely the protocol of choice for modern data transfer, certain protocols have attemped to enhance the functionality of data streams.

2.2.1 Structured Streams Transport

In his 2007 paper [14], Ford introduces a new kind of transport abstraction with the goal of providing the best features of both stream- and datagram-based data transfer as necessary. This abstraction is known as SST (*Structured Stream Transport*), and contains key features such as the ability to create child streams, support for both reliable and best-effort data delivery, and dynamic stream prioritization. SST consists of three related subprotocols, as shown in Figure 2. The channel protocol provides connection-oriented data delivery and related functionality, while the negotiation protocol handles its state and optional features. These protocols are built upon by the stream protocol, which implementes the actual transport abstraction that applications see. SST can be deployed at either the transport layer, or the application layer; implementing SST as a library on top of UDP as opposed to a native transport protocol makes its usage more straightforward, and enables native NAT (*Network Address Translation*) traversal.

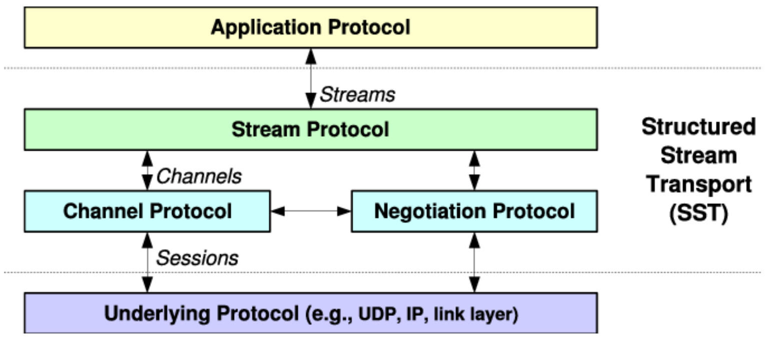


Figure 2: SST architecture.[8]

While TCP and UDP both have their optimal use cases, as Ford mentions in his paper [14], there are situations where the use of either on its own is hardly optimal. UDP is traditionally the protocol of choice whenever data sizes are small, and when minimizing overhead is of higher importance than data integrity. In contrast, TCP is more suited for situations with large data, longer connections, and emphasis on data delivery happening in order and the data arriving intact. [4] A situation that includes high variance in terms of data segment sizes and transfer times is, therefore, fundamentally incompatible with either protocol, especially if data integrity and proper congestion control is also desired. HTTP [15] is a protocol with such characteristics and requirements, and with its current prevalence in the functionality of the Internet, the problems it faces cannot be ignored.

While HTTP can, and is usually, run solely over TCP, this presents a number of problems due the variable nature of HTTP data transfer. One such problem is the need to open multiple individual TCP streams for loading elements from a single logical entity, like a web site for example, to decrease load times. This increases overhead, and brings with it extra delay due to operations required to properly open and close connections. This can also create unfair situations between applications in regards to the usage of available network resources. While certain applications and subsequent HTTP versions have implemented some levels of multiplexing, they do still run on top of TCP and thus exist within a single TCP stream, meaning they cannot completely escape issues like head-of-line blocking. Finally, neither TCP or UDP is properly equipped to deal with protocols or applications that take advantage of multiple related transport instances. This results in such protocols needing to pass additional information in messages, which can cause issues if this information is not handled properly by a node the data passes through, such as when traversing NATs. [14]

SST solves all these issues with the implementation of a hierarchical hereditary structure, which enables existing streams to spawn child streams without additional handshake procedures or closing latency. These substreams are only accessible by the intended recipient, and keep the original structural context. Unlike HTTP streams

[8]https://www.researchgate.net/profile/Bryan_Ford/publication/221164360/figure/fig1/AS:669023108857863@1536518947280/SST-Protocol-Architecture.png

within TCP, these streams are also functionally independent, with separate data transfer and flow control capabilities. This means that data can be transferred in variable order and rate between substreams. Some connection features are implemented across all substreams within a single context, however, like congestion control, sequencing, and the security state. This is done to minimize overhead and enable the application as many substreams as it needs, though there are methods in place to prevent overload due to too many substream requests. Since all substreams between two hosts share a security state, they can also be opened and closed with minimal latency and overhead, which means they can be dynamically managed based on data transfer needs. The substreams can alsoy be prioritized if necessary, and additional substreams can be used to do so mid-stream. Excluding the substream functionality, SST streams are nearly identical to TCP streams in terms of semantics to make porting applications easier. [14]

Another common problem in data transfer is ensuring data integrity. In this regard, UDP operates on a best-effort system, while TCP guarantees data is delivered in order and exactly as it was on the other end of the pipe. [4] The problem is that when the data flow varies massively in size and purpose, both of these models can be useful, or detrimental, depending on the current circumstance. SST recognizes this, and thus supports both methods. In practice this means that while data can be transmitted using best-effort delivery within a stream, these types of streams are indistinguishable to the receiver from other, more reliable streams. This enables an ephemeral fallback to guaranteed delivery if the stream encounters a datagram large enough where losing a part of it is practically guaranteed. [14]

2.2.2 SCTP

As defined in RFC 4960 [16], SCTP (*Stream Control Transport Protocol*) is a transport layer communication protocol that combines functionality of both UDP and TCP, while also providing additional functionality absent from either of these more well-known protocols. Just like the Structured Streams Transport protocol covered in Section 2.2.1, the creation of SCTP was inspired by certain limitations of TCP. These limitations include, but are not limited to, the inflexibility of TCP's data sequencing, the limited scope of the socket implementation, and vulnerability to denial-of-service attacks.

Like TCP, SCTP is a connection-oriented transport protocol that provides reliable data transport with congestion control functionality. Unlike TCP, however, it is message-oriented as opposed to stream-oriented. This means that data transferred via SCTP moves in independent groups of bytes instead of a single stream, enabling support for both ordered and unordered data. These byte groups, or messages, are organized into chunks for transport, and may be fragmented into multiple separate chunks depending on their size. The messages and related control data have separate chunk types designated by their header. These chunks are then combined into a SCTP packet following a general packet header. While these packets can contain multiple chunks, each chunk only contains data related to a single message. Multiple sequences of these chunks can be sent in parallel under a single SCTP association between two endpoints, where a block on one sequence does not affect the others, therefore effectively supporting multiple datastreams simultaneosly. Each message can also be assigned a sequence number for ordered delivery, although this is optional. [16]

Another key feature of SCTP is multihoming: if an endpoint supports multihoming, it will broadcast multiple IP addresses linked to the same SCTP port upon association startup, all of which can be used to send data to, and receive data from, said endpoint. This system provides the protocol with additional redundancy, which makes it more resilient to failures in the network. To counter flooding attacks, SCTP also features a 4-way handshake upon startup, bolstered by the use of cookies to disencourage resource commitment until handshake completion. The packet headers do also feature verification tags and checksums for added security. [16]

2.3 Security

In a world of ever increasing digitalization of information and services, as well as the growing popularity of IoT (*Internet of Things*) devices, proper implementations of all aspects of data security are increasingly important. This is especially true for all the protocols that facilitate the data transfer between the estimated 50 billion devices connected to the Internet [17]. In essence, to secure a network resource is to prevent any unauthorized access to it, whether from within the resource itself, or externally. In terms of the transport layer, this generally means protecting data while it travels between the communicating endpoints.

The main services used to provide this protection are: access control, authentication, confidentiality, integrity and nonrepudiation. Access control is a service used to determine whether an entity using a service is allowed to do so, whereas authentication is used to ascertain the identity of said entity. Confidentiality guarantees that data traveling between nodes in the network cannot be viewed by unauthorized parties, while integrity ensures that those parties do not alter this data in any way. Finally, nonrepudiation provides proof of where the data originated from. [18]

2.3.1 IPsec

IPsec (*Internet Protocol Security*), as the name would suggest, is a collection of security protocols for networks based on IP. IPsec is a set of authentication and encryption protocols, spanning multiple RFC's, and developed by the IETF. It functions invisibly to both transport and application layers, and supports both IP versions, with a goal of providing the network access control, integrity, confidentiality, authentication, as well as replay protection. The two main parts of IPsec are the AH (*Authentication Header*) and ESP (*Encapsulation Security Payload*) protocols, where both provide authentication and integrity, but only the latter provides confidentiality. The AH protocol functions by inserting a special header into an IP datagram, separating the IP header from the data, shielding and encapsulating the payload into a new IP datagram. ESP instead encapsulates the entire IP datagram into a new datagram with new headers and added encryption for confidentiality. As ESP also provides confidentiality, it has ended up as the more popular choice when it comes to these protocols. [1]

To provide this level of security, IPsec needs information about the nodes in the form of SAs (*Security Association*), or parameters and procedures the nodes must exchange and commit to in order to ensure secure data transmission - this is done via logical connection-oriented network layer communication channels. While the network layer is connectionless, these connections are maintained by state information each endpoint keeps of their peer, which consists of an identifier, the source and destination interfaces, and encryption information, such as negotiated keys and algorithms. The channels are not bidirectional, however, so one must be established for each direction of data transfer. For implementing SAs, two different modes are available: transport and tunnel. The main difference between these two is, that transport mode leaves the original IP header unencrypted for routing, whereas tunnel mode encapsulates the original packet into a new one with a new header, and encrypts the entire original packet. [18]

2.3.2 TLS

TLS is a network security protocol, aiming to provide a secure and encrypted end-to-end connection between network nodes, independent of application platform. To this end, TLS provides the connection with data confidentiality, integrity, as well as both client and server authentication. If properly configured, TLS can also provide forward secrecy. As TLS is the security protocol of choice for HTTPS (*Hypertext Transfer Protocol Secure*), its presence is ubiquitous in modern networks. While TLS can be layered upon by higher-level protocols, it does not specify any rules on how said protocols should add security. In regards to the OSI-stack, TLS does not fit into any single layer, and instead resides between the network and application layers. TLS is a successor to a protocol known as SSL (*Secure Sockets Layer*) - since TLS originated as a version of SSL, these terms are sometimes still used interchangeably. [18]

As illustrated in Figure 3, TLS contains two layers of protocols: the handshake layer on top, and the record layer on the bottom. The record layer resides on top of some type of reliable transport protocol, and provides the connection with privacy via symmetric cryptography and reliability via MACs (*Message Authentication Code*). Using MACs is technically optional, however this is generally only the case during parameter negotiation. Higher-level protocols are encapsulated by the record layer, such as the other part of TLS - the handshake protocol. The handshake provides the connection endpoint authentication via asymmetric cryptography and a way to negotiatie connection parameters. During a TLS handshake, the connection endpoints will generally negotiate the highest TLS version they can both support, agree on a cipher spec, authenticate one or both endpoints, and generate session keys for encryption. This process can be conducted safely even in unsafe networks due to the use of verified certificates and PKI (*Public Key Infrastructure*). After both endpoints have the necessary keys and parameters, the encrypted data transfer can begin. [19]

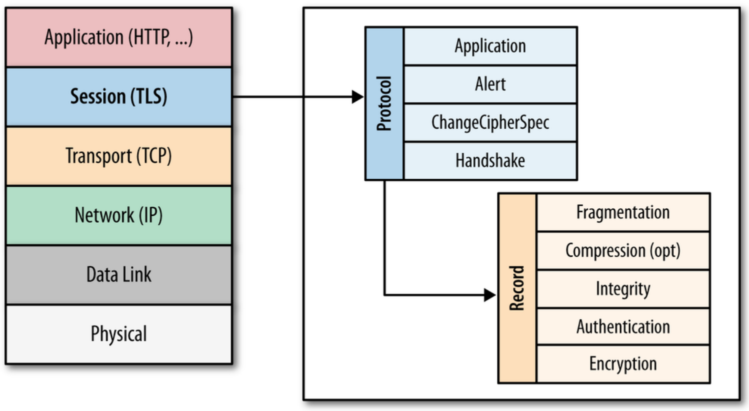


Figure 3: TLS architecture.[9]

Since adding extra steps to a connection is never free, TLS does introduce some performance impacts to systems it is implemented in. Data decryption especially can be quite the costly operation. However, to minimize these impacts, TLS as a protocol has already gone through a few iterations, the latest of which is TLS 1.3. Among other changes, improvements were made to the handshake protocol, and some older algorithms were deprecated. There are also additional methods of reducing latency even more, such as session resumption for pre-established connections and a TLS False Start protocol. While TLS is generally applied to securing HTTP data, denoted by the URI (*Universal Resource Identifier*) schema https, it can be taken advantage of by any application running over TCP. TLS is also a built-in part of QUIC. [19]

2.3.3 DTLS

As mentioned in the previous section, TLS requires a reliable transport channel in order to function properly, meaning that it cannot be used to provide security for unreliable transport protocols such as UDP. However, protocols that rely on UDP for data transfer are increasing in popularity, and the security options available to them without a TLS equivalent are not applicable to all situations. Out of the two main alternatives, IPsec has certain limitations application-wise, further detailed in RFC 5406 [20]. The other option would be a custom security protocol, which can prove to be too expensive or too much effort to design specifically for a project. DTLS (*Datagram Transport Layer Security*) was designed as a solution to these problems. It is purposefully as similar to the original TLS as possible, with only minimal changes to maintain functionality in cases of lost or re-ordered packets. The goal is to simply extend a TLS equivalent service to unreliable transport protocols without reinventing the wheel. [21]

DTLS runs in application space, and does not require any kernel modication. It also does not alter the reliability of data transport, nor compensate for lost or re-

[9]https://hpbn.co/assets/diagrams/9873c7441be06e0b53a006aac442696c.svg

ordered data. The goal is to preserve application behaviour running on top of DTLS, and thus UDP, while providing security. The key changes to DTLS in contrast to TLS are to do with problems created by data unreliability; the core problems to solve are the inability to decrypt individual records due to reliance on ordered sequence numbers, and critical errors upon failures to reliably deliver handshake packets. In order to solve the first problem, the inter-record dependacies in TLS's record layer are removed. This is done by banning stream ciphers, as they are retained between records in TLS, and changing implicit sequence numbers to explicit. To solve the handshake-related issue, DTLS uses a simple retransmission timer on both ends of the connection. Handshake messages are also given specific sequence numbers in the context of said handshake, and can thus be processed or queued depending on if they were received in the correct order. Since handshake messages are generally quite large compared to average datagrams, and thus fragmented over several DTLS records, a fragment offset and length is provided for each for reconstruction. An optional replay detection functionality is also available, using AH or ESP from IPsec. [21]

2.4 Web and HTTP

HTTP is a request-response protocol in the application layer, and is the base for a large part of Internet data communication. The protocol defines how web clients and servers interact with each other and how data is transmitted - both the message structures and their exchange procedures are specified by the protocol. While there is a great deal of nuance in HTTP, at a high level it functions by a client node submitting a request to a server node, and waiting until it responds appropriately with the requested resource if possible, alongside a relevant status code in a response message. There are several possible request methods, such as GET for requesting resources and POST for submitting an entity to a resource. The methods all operate via request messages, which are written in plain ASCII and consist of one request line and an indefinite amount of header lines. The request line includes the request type, resource URL (*Uniform Resource Locator*), and the protocol version, while the header fields consist of various optional parameters used to specify certain aspects of the connection.[10] [1] The latest iteration of the *original* HTTP protocol is known as HTTP/1.1 [22].

The basic operations of HTTP revolve around hypermedia documents, which are linked to each other via hyperlinks. URLs serve as a way to access HTTP resources, typically consisting of up to five sections. The sections, also illustrated in Figure 4, are as follows:

1. Scheme, referencing a protocol being used, such as HTTP. This cannot be empty.

[10]https://developer.mozilla.org/en-US/docs/Web/HTTP

[11]https://upload.wikimedia.org/wikipedia/commons/thumb/d/d6/URI_syntax_diagram.svg/2136px-URI_syntax_diagram.svg.png

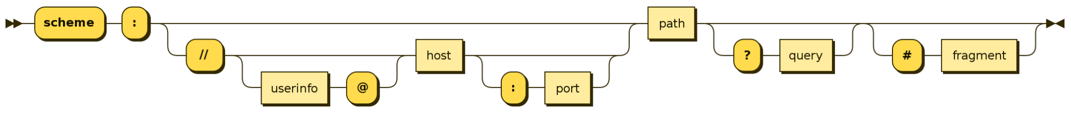


Figure 4: URL structure.[11]

2. Authority, further divided into the following subcomponents: login credentials, some type of address to connect to, and a port number.
3. Path, which is a filepath sequence of slash-separated segments used to access resources. While this is always defined, it can be of zero length.
4. Query section, containing some type of query string.
5. Fragment section, containing a fragment identifier. [23]

Any type of web browser is a typical HTTP client, whereas an application hosting some type of content online would be a typical HTTP server. HTTP is a stateless protocol, meaning that theoretically no data between two requests should be kept by the server. In reality this may not always be the case, as HTTP does allow for a variety of intermediate network elements, such as cache and proxy services for improving connections in different types of situations, such as when duplicate requests occur frequently. While HTTP operates on persistent connections by default, it can also be configured to use non-persistent connections if so desired. As an application layer protocol, HTTP needs a functional transport layer protocol in order to function, such as TCP, UDP or QUIC. Two major HTTP versions exist so far, both of which rely on TCP for data transport. The next version of the protocol, HTTP/3, is designed to use QUIC as its transport protocol, however. The reasons for this change are the potential performance improvements it can offer due to native support for multiplexing, with one key feature being the elimination of head-of-line blocking. While HTTP/2 already attempted to alleviate this issue by implementing connection multiplexing, it is still restricted by TCP as the underlying transport protocol. [1]

2.4.1 SPDY

SPDY is a protocol developed as a potential improvement for the original HTTP protocol, dedicated to reducing latency and moving away from using multiple TCP connections to process multiple resources concurrently. This type of behaviour caused longer connection setup times due to extra round trips, alongside other delays and problems related to rationing connections to a given server. Opening multiple connections to achieve concurrency can also cause unfairness in regards to resource sharing between endpoints, depending on how many connections are used and how stable said connections end up being. [24]

The improvements SPDY provides over regular HTTP are essentially the same improvements HTTP/2 provide as well: multiplexing, resource prioritization, header compression, and server push. Other similarities to HTTP/2 include the lack of changes to original semantics, as well as the usage of streams and frames in data transfer. SPDY is split into a framing layer and a HTTP layer, however these are interwoven and not designed to be used individually. For compatibility reasons the HTTP layer abides mostly by HTTP/1.1 rules, however certain aspects related to data processing have been changed or improved, like the implementation of server push features. While a SPDY implementation was succesfully developed and tested, with results showing improvements over HTTP, SPDY was eventually deprecated and replaced by HTTP/2. Substantial contributions from developers involved with SPDY are acknowledged in the HTTP/2 specification, however, which explains certain similarities between the two protocols.[12] [24]

2.4.2 HTTP/2

The second HTTP version is not as much an entirely new entity, but rather a more optimized version of the original. It does not change existing semantics, nor does it obsolete old message syntax - all core features of the older version are also fully supported. What it does offer, however, is content prioritization, connection multiplexing to a degree, unsolicited server push abilities, and header compression. Content priorization allows developers to give specific content static priorities, ensuring consistent load ordering of resources. This can have a large impact on both the actual, as well as perceived load times for web content. This is further combined with a stream-based multiplexing system, where streams are independent bidirectional frame sequences, prioritized to match high priority content accordingly. In addition, the streams are flow controlled both individually, and as a whole, to avoid overloading the receivers and preventing streams within a connection from consuming too much bandwidth. The flow control system is credit-based, where a receiver advertises its limits using WINDOW_UPDATE frames. Flow control is a mandatory part of the protocol, however only DATA frames are flow controlled. Unfortunately the multiplexing system does still face some TCP-related issues, as TCP's loss recovery mechanisms are not properly aware of it, keeping head-of-line blocking an issue.[13] [15]

Other HTTP/2 performance-related features include the ability for servers to push content to clients in an unsolicited manner, which can be more efficient than waiting for the client to request specific content in certain scenarios where content is especially partitioned. The servers can create synthetic requests for content they anticipate a client may need in advance, and use PUSH_PROMISE frames to communicate said data. HTTP/2 also provides header compression to reduce redundant data and fit more requests into a single packet. A compression method called HPACK [25] is used to better represent the header fields, which are then divided into fragments and transmitted via HEADERS-, PUSH_PROMISE-, or CONTINUATION frames. These frames containing the headers have to be transmitted in sequence with nothing else in between, until the flag indicating the end of the header block is received. [15]

[12]https://www.chromium.org/spdy/spdy-whitepaper
[13]https://www.cloudflare.com/en-gb/learning/performance/http2-vs-http1.1/

2.4.3 HTTP/3

Similar to its predecessor HTTP/2, and largely based on it, HTTP/3 is an improved way of handling the same syntax and semantics, however the major change in this version is the change in base transport protocol. Due to multiple favorable features, such as true multiplexing support, stream-based flow control, low latency and more, HTTP/3 has chosen QUIC over TCP as the transport protocol. Not only does this finally resolve the blocking issues of TCP, but it also allows HTTP/3 to delegate certain features to QUIC, as they already exist in the base protocol. This includes multiplexing and flow control, as well as TLS integration for security concerns. Similarly to HTTP/2 and QUIC, HTTP/3 uses frames as its base communication unit within streams. A separate control stream also exists for frames that affect the whole connection, as opposed to a single stream. The server push and header compression features of HTTP/2 are carried over as well, however HPACK is replaced with QPACK [26]. Some other key differences between HTTP/2 include support for a larger amount of streams, flow control for all frames, changes to frame types and removal of some due to QUIC features, and lack of priority signaling. A more detailed description of differences can be found in Appendix A of the HTTP/3 RFC Draft. [27]

3 QUIC

This section covers the QUIC protocol in detail, which is the main focus of study in this thesis. QUIC is a general-purpose stream-based transport protocol, with an emphasis on security and speed. Even though QUIC runs on top of UDP, it is a connection-oriented and stateful protocol, like TCP. It uses a handshake to initiate connections, where a TLS security handshake is integrated into the transport handshake to save time. Immediate data transfer is also supported for subsequent connections between the same endpoints with proper configuration. Data transfer in QUIC occurs via authenticated packets containing frames, which are encrypted by default to the maximum practical extent, while maintaining sufficient routing capabilities. Both bi- and unidirectional streams are supported, and a credit-based scheme is used to control them, where an endpoint will communicate its limits to its peers. Reliable delivery and congestion control algorithms are also supported, and further detailed in Section 3.4. Client connections can migrate to new network paths if necessary, and multiple options for connection termination exist. While the protocol is approaching finalization in terms of technical specifications, it is *technically* still in development as of writing, so the following information is subject to change. For the same reason, this entire chapter refers to the following RFC drafts unless otherwise specified: QUIC Transport [6], QUIC Recovery [28] and QUIC TLS [29].

3.1 Origins

As has been covered thus far, TCP and UDP are an ubiquitous part of data transfer infrastructure globally, with TCP being the de facto choice for reliable data transfer needs in most cases. As technology and the needs for data transfer have evolved, the limitations of a protocol originally designed in 1974 have become increasingly apparent. This has lead to multiple attempts at both improving TCP as far as it can be pushed, as well as the creation of completely new protocols to potentially replace it. As discussed in Section 2.2, some of these protocols focus on the efficient utilization of multiplexed data streams in order to provide scalable, independent connections without classic TCP problems like head-of-line blocking. Two of such protocols with connections to the development of QUIC, SST and SCTP, were detailed in Section 2.2.1 and Section 2.2.2.

QUIC development began in 2012 by a Google engineer Jim Roskind, with the main motivation being improved support for the now deprecated Google protocol SPDY, which has since been succeeded by HTTP/2. QUIC was to be deployed on top of UDP for the additional design freedom it provides. The design process presumably took some inspiration from SCTP, as the design document mentions that QUIC may end up resembling it in the end - which ended up being the case. The encryption system is also mentioned to be similar to DTLS. The reason these two protocols are being iterated upon in a brand new implementation instead of used as-is, is due to the need for the lowest possible latency, among other factors. The existence of SCTP and DTLS as separate layered protocols already introduces inherent latency, since a DTLS connection needs to be set up before an SCTP association [30], resulting in extra round trips. This latency is further elaborated on in the design document, with the concensus that a connection with these protocols might take up to four round trips to establish, whereas QUIC would ideally take one, or none in a case of connection re-establishment. Additionally, QUIC goals include an effective utilization of bandwidth, which is also difficult to achieve with separate protocols. Finally, neither protocol supported FEC (*Forward Error Correction*) at the time, which was another goal. [31]

The more general motivations for QUIC include reductions in latency, fewer retransmits, and a more responsive user experience. Additionally, it was supposed to reduce socket utilization via multiplexing connections into multiple streams, as well as consolidate network traffic and reduce the sending of redundant information. While SPDY is not utilized anymore, the motivations regarding it are still relevant: prevention of head-of-line blocking, TCP congestion control favoring sharded connections over multiplexed ones as it treats the multiplexed connections as one, and TLS-related delays in the form of session resumption and in-order decryption. The design document does also feature a list of twelve design goals, which generally follow the motivations already discussed. The goals that have not been mentioned yet, are: widespread deployability, better mobile support, privacy equivalent to TLS and efficient demux-mux for proxies. [31]

After the protocol was deployed and further experimented on, it was picked up by the IETF for additional development as a more general-purpose protocol. The protocol specifications have since been evolving towards various RFCs as draft versions, with each draft introducing new features or changes to existing ones - although the latest drafts have mainly focused on formatting. The main QUIC RFC [6], alongside the other core RFCs [28, 29, 27], are currently in last review, with the latest draft version being 34. Even in a draft state, QUIC is already supported by all the major browser vendors [32, 33], and is in widespread use by Google [34]. The upcoming HTTP/3 has also replaced TCP in favor of QUIC as the transport protocol of choice. [35]

3.2 Packets and frames

As with other transport protocols, QUIC works with packets encased in UDP datagrams. Certain packets associated with the handshake phase of the connection use a longer header, including the Initial, 0-RTT, Handshake, and Retry packets, whereas the rest use shorter headers for less overhead. Different types of packets also come with differing levels of encryption: Version Negotiation packets have no protection, whereas Retry and Inital packets use an AEAD (*Authenticated encryption with associated data*) [36]. The AEADs are used to safeguard against accidental packet modifications, as well as a soft limit for parties that can generate a valid response. The key material for Retry packets is fixed, while Initial packets use the Destination Connection ID from the first client packet, which is not encrypted. Therefore, neither packet type has confidentiality or integrity protection. The remaining packets, however, are protected by keys and algorithms acquired from the handshake, thus having high integrity and confidentiality protection. On top of payload protection and an AEAD, these packets also have header protection for fields that are not necessary for on-path elements to know.

The data length field present in certain packets can be used to merge multiple packets into a single datagram for more convenient processing, however coalesced packets should all have the same connection identifier due to routing concerns. Every type of packet, excluding Version Negotiation and Retry, has a packet number, which exists in one of three packet number spaces: initial, handshake, or application data. These spaces support numbers up to $2^{62} - 1$, and provide a processing context for their respective packet types.

Packet numbers are monotonically increasing and unique per packet number space - a connection must be terminated in the case that the packet number space is exhausted. Depending on which header type a packet number is presented in, it is encoded in 1 to 4 bytes. Most QUIC packets' payload consist of one or more frames of various types - a single frame should fit in a single packet. Certain frames have restrictions in which packet number spaces they may be sent in, for example frames related to data transfer may only appear in the application data space. While an exhaustive list of frame types and their functions can be found in Section 19 of the QUIC Transport Draft [6], some key frame types are briefly detailed below:

- PADDING frames only consist of their identifier, 0x00, and can be used to increase packet sizes when necessary. This is useful if the initial packet does not meet the minimun size requirements, for example.

- PING frames, like PADDING frames, have no content, only their respective type identifier: 0x01. PING frames are ack-eliciting, meaning that packets containing them have to be acknowledged, and as such they can be used to probe connection state or keep connections alive.

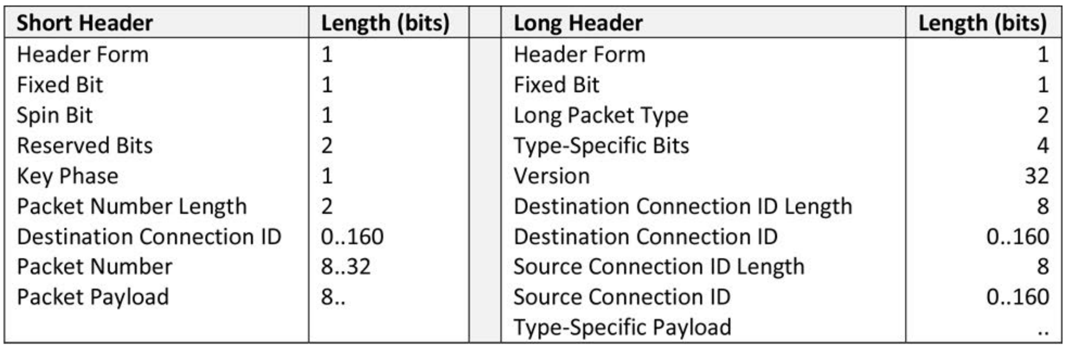
- ACK frames contain a varying number of ACK ranges, which indicated acknowledged packet numbers as a range. ACK frames are specific to one of the three packet number spaces, and once received, irrevocable. Only packets with a packet number can be acknowledged. ACK frames have two distinct types, 0x02 and 0x03, where the latter indicates the inclusion of ECN data.

- STREAM frames are used to create new streams and carry the respective data. The frame type depends on the presence of three different fields: offset, length and FIN, and thus can have a value between 0x08 and 0x0f. In addition to these fields, a STREAM frame has a stream identifier, and stream data.

- CRYPTO frames are used to transmit cryptographic handshake data, and are of type 0x06. They are otherwise similar to STREAM frames, except that they do not have a FIN bit, nor are the offset and length fields optional. They also do not include a stream identifier, and are not flow-controlled.

3.2.1 Formats

Both header type formats mentioned in Section 3.2 are shown in Figure 5. The header form field describes the header type, a value of one in the fixed bit-field indicates a valid packet, type specific bits are determined by packet type, the spin bit is an optional field containing latency information, and the key phase indicates the correct decryption keys to use, whereas the rest of the fields are self-explanatory. Packets with long headers are used in the handshake phase, while short headers are used by 1-RTT packets. Headers for Version Negotiation, Initial, 0-RTT, Handshake, and Retry packets have similar structures with certain alterations, and can be found in Section 17 of the QUIC Transport Draft [6]. All but Version Negotiation packets essentially use slightly modified long headers.



Short Header	Length (bits)		Long Header	Length (bits)
Header Form	1		Header Form	1
Fixed Bit	1		Fixed Bit	1
Spin Bit	1		Long Packet Type	2
Reserved Bits	2		Type-Specific Bits	4
Key Phase	1		Version	32
Packet Number Length	2		Destination Connection ID Length	8
Destination Connection ID	0..160		Destination Connection ID	0..160
Packet Number	8..32		Source Connection ID Length	8
Packet Payload	8..		Source Connection ID	0..160
			Type-Specific Payload	..

Figure 5: QUIC short and long headers.

3.2.2 Packetization, reliability and datagram size

In order to increase performance, packets should include as many frames as possible; frames from multiple streams can coexist in a single packet as well. For maximum efficiency, implementations may even decide to wait some time to collect larger bunches of data to send, however these wait times should be decided upon carefully to avoid negatively affecting performance. It should be noted, however, that coalescing packets from multiple streams will cause all of said streams to get blocked if such packets are lost.

Packets should only be acknowledged when packet protections have been removed, and all frames processed. This does not mean that the data in said packet has to have been consumed, however. The ackowledgement should be done via ACK frames if ack-eliciting frames are present; receiving acknowledgement of an unsent packet number should be considered a protocol violation and treated accordingly.

Whenever a connection is first established, an endpoint should advertise a maximum ack delay as a transport parameter, and adher to it to the best of its ability to avoid impacts to RTT and unnecessary retransmissions. Additionally, acknowledgements should be sent immediately in case of Initial and Handshake packets, ECN marked packets, as well as whenever gaps are detected in packet numbers. ACK frames should be sent sparingly, however, as they are not congestion controlled - the ackowledgement frequency is left at the discretion of the implementation. When deciding on this frequency, it is important to note that an ACK frame needs to fit in a single packet, therefore multiple large ack ranges resulting in long waits may not be ideal. Any intentional delays between receiving and acknowledging a packet should be communicated via the ack delay field. Packets should only ever contain frames from the same packet number space - this also applies to acknowledgements.

When it comes to datagram size, QUIC should not be used in network conditions where the maximum datagram size is less than 1200 bytes, where the size refers to QUIC headers and payloads within a datagram. Therefore, all Initial packets have to be padded or coalesced to this minimum, as they are discarded otherwise. This will also confirm that the network path supports the correct packet size. Larger Initial packets may be sent, but this might also lead to processing problems. The datagrams must not be fragmented at the IP layer. Additionally, IP packet size of 1280 or more is assumed to be available, whether IPv4 or IPv6, which should not be a problem for modern IP implementations. Higher sizes are recommended, if any extensions for IP protocols are desired or necessitated. If protocols that can measure maximum datagram size in a given network path are available, they should be used and the maximum datagram size set accordingly. If not, endpoints should stick to the minimum value of 1200 bytes.

3.3 Streams

In order to achieve faster and more fluid data transfer, QUIC is designed around using streams. As mentioned previously, streams are sequences of elements made available over time. In the context of QUIC, streams are abstractions for ordered byte-streams, similar to TCP. Streams are created by either endpoint when sending data, with support for both uni- and bidirectional streams, meaning that data transfer can occur either in one direction only, or both ways. Stream management also includes ending and cancelling streams, as well as flow control. Additionally, streams can be prioritized with different values, however a method for exchanging these values is not provided. Streams and stream-related processes are also designed with minimal overhead in mind - a single STREAM frame can include everything required for a "complete" stream: opening and closing, as well as the data itself. However, streams can also persist throughout the entire connection if necessary.

Since QUIC operates on top of UDP, streams are completely independent from each other - data between separate streams can be transferred concurrently with no blocking. However, QUIC does not ensure proper ordering of data between separate streams. While there are no hard limits for streams or data transfer, everything is still governed by the flow control rules discussed in Section 3.4. An endpoint cannot send data to a stream, if it is not withing verified flow control limits. In order to make multiple distinct streams possible, QUIC uses unique 62-bit integers as stream IDs to identify them within any given connection - these cannot be reused within the same connection. These IDs are encoded as variable-length integers, so that smaller values do not take the same amount of space as larger ones. The two most significant bits display the integer length, based on the binary logarithm, resulting in lengths of 1, 2, 4, or 8 bytes. The remaining bits contain the encoded integer in network byte order. The two least significant bits of the ID are used to describe different connection types: whether the stream was initiated by client or server, and whether it is uni- or bidirectional. Stream IDs for streams of the same type must increment and be used in order to ensure consistency.

To achieve ordered data, STREAM frames can be associated to their respective streams via their Stream IDs, and the data they carry ordered accordignly based on an offset value. If data arrives to an endpoint out-of-order, the endpoint must be able to buffer said data up to flow control limits. If data with duplicate offsets is received by an endpoint, it can be dropped as long as it does not differentiate from previously received data. In a case of different data with same offset values, a protocol violation may be raised. While ordered data delivery is mandated by QUIC, the option for providing out-of-order delivery is left at the discretion of the implementation.

Both endpoints connected to a stream have a set of operations available to them; the sender can write data, as well as end or reset the stream, whereas the receiver can read or stop reading data. Both endpoints also have the ability to request information on stream state changes. To further illustrate the operation of both senders and receivers, state machines showing the stages a stream goes through from both points of view are shown in Figure 6. Unidirectional streams use one of the shown state machines depending on their role, whereas bidirectional streams use both at both ends of the stream.

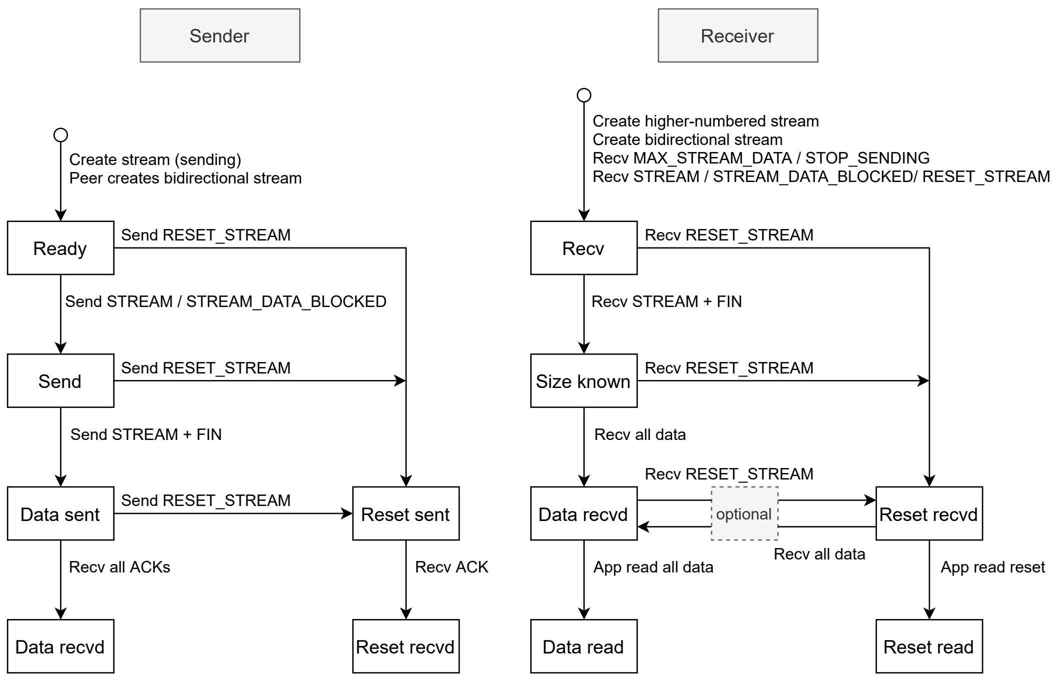


Figure 6: QUIC sender/receiver state machines.

The "Ready" state of a sender represents a new stream capable of sending application data. This state can include data buffering. When the first STREAM or STREAM_DATA_BLOCKED frame is sent, the state changes to "Send" - it is possible to only allot stream IDs after first moving to this state, which can improve stream prioritization. The "Send" state encompasses both data transfer, and retransfer, using STREAM frames. Flow control rules apply, and the endpoint can announce if it is blocked from sending data due to said rules via specific frames. After all data, including the FIN bit, is sent, the state changes to "Data sent", wherein data will only be retransmitted if necessary. When the receiver acknowledges all data sent, the sender enters its terminal "Data recvd" state. The stream can be terminated from any non-terminal state by either endpoint, which results in a reset frame and the "Reset sent" state. When the reset is acknowledged, a terminal state of "Reset recvd" is reached. A sender may only send STREAM and STREAM_DATA_BLOCKED frames in non-terminal states - this means these frames cannot be sent after a reset frame has been sent.

As opposed to the sender, a receiving part of a stream is created upon receiving one of the following frames: STREAM, STREAM_DATA_BLOCKED or RESET_STREAM. For a bidirectional stream, a receipt of MAX_STREAM_DATA or STOP_SENDING will also create a receiver. The initial state for a receiver is "Recv", in which the data is received in the form of STREAM frames. A receiver may also receive STREAM_DATA_BLOCKED frames, which indicate that the sender is blocked by flow control limits. This data is buffered and reassembled as necessary. The receiver will inform the sender of open buffer space via MAX_STREAM_DATA frames, which can only be sent in this state. When the FIN bit is received, the state changes to "Size known", which is the counterpart for the sender's "Data sent" state in the sense that only retransmissions are processed. Once everything is succesfully received, "Data recvd" state is entered and all subsequent data can be dropped. Once the application has received all the stream data, the receiver enters the terminal "Data read" state. As with the sender, the stream can be reset at any non-terminal state. In the case that this happens after all data has been received, or vice versa, the implementation is free to operate as it chooses.

3.4 Control and recovery

This section covers the methods a QUIC endpoints has at its disposal to control the rate of data transmission based on its capabilities in regular situations, when the network is under heavy load, as well as in situations where packet loss has occured.

3.4.1 Flow Control

To ensure that connections are not going to overload an endpoints processing capabilities, the connections need to be flow controlled, which means that endpoints impose certain limits to connections that they know they can handle. Since QUIC is a stream-based protocol, flow control needs to be applied to both connections as a whole, as well as individual streams, to ensure connections can be processed properly, and streams allocated fairly. To this end, the streams that make up a connection are flow controlled both separately against each other, as well as a whole in a singular context. CRYPTO frames, however, are not subject to regular flow control. The normal flow control operates by endpoints announcing receivable byte limits, or flow control credit, as transport parameters for any one stream, or a combined limit for a connection, as well as a cap for simultaneous streams per peer. These limits can be used to maintain fairness between multiple separate connections, as well as the individual streams within each of them, and can be *increased* by specific frames during the connection - advertising smaller limits has no effect.

It may be prudent for implementations to account for packet loss and delays when updating flow control limits by sending multiple instances of update frames, or sending them earlier than strictly necessary. To avoid excessive overhead, the update frequency and windows sizes should be carefully calibrated to match possible resource commitments. For further optimization, it may be good practive to avoid sending flow control packets on their own if possible. Violating advertised flow control limits results in a connection termination, therefore instead of sending more data, a blocked sender should announce their state to the receiving endpoint via special frames indicating their status. No matter how a stream is closed, the sender should always notify the receiver of the final stream size so they can compare and update their connection level flow control accordingly. No data should be sent, nor processed after this.

3.4.2 Congestion Control

For congestion control purposes, QUIC provides generic signals for supporting various sender-side algorithms. Therefore, any number of congestion control algorithms can be used, however any such algorithm should abide by guidelines set in the third section of RFC 8085 [37]. While packets with nothing but ACK frames are not considered in-flight, and thus are not congestion controlled, they may be used for congestion control purposes since QUIC can detect the loss of such packets.

While QUIC can support a variety of different congestion control algorithms, a default sender-side algorithm is also specified in the QUIC Recovery Draft [28]. Whereas the minimum congestion windows for QUIC is recommended to be twice the maximum size of a single datagram, or two packets, it is recommended that the initial window be set at ten packets. Changes to the maximum datagram size throughout the connection should affect the congestion window accordingly. As the default congestion controller is based on TCP NewReno [9], it has the following three states:

- Slow Start: This state persists while the congestion threshold, which is initially infinite, is greater than the congestion window. During slow start the congestion window is increased directly by acknowledged bytes, until a packet loss or ECN triggers a recovery period. Slow start is only re-entered during persistent congestion, meaning a period of time exceeding a set duration, throughout which all sent packets over all packet number spaces are lost, including at least two ack-eliciting packets. An rtt sample must also exist before the loss occurs.

- Recovery Period: A recovery period begins with updating the congestion threshold to half of the current congestion window; this can be done instantly, or gradually. Further packet losses or increases in ECN values do not re-trigger recovery, nor have any effect on the congestion window while the state persists. Whenever any packet sent during a recovery state is acknowledged, congestion avoidance begins.

- Congestion Avoidance: While in this state, the increases to congestion window must be limited to one datagram size per acknowledged window. This state persists while the congestion threshold is smaller than the window, and no packet loss or ECN increases are detected.

While endpoints may ignore packet loss of packets that they don't have decryption keys for, if a packet is acknowledged in a specific packet number space, packets that have been sent after said packet cannot be ignored. The congestion controller should also not block probe packets, even if they temporarily exceed the congestion window. Senders should use the congestion controller to pace their packets, excluding any that include ACK frames, and avoid bursts that the network cannot handle. The initial congestion window is a good limit for bursts, if more accurate information cannot be acquired. The congestion window should not be increased, if it it under-utilized.

3.4.3 Recovery

While the recovery algorithms for QUIC can be similar to certain TCP algorithms, the following key protocol differences cause some deviation:

- Separate encryption levels have their own packet number spaces, aside from 0-RTT, and 1-RTT keys.

- Packet numbers increase monotonically, with higher numbers indicating later transmission. Retransmission of a previous packets has its own independent packet number, removing ambiguity from acknowledgements.

- QUIC ends its loss epoch after any received packet after a loss, instead of waiting for the original packet, resulting in more accurate congestion windows as the updates can happen faster.

- QUIC does not allow packet acknowledgement reneging; if a packet is acknowledged, the acknowledgement cannot be revoked.

- QUIC supports more than three ACK ranges.

- Endpoints measure time between receiving a packet, and sending the corresponding acknowledgement.

- QUIC replaces static timeouts with a timeout based on receiving acknowledgements on some ack-eliciting probe packets, where the timer is based on a peer's maximum expected acknowledgement delay. Additionally, the congestion window is only collapsed upon declaration of persistent congestion, instead of probe timeout expiration, which reduces unnecessary reductions.

- The minimum congestion window for a connection is two packets, as opposed to one.

Since packet numbers never repeat in a given packet number space within any one connection, and increase monotonically, a significant amount of complexity is avoided in terms of having no need to separate normal transmissions from retransmissions. This also enables better RTT measurements, easy detection of false retransmissions, and universal Fast Retransmit.

QUIC data consists of various different frame types, which can trigger different behaviour based on their importance: CRYPTO frames have shorter timers, non-ack-eliciting packets are only acknowledged when mixed with ack-elicing packets, ACK and CONNECTION_CLOSE frames don't count towards congestion control limits, and PADDING frames contribute towards bytes in flight while not eliciting acknowledgements.

To estimate round-trip-time, an endpoint uses multiple samples of the difference between packet sending time and acknowledgement of said packet, alongside reported host delays. If no previous RTT data is available, the initial RTT should be set to 333ms, or optionally the values gained from path probing. The largest received acknowledgement number must be new to the node, and be an ack-eliciting packet. From these, it calculates a minimum and a weighted RTT and mean deviation. The minimum RTT is used to reject RTT samples that are implausibly small, and should be updated if applicable - host delays are not factored in here. The smoothed RTT is a moving weighted average of samples including host delays, whereas mean deviation describes variance.

QUIC handles loss detection via acknowledgements and probe timeout separately in each packet number space; a probe timeout sends some ack-eliciting probe datagrams to test the network path. Even though probe timeout is specific to a given packet number space, when triggered in one, each packet number space should be probed. QUIC's loss detection algorithm is inspired by multiple TCP algorithms, such as Fast Retransmit [38], SACK [39], and others. In QUIC, a packet is lost if it has not been acknowledged by the peer, is considered in-flight, and was either sent a certain packet or time threshold before an acknowledgement for a later packet. Similar to TCP, the initial recommendation for said packet threshold is three, and it should not be set *lower*. The corresponding time threshold should be set according to local timer granularity, and be adjusted by the smoothed and latest RTT values.

3.5 Connections

As QUIC is a stateful, connection-oriented protocol, a connection between two endpoints represents a shared state between them. A connection is initialized with a handshake, combining negotiations for security and transport parameters alike. While a handshake is generally required for a connection, there are some exceptions. When sufficient configuration or previous familiarity exists between two nodes, instantaneous data transfer, or 0-RTT, can occur without a handshake. Reversely, a server may also send data to a client before all cryptographic processes are complete. These exceptions can be implemented at the discretion of the application protocol, sacrificing security for speed.

3.5.1 Identifiers and establishment

In order to identify connections better, and to enable features such as connection migration, each connection endpoint selects a unique identifier associated with it, used by its respective peer - the method of selecting these IDs is implementation-specific. Each connection has multiple indentifiers to make it harder for observers to pinpoint data to said connection. To that end, the identifiers must also be such that they can't be used to infer the other identifiers. The packet header length influences whether both IDs are included (long headers), or just the destination (short headers). An ID of length zero can be used in cases where an identifier is not essential for routing, however multiplexing connections like this can cause issues when peer connection migration, NAT rebinding, or client port reuse is in use. The IDs also make QUIC connections resistant to changes in lower level protocols, that could result in packets being delivered to the wrong node.

Connection identifiers are associated with sequence numbers to assist in matching references to them. The initial connection identifier can be found in the Source Connection ID field of long headers, and its sequence number is either 0 or 1. Further connection identifiers can be exchanged via specific NEW_CONNECTION_ID frames, the sequence numbers of which are incremented by one. When an identifier has been issued, it is valid until specifically retired, meaning that any data sent with said identifier must be accepted at any time. Endpoints should provide each other a sufficient amount of unused connection IDs, however making sure to not exceed each other's limits; each retired ID should be followed by an unused replacement.

An endpoint can consume the pool of identifiers it has for a connection at will. When an identifier is no longer needed, the endpoint can communicate this to its peer with a specific frame, indicating that the identifier in question should be replaced with a new one. As connection identifiers are limited to a single source and destination address, the identifier should be changed in the case that either of these addresses change, whether it be due to migration or other reasons. If an endpoint needs to retire multiple already issued identifiers, it can do so via a frame indicating new identifiers, with the "Retire Prior To" field set accordingly. The peer must then retire requested IDs and assign new ones. Endpoints should limit retired identifiers that have not been confirmed by peers, as well as only forget IDs after they have been retired.

Upon receiving packets, an endpoint will try to associate it to an existing connection, or failing that, create a new connection if possible. In a case where neither is possible, a stateless reset can be sent back to signal an unusable connection. Even if a packet matches a connection, it may be discarded if inconsistencies are detected, such as different protocol versions or inability to properly use existing cryptographic keys.

Client endpoints will only accept packets with either valid destination identifiers, or valid port and address information if zero-length IDs are accepted - all other packets are discarded as clients cannot create new connections. While a client may decide to keep packets with encryption it does not yet have the keys to, in case said packet has been delivered out-of-order, packets with different version numbers must be dropped. If a client supports multiple QUIC versions, it should pad the first packet it sends to the largest minimum datagram size, as the server's version negotiation protocol is size-dependent. In terms of processing a potential version negotiation packet, if a client only supports a single QUIC version it should terminate the connection upon receiving such a packet, unless a packet from the same peer has already been succesfully processed, or the packet described the version the client uses. In both exceptions version negotiation packets should be dropped.

If a server receives a packet of sufficient size indicating an unsupported version, the server should engage in version negotiation before establishing a proper connection, so as to not retain state. The following version negotiation packet should include all supported versions. It should be noted, however, that a version may be different enough where the server cannot even interpret the original packet properly. Conforming packets with supported versions are either associated to an existing connection, result in a handshake and initialization of a new connection, or are followed by an error if the server cannot accept new connections. 0-RTT packets may be buffered in wait of Initial packets, unprompted client handshake packets are dropped.

Any QUIC implementation must, at the very least, implement specific operations for endpoints. Client nodes must be able to open a connection, and server nodes must be able to listen for incoming connections. Both node types can also decide the minimum amount of streams of each type to accept initially, manage flow control for both streams and the connection, identify the handshake status, prevent silent closure of connections, and have the ability to immediately close a connection. If early data is supported, clients must also be able to enable it and know whether it has been accepted or rejected, whereas the server must be able to embed and retrieve application-controlled data from TLS resumption tickets.

As mentioned previously, QUIC uses a combined handshake including both security and transport parameter negotiation simultaneosly. As with many other features of QUIC, the handshake has a specific frame type, CRYPTO, used in transmission. The offsets for CRYPTO frames always start at zero. It should be noted, that since CRYPTO frames are not subject to flow control, any endpoint should be capable of buffering a sufficient amount of out-of-order CRYPTO frames. The current QUIC version, identified as 0x00000001, uses TLS as its security protocol of choice. The handshake must provide a specific set of properties to ensure proper security. This includes a key exchange where at least the server is authenticated, where keys for each connection are unique, and keying material can be used for all packets. Additionally, the transfer of transport parameters and negotiation of application protocol must be authenticated. Finally, server transport parameters must have confidentiality protection. The handshake will also include encoded transport parameters. These are made by both endpoints independently of each other, and each endpoint must comply with their own parameters after they are declared. These parameters should only ever be declared once, and are validated by the peer after handshake is complete. If multiple application protocols are offered, all must comply with the chosen transport parameters. 0-RTT connections use previously negotiated parameters, until a new handshake is completed. Any parameter unknow to the endpoint should be ignored.

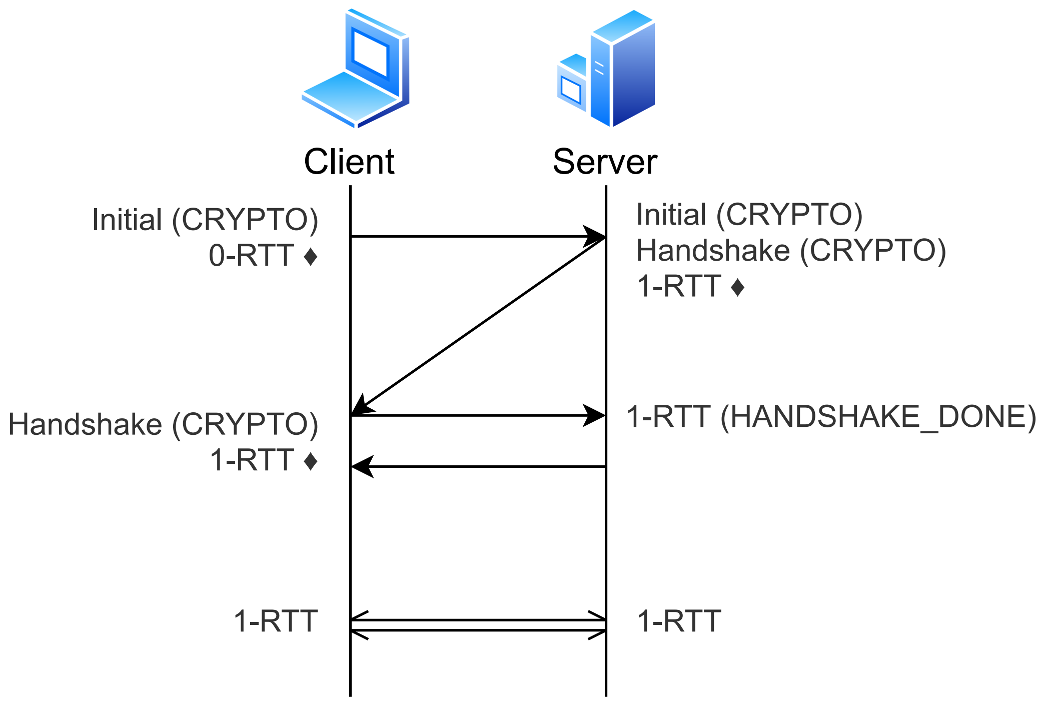


Figure 7: QUIC handshake.

As shown in Figure 7, the QUIC handshake is initiated by the Initial packet from a client, containing a CRYPTO frame. The server then responds with its own initial data, acknowledgements, as well as the Handshake frame(s). The client does the same, excluding the initial data, and after a final server ACK the connection is live. Each step marked with a diamond also has the potential to transmit application data. The connection identifiers mentioned previously are also exchanged in the handshake, as each peer sets their desired identifiers in the Source Connection ID field of the long header. Before the client knows the server's desired ID, it should use the same, 8-bit long unpredictable value, in its place - this is used to generate packet protection keys for Initial data. 0-RTT data uses the same values as the Initial packet.

After both peers have succesfully exchanged identifiers on a connection, any packets with different identifiers should be dropped. The IDs should initially only be changed once from the first Initial packets, any further changes are only allowed from NEW_CONNECTION_ID frames. The initial IDs are included in transport parameters for authentication - the received values must match the values sent. Failure to match said values, or deliver the required transport parameters, should result in a transport parameter error or protocol violation.

3.5.2 Migration

Due to the fact that connections are associated with unique connection identifiers, they are able to go through connection migration of IP address and port if necessary, assuming that the connection has been properly initialized via a handshake. If the ability to migrate is declared as disabled via transport parameters, then packets arriving to an endpoint from differing IP addresses must either be dropped, or the new path validated and migrated. Even in cases where migration is declared enabled, any changes in network paths must be validated. Failing this, a connection must be terminated. It is noteworthy that servers never initiate migrations.

Probing a new network path can be done via probing packets, including the following frames: PATH_CHALLENGE, PATH_RESPONSE, NEW_CONNECTION_ID and PADDING. The migrating endpoint may exempt any probing packets used for finding new network paths from its congestion control algorithm, so they do not affect send rates of the original connection. A connection can only be migrated to a new address if a validated path to it exists.

To initiate a connection migration, an endpoint must send any non-probing packets. After such packets are received, both endpoints validate each other's addresses - the only case in which this validation does not need to happen is if the new address is recently known to the server, and already validated. To prevent amplifying attacks, connection rates to unvalidated addresses are limited, whereas to prevent on-path attacks, failed address validations should result in reverting to the last validated address. After the connection has migrated, the migrating endpoint will reset its congestion controller, RTT estimate and ECN capability. This is to avoid the characteristics of the old network path affecting the new paths congestion control. The only case where these metrics should not be reset, is if the only change in address is the port number.

For privacy reasons, connection IDs cannot be reused even when probing new paths from new addresses, nor when responding to said probes or interacting with migrated endpoints - endpoints should make sure to have not exhausted their pool of unused IDs before a migration. Due to the same reasons, migrations should not be done with peers with zero-length identifiers.

A server may advertise a preferred IP address in its transport parameters, different to the one the client initially connects to. Since server migration is not supported by this QUIC version, a client may decide to connect to this preferred address via normal procedure if it wishes to. A server cannot switch to its preferred address before path validation is done and it receives a non-probing packet to said address from the client. If a client address migration is triggered before the migration to a preferred IP is complete, the client should try to validate both server addresses from its new location in case the preferred address validation fails. The server should of course validate the new address as well.

3.5.3 Management

To avoid amplification attacks, any unvalidated address should not be sent more than three times the received amount of either validated or discarded data. For this reason, clients should ensure that their initial packets are sufficiently large to allow the server to respond properly - datagrams should be at least 1200 bytes, using padding as necessary. To avoid a deadlock in the handshake phase, clients should probe the server if they stop receiving packets before the handshake is fully complete.

For connection establishment, validation happens via confirmation of a processed Initial packet - a receipt of a handshake key protected packet can be considered as such confirmation. Another method of confirmation is the use of recognized connection ID with 64 or more bits of entropy. For clients, packets protected by keys derived from their initial destination ID indicates a valid address, as well as finding said ID in Version Negotiation or Retry packets. If a server wants to validate a client address before handshake, it can do so via tokens that the client echoes back. These tokens may also be made valid for future connections, however they should have a expiration time.

In order to verify that a connection is reachable, path validation is used by both endpoints before any other data is sent - this also protects against address spoofing. While this method is generally used during connection migration, it is not limited to such a situation and can be used for other purposes as well. Probing packets for validation should contain specific path frames with unpredictable data, as well as use new connection IDs - acknowledgements of these packets are not enough to validate a path in the reverse direction. Upon receival, this data should be echoed back through the same network path. Multiple paths can be probed simultaneously, assuming sufficient amount of unused connection IDs.

Detected errors with a connection should be communicated to the respective peer with the relevant error code(s). In the case of terminal connection errors, CONNECTION_CLOSE frames with specific frame types matching the situation should be used. While these close the connection, they may need to be resent if lost to avoid continued data from the client. If an error only affects one stream, said stream should be reset instead of closing the entire connection. While the application protocol error codes are the responsibility of the implementation, the transport error codes can be found in Section 20.1 of the QUIC Transport Draft [6].

3.5.4 Closure

Terminating a QUIC connection can occur in multiple ways, the first of which is idle timeout. An idle timeout threshold can be specified as a transport parameter during the handshake by either endpoint, and upon reaching the minimum of either threshold without activity, a connection will be silently terminated. Activity in this case is either receiving and processing a packet from a peer, or sending an ack-eliciting packet. An endpoint can test if a connection has timed out, or just reset the timer, by sending non-data frames like PING before sending proper data if it is concerned about timeouts. However, deferring timeouts via PING frames should be done sparingly to avoid unnecessary performance impacts. It should be noted, that middleboxes may lose state faster than a threshold set by either endpoint.

The second connection termination method is immediate closure. This is done via a CONNECTION_CLOSE frame, terminating the connection and all streams at once. An endpoint initiating a close via sending the frame will enter a closing state, whereas the endpoint receiving the information will enter a draining state. Any protocol violations by either endpoint should result in immediate closure of the connection, however this method can also be used following a graceful shutdown by the application. The terminal states of each endpoint following an immediate closure should last long enough to deal with any residual packets, and close cleanly. An endpoint in a closing state is *only* able to attribute packets to its connection, and respond to them with CONNECTION_CLOSE frames at a limited rate. A closing should proceed into a draining state when receiving a CONNECTION_CLOSE frame. An endpoint in a draining state acts similarly to a node in closing state, except it is not allowed to send any packets, excluding a single CONNECTION_CLOSE frame upon entering the state. If a connection closure needs to happen before a handshake has been completed, the closure frames should be included in Initial and Handshake packets whenever possible to ensure they are received and processed.

The third method of closing connections is a stateless reset. This type of closure occurs when an endpoint cannot for whatever reason access connection state. This should not be used to handle errors in active connections, it is a last resort measure. A stateless reset is triggered by sending a 16-byte token in a specific field of the NEW_CONNECTION_ID frame, therefore tying it to the connection identifier - a token is only valid if the corresponding ID is valid. A peer should then return the token in a stateless reset datagram, ending in the token. This datagram should be less than three times the size of the packet containing the reset token to avoid amplification. If the last 16 bytes of a datagram can be matched to any valid tokens from its sender, the receiving endpoint should go into a draining state.

3.6 Security

QUIC is designed to be a secure transport protocol, and as such one of its key security features is the integrated TLS 1.3 handshake, with some additional modifications. In addition to the confidentiality and integrity brough upon by the keys derived from the handshake, the handshake is also used to communicate and authenticate transport parameters in a TLS extension, which describe certain connection qualities. For a handshake to be considered complete, the TLS stack must have processed both the outgoing and incoming Finished-messages. A server should communicate this to a client via a specific frame.

If a transport parameter extension is not present in the appropriate messages, the connection should be terminated. While these parameters are available before the handshake is completed, and can be used, it should be noted that they are not authenticated until a finished handshake. This TLS extension should not be used with protocols other than QUIC. Another change to standard TLS operation procedure is the exclusion of the EndOfEarlyData messages - these should never be sent. Finally, the TLS Middlebox Compatibility Mode is not used in QUIC; requesting it should result in a protocol violation.

Instead of being separate entities, TLS and QUIC are integrated with each other as shown in Figure 8. TLS uses certain QUIC features to send and receive messages reliably, whereas QUIC relies on packet protection keys and state changes in return. Instead of using TLS application data records, QUIC applications send their protected data directly via QUIC frames; handshake data is transmitted in CRYPTO frames.

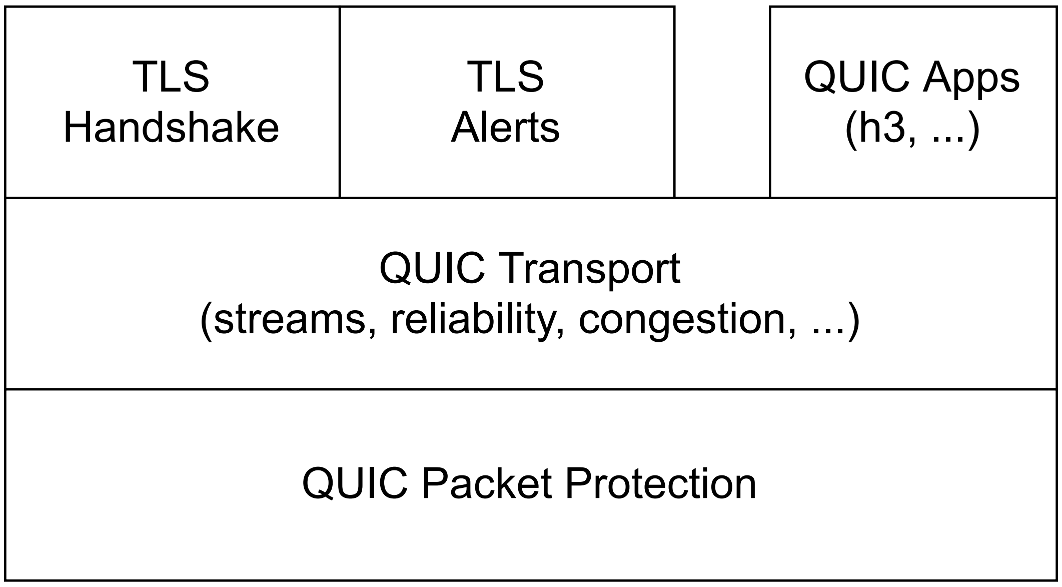


Figure 8: QUIC Layers.

Unless another protocol is mutually agreed upon, the application protocol negotiation is conducted via ALPN (*Application Layer Protocol Negotiation*). If the negotiation were to fail, either because no protocol is offered or endpoints cannot match protocol versions, both endpoints are responsible for immediately terminating the connection. All QUIC packets, excluding Version Negotiation packets, are protected by authenticated encryption acquired by the handshake. This ensures that payloads can only be processed by the endpoints who were part of a succesful handshake and have the correct keys. It is highly unlikely that an attacker would be able to create or modify packets in a way where either endpoint in a connection would consider them valid, since any changes to authenticated sections will result in the packet being ignored. The worst that an on-path attacker should be able to achieve, is to block or slow the connection by either delaying the packets, or tampering with them. This can always be solved by migrating to a new network path, assuming that such path is available.

In order to successfully use the 0-RTT feature that QUIC provides, a client must remember a set of key parameters from a previous connection, including TLS state, transport parameters, and application protocol along any related information. All this information must originate unaltered from a single connection. Certain constraints apply to 0-RTT usage, such as a seven day time limit from the original connection. A server may also advertise in its transport parameters, that it does not support 0-RTT data, or it might reject the data because its configuration has changed since the last connection in a significant way.

To counter amplification-type attacks, QUIC validates address ownership of endpoints, and imposes send limits before such validation is completed. New network paths during connection migrations are also validated before a large amount of data is allowed to be sent. The Retry packets provide servers a cheap way of exchanging tokens and validating client addresses, which guards againt denial-of-service attacks. Additionally, QUIC will also discard any unauthenticated packets, aside from a limited amount of ICMP and stateless reset packets. A more in depth list of considerations and countermeasures for specific attacks can be found in Section 21 of the QUIC Transport Draft [6]

4 QUIC implementations

This section will cover the currently known QUIC implementations briefly, with a more detailed look into the three chosen for further analysis. The reasoning behind the choices is also explained.

4.1 Overview

The QUIC IETF working group maintains a GitHub page listing all of the current implementations[14]. There are currently over twenty different implementations with varying design goals, covering multiple programming languages, most of which offer support up to draft 29 [40] or greater in terms of QUIC specifications - for the purposes of the following overview, this is true unless otherwise specified. All implementations are encouraged to take part in interoperability testing, and a selection of implementations also provide a number of public test servers and pre-made client/server programs for local testing in various conditions. Most implementations link to their own separate GitHub page or an equivalent, however the level of documentation varies greatly between them. The QUIC implementations listed by the working group, excluding the ones chosen for the thesis, are as follows:

- AppleQUIC is listed as a QUIC implementation, however no links to a page containing code or documentation are provided, nor can any be found with a simple web search. The implementation is supposedly programmed in C and Objective-C, and supports QUIC draft 27. No further information, or testing methods are provided. The notion of Objective-C as a programming language indicates this implementation is likely related to OS X and/or iOS [41].

- Ats, which is for the Apache Traffic Server. As such, it is not a standalone implementation, but a built-in feature of the Apache project. For testing purposes this version does support four different SSL libraries, and does provide some premade testing tools, however these seem fairly limited based on the documentation. Ats is implemented in C++, and has multiple public test servers listed.[15]

- Chromium's QUIC implementation, which is the version Chrome is already using to transfer a large part of their traffic. As with ats, this is not a standalone implementation, but an integrated one, so accessing sample programs and testing requires the Chromium source code and a few additional steps. The documentation and sample programs seem fairly limited, and are noted to be intended for integration testing. Chromium's QUIC is programmed in C and C++, and features a few public test servers.[16]

[14]https://github.com/quicwg/base-drafts/wiki/Implementations
[15]https://cwiki.apache.org/confluence/display/TS/QUIC
[16]https://www.chromium.org/quic/playing-with-quic

- F5, developed by a US based tech company. No documentation or code is provided, but it is said to be a part of the F5 TMOS, which is a custom-made real-time operating system consisting of a collection of different modules - the QUIC implementation is therefore likely one of these modules. This implementation supposedly supports draft 32, and is programmed using C.

- Haskell QUIC is an implementation in its namesake programming language, mainly developed by Kazu Yamamoto. The documentation is very brief, mainly consisting of a few blog articles relating to the project and information on what is included. The readme also notes that unreleased packages are required for properly building the implementation, which makes testing impossible for the time being.[17]

- Kwik is currently the only Java-based QUIC implementation on the list. While it does support the latest drafts, it is not yet feature complete in terms of the QUIC specification. It is also only a client implementation, and thus provides no server functionality, although it does claim to be among the best client implementations participating in the interoperability testing. Kwik is developed and maintained by Peter Doornbosch, and includes both a library, and a command line tool for testing.[18]

- MsQuic, which is Microsoft's C-based QUIC implementation, designed to be cross-platform and general purpose. It claims superior optimization to other QUIC implementations, alongside an asynchronous IO, RSS support, and UDP send/receive coalescing support. MsQuic supports the latest drafts, has a few public test server and a very extensive set of documentation, however no existing test setup or sample client/server is provided.[19]

- Mvfst is an implementation by Facebook, mainly programmed in C++. Unlike a lot of the other implementations, it has supposedly been tested at scale on mobile platforms and servers, and has support options regarding large scale deployment. The implementations depends largely on two other libraries by Facebook, and comes with an automated build script and sample programs for testing. Unfortunately while mvfst has a sample HTTP/3 implementation called proxygen with a solid suite of functionality and extensive documentation, it appears to be woefully behind in terms of version support at only draft 23 - otherwise it would've been a strong contender for further testing.[20]

- Neqo is a QUIC implementation by Mozilla in Rust. While the development appeared halted throughout the writing of this thesis, the implementation has been recently updated with support up to v1 of QUIC. Provided documentation is fairly brief, with little information on protocol specifics.[21]

[17]https://github.com/kazu-yamamoto/quic
[18]https://bitbucket.org/pjtr/kwik/src/master/
[19]https://github.com/microsoft/msquic
[20]https://github.com/facebookincubator/mvfst
[21]https://github.com/mozilla/neqo

- As with a few previously mentioned web servers, Nginx also has their own integrated C-based QUIC implementation. Latest drafts are supported, however no client implementations obviously exist. Another Nginx implementation by Cloudflare exists as well, although it is a patch that depends on another QUIC implementation.[22]

- Picoquic is a C-based minimalist QUIC implementation. It consists of test tools, with a goal of providing feedback for the development of the QUIC specification, as well as exploration of the protocol outside of HTTP, such as with DNS. Some demo applications are provided, however the documentation is currently still in progress. A framework for plugin exchange between servers and clients, called PQUIC, is built on top of picoquic.[23]

- Quant is a C-based QUIC implementation for research purposes, mainly developed by Lars Eggert, a chair of the QUIC working group. It uses a UDP/IP stack called warpcore, also developed by Eggert, which supports the netmap I/O framework and some additional IoT stacks. This means Quant supports embedded systems as well, which is definitely a stand-out feature. HTTP/3 functionality is not implemented, however.[24]

- Quiche is a Rust-based implementation developed by Cloudflare, and used in their edge networks. It features a minimal low level API for packet processing and handles the connection state. It is also designed to be reusable in different contexts, and comes with design considerations to make integration with other languages easier.[25]

- Quicly is an implementation intended to be used with the H2O HTTP server, which is a optimized HTTP server with the goal of providing faster response times with less CPU usage. Two public test servers are available, however quicly only supports up to draft 27. It is programmed mainly in C.[26]

- Quinn is a purely Rust-based implementation, supporting the latest drafts. Its feature include simultaneous client/server operation, pluggable cryptography, and an asynchronous API among others. Quinn comes with a HTTP/3 support and sample client and server applications.[27]

- Quic-go is an implementation developed purely in the Go language. HTTP/3 support exists, as well as sample applications for testing.[28]

[22]https://github.com/quicwg/base-drafts/wiki/Implementations
[23]https://github.com/private-octopus/picoquic
[24]https://github.com/NTAP/quant
[25]https://github.com/cloudflare/quiche
[26]https://github.com/h2o/quicly
[27]https://github.com/quinn-rs/quinn
[28]https://github.com/lucas-clemente/quic-go

4.2 Implementations chosen for analysis

The following aspects were considered when deciding upon suitable implementations to test:

- The most important aspect was adherence to the latest QUIC drafts, and the speed at which updates to newer versions occurred. While this is less relevant now that most implementations support the latest update with major changes, draft 29, the development of the test framework started multiple versions ago when the protocol was still under constant change, and without a finalization date in sight.
- The existence of a premade server and client applications for testing. The reasoning behind this is mostly related to time concerns, as making multiple such testing applications would be infeasible considering the time constraints. Some further requirements also apply to these sample applications, in order to be suitable for the test scenarios:
  - Linux support for compatibility with the test framework.
  - HTTP support for sending/receiving data.
  - The ability for the client to request and receive files of varying sizes from the server.
- The amount of additional customizable parameters for either application was also considered as a bonus, as well as the inclusion of functional qlog [42] support, which provides a wealth of data in addition to the capture files and logs. Qlog is a standardized logging format for QUIC applications.
- Final considerations include some level of variety in programming languages used, and to a lesser extent, ease of use.

While it would obviously be ideal to compare as many implementations against each other as possible, this smaller selection of implementations was chosen to maintain a reasonable timeframe for the thesis. Additionally, while most of the implementations described in Section 4.1 fulfill some, or most, of the chosen criteria, very few of them actually check all the boxes. Since all of the chosen protocols do not support the latest drafts as of writing, the tests have been run with draft 29 versions of each for fairness. This should be very similar, if not identical, in terms of the actual protocol functionality compared to draft 34.

4.2.1 aioquic

The first implementation chosen was aioquic, which is currently the only Python-based implementation listed, although the cryptography has been implemented with C for performance reasons. Aioquic was chosen initially, because it met all of the requirements, and because the test framework was also developed in Python, making initial feature and integration testing relatively fast and easy. It is also a very interesting inclusion in a performance-sense, since Python is not a typical choice for a protocol with a heavy emphasis on being as fast as possible. While the aioquic repository includes eight contributors, it appears to be mainly developed by a single person - Jeremy Lainé. While aioquic does currently support up to v1 of QUIC, this update was quite recent and unfortunately happened after all of the tests had been run and results analyzed, with no time for a rerun. In terms of the rest of the criteria mentioned, aioquic provides the QUIC library, as well as a few different client/server pairs for testing - these include HTTP/3 versions, which achieve all the required functionality with minimal changes. Aioquic also provides a minimal TLS implementation with an ability to log traffic secrets, IPv4 and IPv6 support, and can provide qlogs if desired. Aioquic does not perform any I/O by itself; instead this is left to user to make the use of aioquic easier and more dynamic. Aioquic also includes a QUIC convenience API (*Application Programming Interface*), which is based on asyncio.[29]

4.2.2 LSQUIC

The second chosen implementation was lsquic, which is a C-based QUIC implementation by LiteSpeed Technologies, a US based tech company specializing in server software design. As with aioquic, LSQUIC fulfills all of the main requirements. Not only does it support the currently latest draft 34 (and v1), but it has typically been among the first few implementations to update whenever a new draft has been released. Additionally, it also provides a variety of premade test applications, including a client/server pair that supports dynamic file transfer. Unfortunately while qlog functionality is listed in the documentation, it has been outdated for a long time and doesn't seem to be a part of the example applications yet. While qlog support would be ideal, and the implementation was initially chosen with the hope that it would be updated before the final testing began, LSQUIC is still a very good candidate for the thesis. Alongside a GitHub page, LSQUIC also has a very extensive and detailed documentation page and a solid build guide. Multiple public test servers are also available with some different functionalities. According to the developers, LSQUIC is fast, flexible and production-ready. It supports almost every QUIC and HTTP/3 feature, including ECN, spin bits and NAT rebinding. A list of potential extensions, such as loss bits and delayed ACKs is also provided. With all this in mind, LSQUIC is one of the most feature-rich implementations thus far. Architecture-wise, LSQUIC does not use sockets, nor does it require the use of event loops. Connections are also kept in separate data structures and queues to prevent unnecessary processing.[30]

4.2.3 ngtcp2

The third, and final implementation is ngtcp2 - another C-based QUIC implementation, although this time developed mainly by a single person, Tatsuhiro Tsujikawa. While this implementation is also developed in the C language, the design philoso-

[29]https://github.com/aiortc/aioquic
[30]https://github.com/litespeedtech/lsquic

phies between a single developer and a company specialized in server software might provide some interesting divergence. As with aioquic and LSQUIC, updates to the latest QUIC versions have been quite swift, and ngtcp2 currently supports the latest drafts. Example HTTP/3 client and server applications are also provided, based on another project by the same developer - nghttp3. Finally, the implementation also comes with support for faster connection re-establishment via session files or tokens, functional qlogs, and a crypto helper library.[31]

[31] https://github.com/ngtcp2/ngtcp2

5 Research methods

This section includes a technical overview of the test framework, as well as the environment it was operated in. Additionally, the various test scenarios and their relevant metrics are presented.

5.1 Test environment

The testing framework, further elaborated on in Section 5.2, provides the following sources of information for analysis:

1. The full tshark capture files, as well as trimmed text versions for each individual test.
2. Logs for both the overall test scenario and server(s), as well as individual logs for each client of every test iteration.
3. The output of the client implementation(s), and TLS secrets if supported by server implementation.
4. Qlog files if supported by QUIC implementation
5. A graph file for each test containing various key information, such as iteration count, link properties and average connection metrics. The file also includes various graphs representing packet traffic between client and server nodes.

While the graph files are automatically generated upon each test iteration, they can also be regenerated manually for existing test data if deeper analysis is required on specific test cases. Multiple additional separate scripts also exist, most of which provide some further analysis, or more specialized graphs of the capture data. For the purposes of this thesis, these graphs and relevant information will be presented in a condensed and averaged manner in the interest of clarity.

The test environment was developed using Python 3, and uses Mininet 2.3.06d[32] as the network emulation tool. Python was chosen as the programming language due to familiarity, as well as enabling a relatively fast and efficient development process for a program of this size, in addition to providing the ability to make additional scripts quickly if needed. Mininet is the network emulator of choice due to a good mix of simplicity and robustness, with good integration options with Python-programs, including a full API. Mininet also allows quick and easy configuration of traffic control for links and nodes, and dynamic network creation via Python classes. The main downside of using Mininet is its lack of support for newer versions of related software, such as latest Python versions, or the latest release of the operating system the framework runs on. The framework resides on a single regular laptop running Ubuntu 18.04. Virtual containers are not utilized.

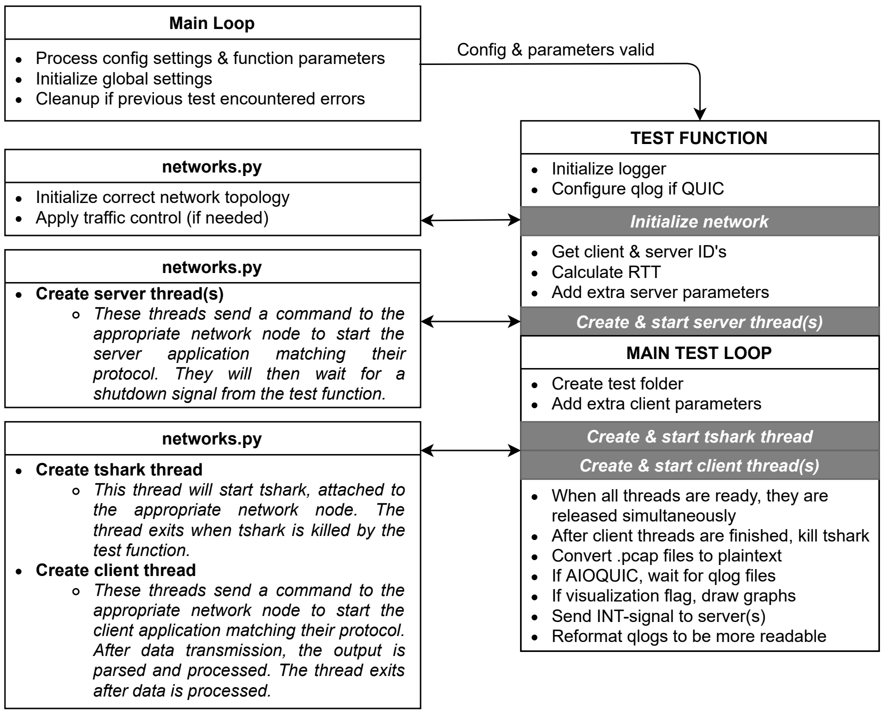
Due to a gap of updates for aioquic, only resolved after v1 of QUIC became available, all QUIC implementations are running their draft 29 compatible versions

[32]http://mininet.org/

to ensure a fair and compatible test environment. Judging by the changes in the drafts following draft 29, the results shouldn't vary greatly to later versions as no major protocol changes have occured. Any other relevant libraries required by these implementations are using the versions specified in each respective implementation's documentation for their draft 29 versions. The comparison implementation of TCP was built using AIOHTTP 3.6.2 and asyncio. TCP segmentation offload is disabled in all scenarios due its behavior in an emulated environment, such as data transfer being consolidated into massive unrealistic chunks in the tshark capture, as well as potential unrealistic effects on transfer speeds. Any libraries not part of the Python standard library used in the framework are running the latest version supported by Python 3.6.9 at the time of development. In order of achieving minimal program-related interference, all implementations run in modes with the least possible amount of debug options and printing. In terms of the test framework itself, no results are printed or processed while data is still being transmitted.

5.2 Test framework

Figure 9 shows a high-level depiction of the main functions of the testing framework, as well as a simplified workflow through a test. In addition to the functions and classes shown, the framework also includes some helper functions and a visualizer, which are not detailed further for brevity.


Main Loop
Config & parameters valid
TEST FUNCTION
networks.py
Initialize network
networks.py
Create & start server thread(s)
MAIN TEST LOOP
networks.py
Create & start tshark thread
Create & start client thread(s)

Main Loop
- Process config settings & function parameters
- Initialize global settings
- Cleanup if previous test encountered errors

networks.py
- Initialize correct network topology
- Apply traffic control (if needed)

networks.py
- **Create server thread(s)**
  - *These threads send a command to the appropriate network node to start the server application matching their protocol. They will then wait for a shutdown signal from the test function.*

networks.py
- **Create tshark thread**
  - *This thread will start tshark, attached to the appropriate network node. The thread exits when tshark is killed by the test function.*
- **Create client thread**
  - *These threads send a command to the appropriate network node to start the client application matching their protocol. After data transmission, the output is parsed and processed. The thread exits after data is processed.*

TEST FUNCTION
- Initialize logger
- Configure qlog if QUIC

Initialize network
- Get client & server ID's
- Calculate RTT
- Add extra server parameters

Create & start server thread(s)

MAIN TEST LOOP
- Create test folder
- Add extra client parameters

Create & start tshark thread

Create & start client thread(s)
- When all threads are ready, they are released simultaneously
- After client threads are finished, kill tshark
- Convert .pcap files to plaintext
- If AIOQUIC, wait for qlog files
- If visualization flag, draw graphs
- Send INT-signal to server(s)
- Reformat qlogs to be more readable

Figure 9: Test framework overview.

A typical execution of a test scenario proceeds as follows:

- Starting in the main loop, a configuration file containing test parameters, implementation-specific parameters, and link properties is loaded and processed. This is followed by some setup and verification of the data in the configuration file, after which the test function is called.
- The test function will first initialize a global logger, followed by the creation of the Mininet network based on a pre-determined function detailing the network structure. Some IDs are gathered next, followed by the RTT measurement. Some server parameters have to be augmented here, as they require information about the network nodes and protocols.
- When the server parameters are set, a thread for each server is created, and subsequently started; these threads will send a command with the proper parameters to the appropriate Mininet node(s) to start the server implementation(s). These threads will then wait until they receive a specific signal, upon which the server and thread are terminated, and the time logged. The servers are thus part of the global context, instead of being reset for each test iteration - this enables the use of multiple clients either simultaneously, or consecutively.
- Following the server(s), the test function will enter the main testing loop, the length of which corresponds to the desired amount of iterations.
- For each loop, a test-specific folder and logger are created, client parameters established, and new threads created and started for both tshark and the client programs.
- The tshark thread, which functions similarly to the server thread, is started before the client thread(s), and runs until it is killed after all client threads of the current iteration are finished.
- The client thread(s) will execute their requests to their respective server, and upon receiving all data and a verification of the termination of the client program, will proceed to analyze and save the server response, and exit.
- After all tests are complete, the data acquired is processed, visualized, and saved. The program will then reformat qlog files to a human-readable format, organize the logs, do final cleanup, and finish.

While the network changes throughout the various tests in certain ways, a general schematic of it is presented in Figure 10.

5.3 Test scenarios

The following sections outline the various test scenarios and their relevant parameters. Each scenario includes tests with a variety of data sizes, as well as link parameters such as delay or packet loss. All tests are repeated ten times to account for variance, whether from the test environment itself or the implementation. All scenarios are limited to 1Mbit/s bandwidth to reduce the effects of operating system background

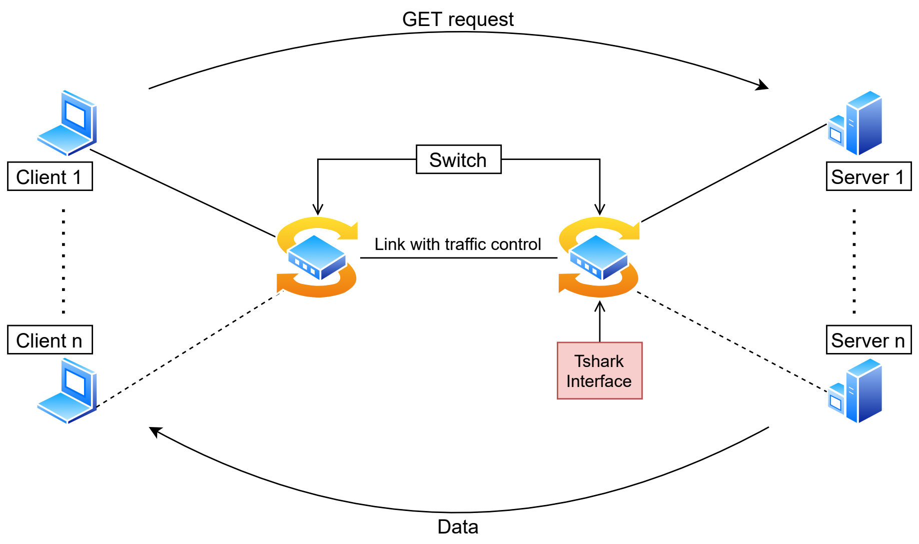


Figure 10: General test network structure.

processes on the tests. Based on comparisons to initial testing without bandwidth limitations, this eliminates a large number of spikes and inconsistancies within all test scenarios. This shouldn't be a problem for testing, as results in such capped conditions are also valuable, and the results should be scalable to higher speeds. Parameters used for all scenarios are presented in Table 1.

5.3.1 Scenario 1. Ideal network conditions

The ideal network test scenario is mainly used to establish baseline control values for all implementations - the only parameter applied to links in this scenario is a small amount of delay. All future tests can then be compared to these results in order to see if they react predictably to the changing network conditions. This scenario is also a good sanity check, ensuring that all implementations produce sensible values. Strange behaviour and results at this point will insinuate problems in either the QUIC/TCP implementation itself, or the testing framework. It is also a good place to see how the various programs perform when they do not have to worry about dropping packets.

5.3.2 Scenario 2. Nonideal network conditions

The nonideal network conditions scenario will focus on more sensible network conditions in terms of link parameters. Delays will be higher, and small amounts of packet loss are present - small amounts of other link characteristics may also be introduced if deemed necessary. This scenario will hopefully show the average performance of an implementation in different situations, were it to be deployed in a real network. This includes some tests with increasingly poor conditions; high delay, large amounts of packet loss and restricted buffer sizes. While modern networks can be quite fast and reliable in most places, the ability to perform well in poor conditions is still important.

5.3.3 Scenario 3. Multiple co-existing implementations

This scenario will be a repeat of various tests from previous scenarios, with the added parameter of having multiple implementations co-exist in the same network and use the same link to transfer data. The main goal of this type of testing is to figure out the fairness between the various protocols and implementations; whether they consume bandwidth equally, or if they will try to dominate others in order to perform better. While this is surely an important and interesting topic, it will not be focused on heavily - each QUIC implementation is run alongside the TCP implementation. This is also the only scenario that will use more than one server.

5.3.4 Scenario parameters

Table 1 shows the test parameters used in each test scenario.

	Scenario 1. Ideal	**Scenario 2.** Nonideal	**Scenario 3.** Parallel
Clients	1	1	1
Servers	1	1	2
Data size	10kB, 200kB, 1MB		
Bandwidth	1Mbit/s		
Delay	10ms	50ms, 200ms	50ms
Packet loss	0%	0%, 1%, 2%, 5%	0%
Buffer	0	0, 10	0

Table 1: Scenario parameters

6 Results

This section will cover the most relevant test results from all scenarios, their implications, as well as some potential explanations for them. In terms of results for any scenario, it should be noted that while TCP has been around for decades, and has gone through multiple iterations of improvements and extensions, the QUIC versions tested essentially represent the first iteration of the protocol (*technically* not even that, since these implementations are not v1 compliant, but they should be close enough). Additionally, in terms of data transfer, these scenarios are still quite basic, and do not take advantage of all of QUIC's capabilities, like transferring multiple resources at a time. That said, however, it is important for QUIC to perform well even in these simpler scenarios. For the sake of transparency it should be noted that while aioquic and ngtcp2 use HTTP/3 in their client/server implementations, AIOHTTP and LSQUIC rely on HTTP/1.1, which may have an impact on the results. Fortunately, in LSQUIC's case the most critical improvements of HTTP/3 over HTTP/1.1, like multiplexing and stream-based flow control, are delegated to QUIC, which should minimize the differences to a degree. As for the other core features of the more recent HTTP versions, content prioritization and server push are not relevant in the context of these test scenarios, and the data amounts transferred and general connection complexity are small enough where header compression should have minimal impact. Finally, while these implementations showcase how QUIC functions, all of them may not be designed or optimized specifically with performance in mind yet.

The 1MB data test results are visualized in the following sections using boxplots that show connection lengths, as well as tables detailing further relevant data. It should be noted that the boxplots are scaled to show all four implementations in relation to each other, and thus the *overall* scale of differences in connection times is more apparent from the tables (i.e. some time differences may seem larger than they actually are). As for interpreting the boxplots, the lines outside the boxes show the minimum and maximum values, whereas the red median line inside divides the box into the 25th percentile below and 75th percentile above. The outlined circles showcase outliers.

As for the tables, the top half of a table show more accurate connection times, a comparison of the average times in relation to TCP, followed by the handshake duration and a similar comparison. The bottom half of a table shows the average packet count in a connection, with a similar comparison to TCP as above, and the average ratio of packets sent by the client and server.

6.1 Scenario 1. Ideal network conditions

The ideal test scenario consists of no other link parameters than a 10ms delay, alongside the 1Mbit/s bandwidth limitation present in all scenarios. Figure 11 and Table 2 illustrate the results of the test when requesting a 1MB data file from the server. As with all scenarios, tests for 10kB and 200kB were also conducted.

As we can see from Figure 11, TCP is both the fastest, and most consistant implementation out of the four, which is not necessarily surprising, as mentioned in the introduction of this section. It has the fastest connection time overall, as well as the fastest maximum connection time. It is important to note, however, that the differences in connection times on average are less than four percent. While aioquic does manage the fastest connection time out of the QUIC implementations, the other two are more consistent overall. In terms of handshake times, we can see from Table 2 that LSQUIC and ngtcp2 manage to deliver a faster handshake time than TCP, ngtcp2 by a significant margin of over 50% no less, whereas aioquic has slightly longer handshakes *on average* than TCP. All QUIC implementations do manage handshakes with fewer packets than TCP, but the effects are not as pronounced with such low latency and longer connections. While aioquic and ngtcp2 handshake information is extracted straight from the qlog files, meaning that it should be quite accurate, LSQUIC information is acquired via the regular capture files in the abscence of qlog capabilities. This means that the LSQUIC handshake times do have a slight margin of error in comparison, as they include the extra time it takes to process and send the final packet to the tshark interface. The link between the server and tshark interface is not traffic controlled, however, so this time should be minimal. The TCP handshakes are calculated in a similar manner to LSQUIC.

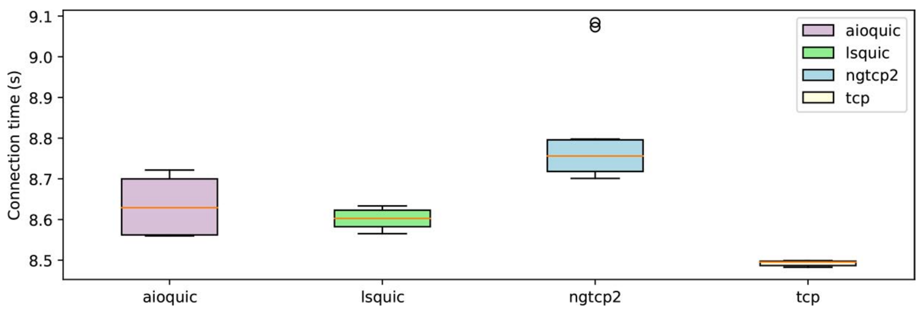


Figure 11: Scenario 1 results: 1MB (10ms, 0%)

Unlike the other QUIC implementations, aioquic handshake times have significant variance in length, going all the way from 40ms to 152ms. All of these spikes seem to be caused by the server sending a large datagram not present in the captures containing the shorter handshakes. While similarly sized packets are present in captures with either handshake lengths during the handshake phase, excluding the one previously mentioned, they are in different orders in either. Additionally, the capture files with the shorter handshakes cannot be fully decrypted with the TLS secret aioquic provides, whereas the files with the longer handshake can, which might indicate some irregularities with the handshake process - the qlog files for all connections do include the handshake completion event, however. Looking at the corresponding qlog files, it would appear that the connections with shorter handshakes process the same events faster as well: the first event of setting transport parameters happens at 0.074ms in one, whereas they are set at time 0.143ms in the other. Similar processing speeds for events occur in all files accordingly. This slowness is not contained to the first event, either, with time differences further exacerbating as the

	Protocol	Time (min, avg, max)			avg%	HS	HS%
	aioquic	8.560s	8.632s	8.722s	+1.6%	0.0933s	+9.6%
	LSQUIC	8.577s	8.616s	8.650s	+1.4%	0.0804s	-5.5%
	ngtcp2	8.701s	8.810s	9.084s	+3.7%	0.0343s	-59.7%
	TCP	8.483s	8.493s	8.499s	0%	0.0851s	0%
10ms, 0%	**Protocol**	**Packet count (avg)**			**avg%**	**C-S Ratio (avg)**	
	aioquic	1702			+50.0%	50%-50%	
	LSQUIC	1282			+13.0%	35%-65%	
	ngtcp2	1485			+30.8%	40%-60%	
	TCP	1135			0%	38%-62%	

Table 2: Scenario 1 results: 1MB (10ms, 0%)

log file proceeds - they seem to even out as the handshake completes. Interestingly, while there is no packet loss in the network, the log files for the connections with longer handshakes report packet loss of the same two packets during the handshake phase, a handshake and a 1-RTT packet from the server, which seems to be the reason for the large packet not present in the captures with shorter handshakes. The reasons for the differences in event processing speeds and the failing decryptions are unclear, although they seem linked in some manner. Were these irregularities resolved, aioquic would be on-par with ngtcp2 in handshake times.[33]

As for packet information, Table 2 shows that the QUIC implementations send more packets on average than TCP - this might be partly explained by the fact, that TCP packets are slightly larger than QUIC datagrams. From Wireshark we can see that the payload part of the TCP packets is 1448 bytes, whereas with QUIC the payload is between 1200 and 1250 bytes. Aioquic especially sends quite a lot more packets than the rest of the implementations, while still staying relatively on-par with connection times. Analyzing the capture files further, we can see that the extra packets are from the client, as each QUIC server sends roughly 900 packets per connection. Based on the packet and burst sizes, it would seem that these extra packets are acknowledgements, meaning that aioquic simply acknowledges server packets more frequently than other implementations. While this may prove detrimental in situations with congestion, it has minimal impact here. From further analysis of the aioquic capture files we can also see, that there is a smaller average time span between packets travelling in the network. While there are multiple potential reasons for this, like differences in congestion controllers or packet processing speeds, the most obvious explanation in this case is the fact that there are simply more

[33]Aioquic has received an update after these tests were performed, these problems *may* have been resolved.

packets in-flight at any given time, since the network conditions are identical in all cases, the data amounts are relatively small compared to the network capabilities, and the connection times are so similar.

Regarding the actual cumulative amount of data being transferred, excluding the 1MB file, TCP manages the least amount of *server* overhead at around 50.5kB, whereas aioquic and LSQUIC sit at around 69kB, and ngtcp2 at 82kB. Another interesting tidbit as far as packets are concerned, is that while TCP only has two different varieties of packets sizes throughout the entire connection excluding TLS packets during the handshake, the QUIC implementations have up to seven different packet sizes. These size differences are generally present in the smaller packets (<100bytes), which are most likely acknowledgements.

As for the tests with other data sizes, the 200kB test results are comparable to the 1MB ones in terms of relative connection speeds and packet statistics, with the only exception being that both aioquic and ngtcp2 manage significantly shorter handshake times compared to TCP and LSQUIC, resulting in aioquic being slightly faster than its peers. This is due to aioquic incurring only one of the previously mentioned handshake irregularities. The 10kB test case is the first one, where all QUIC implementations are, on average, slightly faster than TCP, both in terms of connection speed *and* handshakes. All QUIC implementations also manage equal or lower amount of packets sent, however they still send more actual data by similar margins than before. These results are explained by TCP deviating from its previous client/server packet ratio, and having the client send significantly more packets, relatively speaking, than before. This time aioquic has two handshake irregularities - these affect the connection times more here, as the connections are significantly shorter overall.

In terms of consecutive packets detected from one endpoint, the average is one or two packets, indicating that all endpoints prefer to exchange data in shorter sequences, however TCP and aioquic do exhibit a couple longer bursts in almost every connection, up to six packets. For ngtcp2, this maximum number is four, however bursts over three are exceedingly rare and only seem to occur in the 1MB test case. LSQUIC behaves similarly to TCP and aioquic in this regard, however there are a few burst of up to 60 packets present in the 1MB transfers, although these occur exclusively at the end of the transfer.

6.2 Scenario 2. Nonideal network conditions

The nonideal test scenario spans multiple different link parameters, however the basic idea from the ideal scenario remains unchanged. All of the different combinations mentioned in Table 1 are not covered in detail, since all combinations did not produce uniquely interesting results. For example, changing the delay to 50ms instead of 10ms with no other changes seems to do very little overall, the results are comparable to the ideal scenario with slightly longer connections. Additionally, the 1% and 2% packet loss test cases produced very similar results as well, so only the 2% case will be covered.

6.2.1 Delay: 50ms, Packet loss: 2%

As shown in Figure 12, a few interesting changes can be observed when some packet loss is introduced to the connections. Namely, aside from a few outliers, LSQUIC operates extremely consistently in this scenario, which would imply that the recovery and congestion control methods have been implemented well. Even though LSQUIC was the only implementation that got hit with packet loss during the handshake period out of all of the implementations, which resulted in one very slow handshake of over a second, it is still significantly faster than any other implementation on average by a significant margin. However, even if the connection with the long handshake was ignored, the average handshake time would still be 6.7% longer than TCP. The other QUIC implementations did still manage to beat TCP in this scenario, although their connection times were spread out more and their average times were closer to that of TCP. Ngtcp2 did manage a similar average handshake time difference compared to TCP as in the ideal scenario at around 60% less time overall, while aioquic beat TCP by 26.2% on average, with only three of the previously explained handshake anomalies. Once again, if these anomalies were not present at all, the average handshake times for aioquic would be almost identical to ngtcp2. As we can see from the data, introducing packet loss causes TCP a lot more problems than its QUIC counterparts; it no longer has the fastest connection time, and is in fact the slowest both in terms of average and maximum connection times. It also has the highest spread of connection times, with over four seconds between its fastest and slowest connection. It does still manage a slightly faster handshake on average than LSQUIC, however.

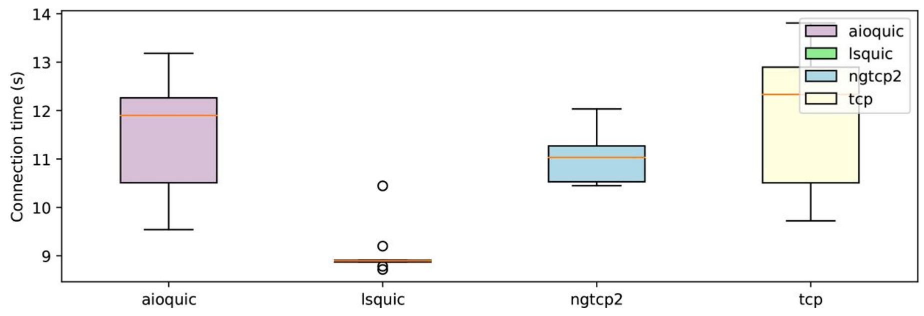


Figure 12: Scenario 2 results: 1MB (50ms, 2%)

In terms of packet counts, the values stay very similar to the ideal scenario, which isn't that surprising since the packet loss introduced is fairly minor. LSQUIC and ngtcp2 actually manage fewer packets in their connections than before - judging by the client-server packet ratio, this appears to be due to less acknowledgements than before. For aioquic, this ratio and the amount of packets remains similar to before. The server overhead values for LSQUIC and TCP are also roughly equivalent to before, however aioquic's increased by around 3kB, whereas ngtcp2's actually decresed by a similar amount. The implementations seem to be exhibiting slightly longer packet bursts than in the ideal test scenario, although the average is still in

	Protocol	**Time (min, avg, max)**			**avg%**	**HS**	**HS%**
	aioquic	9.541s	11.706s	13.235s	-1.2%	0.2185s	-26.2%
	LSQUIC	8.718s	9.063s	10.444s	-23.5%	0.3894s	+31.5%
	ngtcp2	10.448s	11.020s	12.033s	-7.0%	0.1176s	-60.3%
	TCP	9.721s	11.853s	13.808s	0%	0.2962s	0%
50ms, 2%	**Protocol**	**Packet count (avg)**			**avg%**	**C-S Ratio (avg)**	
	aioquic	1719			+49.3%	47%-53%	
	LSQUIC	1242			+7.9%	33%-67%	
	ngtcp2	1309			+13.7%	34%-66%	
	TCP	1151			0%	39%-61%	

Table 3: Scenario 2 results: 1MB (50ms, 2%)

the one-to-two packet range - this could be due to the missing packets causing a lack of acknowledgements on occasion. Interestingly LSQUIC has no more bursts of 60+ packets at the end of some connections either.

The 200kB test case showcases similar results, with a few exceptions. First, LSQUIC has slightly more spread in average connection times compared to the other implementations, whereas TCP has less. The median connection time for aioquic is also the fastest one in this case, and it even manages to barely edge out LSQUIC in average times by 6ms. While ngtcp2 outperforms TCP by 50ms or so, the median and average connection times are 100ms-200ms slower than its QUIC counterparts. TCP has the shortest maximum connection time, but loses to all QUIC implementations on average times. The 10kB results are similar to the 200kB ones, although this time the average and median connection times for aioquic are *significantly* shorter than any other implementation, beating the next fastest implementation by over 100ms, which is considerable since the average connection times are less than a second. The average times for aioquic are once again increased by the handshake anomalies, which do not seem to be caused by actual packet loss since they present in the exact same way as before. Based on the two test scenarios observed so far, it would appear that excluding the anomalies, aioquic has the most efficient connection establishment procedure, whereas LSQUIC shows the best performance in longer connections. Aioquic and ngtcp2 have similar handshake times, which are significantly faster than either LSQUIC or TCP, however ngtcp2 performs better than aioquic in longer connections. The QUIC implementations also seem to handle poor network conditions better than TCP.

6.2.2 Delay: 200ms, Packet loss: 0%

Figure 13 illustrates the results for the test case with a 200ms delay, without any packet loss. While certain elements already observed in the previous tests hold true, like increased packet amounts for QUIC and the client-server packet ratios, the spread of the results is quite different for each implementation. For one, every implementation other than ngtcp2 has almost no variance in connection times. Aioquic and LSQUIC have less than half a second of variation between minimum and maximum connection times, whereas TCP only has a difference of 16ms. It is possible, that as the delay is increased, the relevancy of the timing of the various background processes of the laptop the tests are run on decreases, thus tightening the spreads. Ngtcp2 also has fairly small variation overall, but a few outliers are present. These outliers have longer average times between packets, as well as more long bursts from the server compared to the shorter connections, while overall packet counts remain similar. Conversely the outliers with shorter connections than average have shorter average times between packets, with similar burst counts. The reasoning behind this behaviour is unknown.

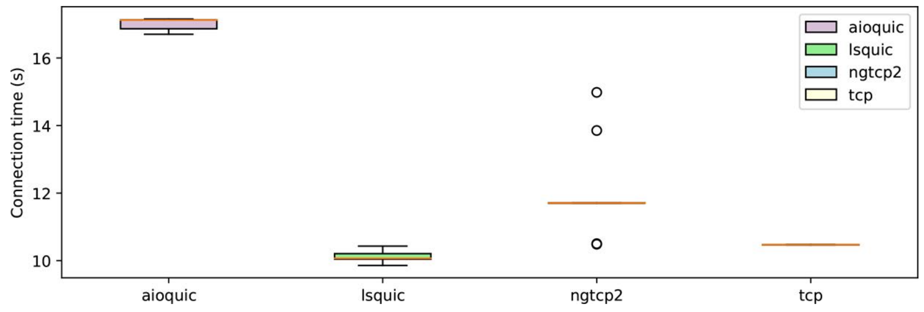


Figure 13: Scenario 2 results: 1MB (200ms, 0%)

In the 10ms and 50ms test cases *without* packet loss, all implementations were on-par with average connection times, but it would appear that increasing the delay further causes TCP to gain a significant advantage over both aioquic and ngtcp2, with the former having connections that last over 60% longer on average. From the 31.8% extra packets that aioquic sends on average compared to TCP, only around 17.5% come from the server - this is an 18.8% increase to what the TCP server sends on average. Looking at the payload sizes of both protocols' packets, alongside the amount of data that needs to be transferred, confirms that this amount is sensible. However, it also indicates that aioquic is processing these extra packets slower than TCP would, as the time difference in connections is over three times the server packet difference, and almost twice the overall packet difference. Since both implementations run on Python as well, it is unlikely that these processing differences are related to the programming language used. While aioquic does also have 6% more relative client traffic than TCP, the traffic itself shouldn't really affect the connection duration this heavily, if much at all. In ngtcp2's case, the *server* actually sends roughly 23% more packets on average than TCP, however the average connection lengths are only 14.7% longer than TCP. This indicates that ngtcp2 processes packets slightly faster than

	Protocol	Time (min, avg, max)			avg%	HS	HS%
	aioquic	16.702s	17.010s	17.153s	+62.5%	0.5203s	-50.3%
	LSQUIC	10.046s	10.200s	10.476s	-2.6%	0.8626s	-17.5%
	ngtcp2	10.486s	12.005s	14.982s	+14.7%	0.4250s	-59.4%
	TCP	10.462s	10.469s	10.478s	0%	1.0461s	0%
200ms, 0%	**Protocol**	**Packet count (avg)**			**avg%**	**C-S Ratio (avg)**	
	aioquic	1514			+31.8%	45%-55%	
	LSQUIC	1273			+10.8%	33%-67%	
	ngtcp2	1288			+12.1%	33%-67%	
	TCP	1149			0%	39%-61%	

Table 4: Scenario 2 results: 1MB (200ms, 0%)

TCP, but not fast enough to compensate for the extra packets. LSQUIC on the other hand actually manages to beat TCP's connection time, if only barely, continuing the trend of high performance in longer connections.

All QUIC implementations manage significantly faster handshakes than TCP in this test case. Unfortunately in connections of this length, the time gained from this is not enough for aioquic and ngtcp2 to beat TCP, though in LSQUIC's case it is enough. The server overhead values for aioquic, ngtcp2, and TCP remain similar to the previous test case, however interestingly for LSQUIC the overhead increases by almost 14kB, even though it is the fastest implementation. Burst sizes for ngtcp2, LSQUIC and TCP remain similar to before, favoring sending one or two packets at a time, whereas roughly half of the packets the aioquic server sends occur in bursts of three or more.

The 200kB results are quite similar to the 1MB ones, the only major difference being ngtcp2, where half of the connections experience the same behavior as the few longer outliers before. LSQUIC is also even faster than TCP in this case, with the average connection time being 595ms faster than TCP, likely due to the handshake having more effect in shorter connections. The 10kB results are also similar, however the relative difference in aioquic connection duration keeps coming down. Once again, this is most likely due to handshake times having more effect in shorter connections.

6.2.3 Delay: 200ms, Packet loss: 5%

The final test case covered in more detail has the worst network conditions tested; like before, the results are illustrated in Figure 14 and Table 5. The results of this test are essentially an amalgamation of the results from the 200ms test without packet loss, and the packet loss tests: the QUIC implementations deal with packet loss more efficiently than TCP is able to, however the longer delays seem to have a larger impact on aioquic than its counterparts. While aioquic is still slower than TCP, as it was in the 200ms test without packet loss, the margin has decreased by 41.7% due to the introduction of packet loss. Ngtcp2's average connection times are 14.6% faster than TCP's, even with a few connections longer than the TCP average. Finally, LSQUIC performs incredibly well in this situation, exhibiting connection times over half shorter than TCP, and three times faster than aioquic. It also stays within fairly tight margins in terms of connection time spread compared to the other implementations despite the relatively large amount of packet loss, at only 18.15 seconds between the fastest and slowest times. Additionally, LSQUIC has the fastest minimum connection time of all implementations by a considerable amount, being 147.6% faster than the next fastest implementation, ngtcp2.

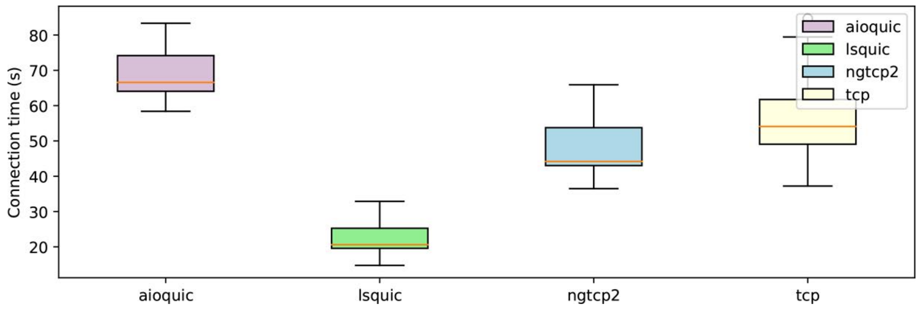


Figure 14: Scenario 2 results: 1MB (200ms, 5%)

	Protocol	**Time (min, avg, max)**			**avg%**	**HS**	**HS%**
200ms, 5%	aioquic	58.602s	68.606s	83.329s	+20.8%	0.7198s	-48.5%
	LSQUIC	14.739s	22.461s	32.887s	-60.5%	1.0004s	-28.4%
	ngtcp2	36.489s	48.504s	65.911s	-14.6%	0.5251s	-62.4%
	TCP	37.218s	56.805s	84.653s	0%	1.3974s	0%
	Protocol	**Packet count (avg)**			**avg%**	**C-S Ratio (avg)**	
	aioquic	1690			+43.3%	42%-58%	
	LSQUIC	1337			+13.4%	37%-63%	
	ngtcp2	1346			+14.2%	36%-64%	
	TCP	1179			0%	40%-60%	

Table 5: Scenario 2 results: 1MB (200ms, 5%)

Just like in the 200ms scenario without packet loss, all QUIC implementations have faster handshake times than TCP, with LSQUIC further increasing the relative difference to TCP. All implementations send more packets than they did previously as well, however their relative amounts compared to TCP stay nearly the same. LSQUIC and ngtcp2 do see a slight uptick in client traffic too, compared to previous tests, which could be the result of probing packets due to lost packets. The server overhead amounts remain similar to the 200ms test without packet loss, with the only exception being LSQUIC reducing said overhead by roughly 6.5kB.

The 200kB and 10kB tests behave very similarly to the ones in the 200ms test without packet loss, in the sense that the overall performance of the implementations remains the same in relation to others, but the shorter connections bring aioquic closer to the others in terms of performance, which is explained by the increasing relevance of the shorter handshakes in the shorter connections.

6.3 Scenario 3. Multiple co-existing implementations

The final test scenario is a repeat of previous test parameters, a 50ms delay and the ubiquituous 1Mbit/s bandwidth cap, however in this case each QUIC implementation is run in parallel with the TCP implementation. While this scenario was originally supposed to be much more comprehensive, for time reasons it is mostly used to observe whether the QUIC implementations consume network resources equally to TCP. The results of the 1MB transfer are shown as bar graphs in Figure 15. The top half of said figure has the QUIC results of both the parallel test, as well as the corresponding solo test with the same delay and packet loss parameters (50ms, 0%). The parallel results are represented first, and in a darker color scheme. As for the bottom half of the figure, the TCP results are shown. The colors of the TCP implementation match the QUIC implementations they were run parallel with, with the last yellow bars being from the solo test, as above.

From the graphs we can see, that while both aioquic and ngtcp2 experience close to twice or more the connection time than before, the TCP implementations running with them only have increases of roughly 30%-45%. Since each implementation performed almost equally in the solo tests time-wise, and since these performance hits are so different, it seems that TCP takes priority during these connections. That being said, the TCP implementations do experience some variance in connection times, so this priority does come with an element of randomness.

In the case of LSQUIC, this effect is the opposite: the average connection time for LSQUIC in parallel is up only 11.8%, whereas the corresponding TCP implementation endures an increase of 95.4%. LSQUIC operates quite efficiently here: while the slowdown TCP incurs here is of similar scale than aioquic and ngtcp2 previously, LSQUIC is only slowed down a third of what TCP did when operating with the other QUIC implementations. The results for the 10kB and the 200kB tests are comparable to the 1MB ones in all aspects, and all in all would indicate that LSQUIC includes some additional performance features, that allow it to consume network resources more aggressively than its counterparts, or TCP. While this is obviously good for LSQUIC, it does raise some fairness concerns, were it deployed into real networks. Conversely, aioquic and ngtcp2 could use a slightly more dominant approach to data transfer. It is noteworthy, however, that this is only one test scenario with a relatively small sample size, so further testing on this aspect of the implementations would be beneficial.

As for the other statistics of the implementations, most remain similar. When running parallel with aioquic, the TCP implementation's packet counts and ratios remain almost identical to the solo tests, whereas with ngtcp2 and LSQUIC a slight increase in client traffic can be seen. Packet counts for servers remain similar in all cases, so there are minimal changes to server overhead, if any. Handshake information was not measured as exhaustively as before, but some minor increases for TCP handshakes can be seen with aioquic and ngtcp2, while running parallel with LSQUIC shows handshakes three to four times longer than running solo. Handshakes for all QUIC implementations remain within small margins of solo tests.

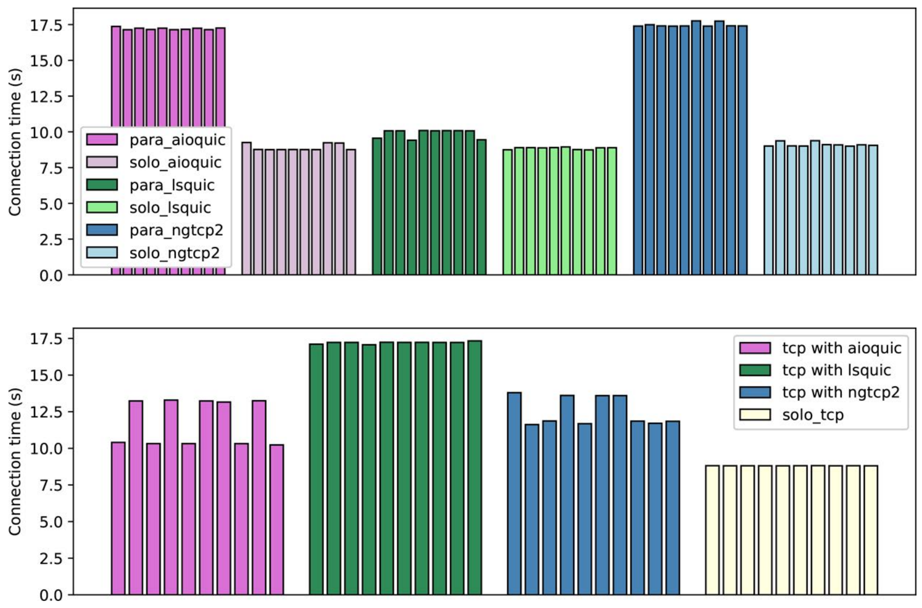


Figure 15: Scenario 3 results: 1MB (50ms, 0%)

7 Conclusions

This final section will briefly summarize the key points of the thesis, as well as cover some related topics outside the scope of it.

7.1 Research summary

In this thesis a performance analysis of different QUIC implementations was carried over using a self-developed Python test framework, taking advantage of Mininet for network emulation purposes. The test framework enables changing various implementation parameters efficiently, using custom network topologies with a dynamic amount of clients and servers, automatic testing, logging, data collection and processing with multiple test iterations at a time, and even parallel operation of up to two implementations. Three different QUIC implementations were tested, along a TCP implementation that acted as a control for the results. The tests included downloading files of varying sizes in increasingly poor network conditions. Data was collected via tshark network captures, as well as various logging features provided by both the test framework itself, as well as the implementations. This data was further analyzed and visualized afterwards with the help of multiple scripts; the most unique and interesting of these results are shown in tables and figures in Section 6.

The test scenarios are quite simplistic, and actually favor TCP as some of the more performance-critical features of QUIC are not utilized fully, like downloading multiple resources concurrently, or being able to migrate connections efficiently. QUIC also places heavy emphasis on handshake optimization, which is less relevant with longer connection times. Despite all this, the results of the tests show that all of the QUIC implementations are able to match, or even beat TCP connection times in certain scenarios. They are also all able to deliver shorter handshakes than TCP, excluding some anomalies, in most scenarios. One out of the three tested QUIC implementations, LSQUIC, actually delivers faster connection times to TCP in all but the ideal connection scenario, and performs very consistently overall. LSQUIC does feature the longest handshake out of the three QUIC implementations tested, however, even slightly exceeding that of TCP in a select few test cases. The results for ngtcp2 include a bit more variance, with slightly improved performance compared to TCP in some scenarios, and slightly worse in others. A significant improvement in handshake duration stays constant throughout, however. Finally, aioquic performs quite well with lower delays and short connections, even surpassing its counterparts in some scenarios, but overall falls short with longer connections and higher delays. While it does generally deliver a shorter handshake than TCP, comparable to ngtcp2, anomalies were present in some connections, affecting the averages negatively.

In terms of packets, all QUIC implementations send a higher amount of packets than TCP. One reason for this is smaller payload sizes compared to TCP, as minimum datagram sizes are used since the test network does not feature any unit discovery protocols. LSQUIC and ngtcp2 generally send similar packet amounts, and exhibit similar client-server packet ratios, whereas aioquic sends significantly more packets in the form of acknowledgements. Server overhead for all QUIC implementations is higher than it is for TCP. Running the QUIC implementations in parallel with TCP shows, that LSQUIC uses network resources significantly more aggressively than TCP, whereas aioquic and ngtcp2 let TCP take priority.

7.2 Limitations of the study

The limitations of the thesis were mostly time-related, however the fact that QUIC was under active development throughout the majority of the thesis writing process, and the fact that it is still not officially a fully finalized RFC did also introduce some complications - mostly in terms of testing. As for the theoretical portions relating to QUIC, the main sources of information that were both reliable and up-to-date were the RFC drafts, which did go through some changes throughout the writing process, and don't always translate well to general human-readable protocol descriptions. Similarly, since the protocol has been in a state of flux and only mostly stabilized functionally at the 29th draft, most of the existing external tests and research are outdated to some degree as well. This also affected the practical portions of the thesis, as an unfortunately large portion of the QUIC implementations did not uphold a very rapid update schedule in terms of new draft versions. While this is understandable, as some of them are essentially one-person projects, it did impact the decision process for Section 4.2, alongside a lack of certain features. While all tested implementations are up-to-date now, due to a sudden significant time period with no updates for aioquic after draft 29, all tests were performed with versions corresponding to said draft. While this shouldn't change protocol behaviour all that much, if at all, the implementations may have received unrelated fixes and improvements in later versions.

The timeframe introduced limitations on the feasible number of implementations to test, as well as the amount and complexity of the test scenarios. It also restricted the implementation selection process, as there was no time to develop custom test applications for every implementation. The testing framework was also developed from scratch, which means that certain originally intended features were never realized, and some improvement ideas born from performing the first tests had to be ignored. The amount of data acquired from even the rather simple set of test scenarios presented is also already quite substantial.

7.3 Suggestions for further research

As the specifications are finalized, and more of the existing implementations catch up to them, it would obviously be good to do similar testing with a larger set of implementations, as well as more complicated network topologies. This should include more mixing of the implementations within one network, both in terms of more end-poins of the same implementation, as well as interoperating implementations. Other scenarios that had to be dropped for time reasons include testing the implementations in longer stretches where data transmissions come in varying bursts, downloading multiple files concurrently, comparing situations where multiple clients are started simultaneosly against when they are started randomly, and comparing the effects of switching networks in TCP and QUIC. Additionally, these tests would surely provide more interesting results when executed in a proper laboratory environment, or an actual network, as opposed to a single regular laptop. A simple repeat of the tests done here after some time would be interesting as well, since the optimization work for these implementations is sure to begin in earnest after QUIC reaches a stable RFC.

One of the original ideas for a research question was whether the integration of TLS to the protocol provided any meaningful performance increases as opposed to how it functions in TCP, however this was abandoned since disabling TLS on the QUIC implementations is quite tricky. However, if the implementations could be set to run either without TLS, or just without having to actually do the encryption work, it might be interesting to compare if the relative performance hits are equal to that of TLS running on top of TCP.

References

[1] J. Kurose and K. Ross, *Computer Networking: A Top-Down Approach.* Pearson, 7th ed., 2017.

[2] "Transmission Control Protocol." RFC 793, Sept. 1981.

[3] "User Datagram Protocol." RFC 768, Aug. 1980.

[4] S. Kumar and S. Rai, "Survey on transport layer protocols: Tcp & udp," *International Journal of Computer Applications*, vol. 46, no. 7, pp. 20–25, 2012.

[5] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante, "De-ossifying the internet transport layer: A survey and future perspectives," *IEEE Communications Surveys Tutorials*, vol. 19, no. 1, pp. 619–639, 2017.

[6] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," Internet-Draft draft-ietf-quic-transport-34, Internet Engineering Task Force, Jan. 2021. Work in Progress.

[7] B. Turkovic and F. Kuipers, "P4air: Increasing fairness among competing congestion control algorithms," in *2020 IEEE 28th International Conference on Network Protocols (ICNP)*, pp. 1–12, 2020.

[8] R. Denda, A. Banchs, and W. Effelsberg, "The fairness challenge in computer networks," in *Quality of Future Internet Services* (J. Crowcroft, J. Roberts, and M. I. Smirnov, eds.), (Berlin, Heidelberg), pp. 208–220, Springer Berlin Heidelberg, 2000.

[9] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida, "The NewReno Modification to TCP's Fast Recovery Algorithm." RFC 6582, Apr. 2012.

[10] S. Floyd, J. Mahdavi, M. Mathis, and D. A. Romanow, "TCP Selective Acknowledgment Options." RFC 2018, Oct. 1996.

[11] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, "CUBIC for Fast Long-Distance Networks." RFC 8312, Feb. 2018.

[12] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "Bbr: Congestion-based congestion control," *Queue*, vol. 14, no. 5, pp. 20–53, 2016.

[13] A. Margara and T. Rabl, *Definition of Data Streams*, pp. 648–652. Cham: Springer International Publishing, 2019.

[14] B. Ford, "Structured streams: A new transport abstraction," *SIGCOMM Comput. Commun. Rev.*, vol. 37, p. 361–372, Aug. 2007.

[15] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)." RFC 7540, May 2015.

[16] R. R. Stewart, "Stream Control Transmission Protocol." RFC 4960, Sept. 2007.

[17] G. Davis, "2020: Life with 50 billion connected devices," in *2018 IEEE International Conference on Consumer Electronics (ICCE)*, pp. 1–1, 2018.

[18] J. M. Kizza, *Guide to Computer Network Security*. Springer, Cham, 5th ed., 2020.

[19] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3." RFC 8446, Aug. 2018.

[20] S. Bellovin, "Guidelines for Specifying the Use of IPsec Version 2." RFC 5406, Feb. 2009.

[21] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2." RFC 6347, Jan. 2012.

[22] H. Nielsen, J. Mogul, L. M. Masinter, R. T. Fielding, J. Gettys, P. J. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol – HTTP/1.1." RFC 2616, June 1999.

[23] T. Berners-Lee, R. T. Fielding, and L. M. Masinter, "Uniform Resource Identifier (URI): Generic Syntax." RFC 3986, Jan. 2005.

[24] M. Belshe and R. Peon, "SPDY Protocol," Internet-Draft draft-mbelshe-httpbis-spdy-00, Internet Engineering Task Force, Feb. 2012. Work in Progress.

[25] R. Peon and H. Ruellan, "HPACK: Header Compression for HTTP/2." RFC 7541, May 2015.

[26] C. B. Krasic, M. Bishop, and A. Frindell, "QPACK: Header Compression for HTTP/3," Internet-Draft draft-ietf-quic-qpack-21, Internet Engineering Task Force, Feb. 2021. Work in Progress.

[27] M. Bishop, "Hypertext Transfer Protocol Version 3 (HTTP/3)," Internet-Draft draft-ietf-quic-http-34, Internet Engineering Task Force, Feb. 2021. Work in Progress.

[28] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control," Internet-Draft draft-ietf-quic-recovery-34, Internet Engineering Task Force, Jan. 2021. Work in Progress.

[29] M. Thomson and S. Turner, "Using TLS to Secure QUIC," Internet-Draft draft-ietf-quic-tls-34, Internet Engineering Task Force, Jan. 2021. Work in Progress.

[30] M. Tüxen, R. R. Stewart, R. Jesup, and S. Loreto, "Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets." RFC 8261, Nov. 2017.

[31] J. Roskind, “Quic: Design document and specification rationale.” https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit, 2013. Accessed: 1-12-2020.

[32] C. Fernandes, “Microsoft to add support for google’s quic fast internet protocol in windows 10 redstone 5.” https://www.windowslatest.com/2018/04/03/microsoft-to-add-support-for-googles-quic-fast-internet-protocol-in-windows-10-redstone-5/, 2018. Accessed: 1-12-2020.

[33] B. Van Damme, “How to enable http3 in chrome / firefox / safari.” https://www.bram.us/2020/04/08/how-to-enable-http3-in-chrome-firefox-safari/, 2020. Accessed: 1-12-2020.

[34] I. S. D. Schinazi, F. Yang, “Chrome is deploying http/3 and ietf quic.” https://blog.chromium.org/2020/10/chrome-is-deploying-http3-and-ietf-quic.html, 2020. Accessed: 1-12-2020.

[35] M. Bishop, “Hypertext Transfer Protocol Version 3 (HTTP/3),” Internet-Draft draft-ietf-quic-http-32, Internet Engineering Task Force, Oct. 2020. Work in Progress.

[36] D. McGrew, “An Interface and Algorithms for Authenticated Encryption.” RFC 5116, Jan. 2008.

[37] L. Eggert, G. Fairhurst, and G. Shepherd, “UDP Usage Guidelines.” RFC 8085, Mar. 2017.

[38] E. Blanton, D. V. Paxson, and M. Allman, “TCP Congestion Control.” RFC 5681, Sept. 2009.

[39] E. Blanton, M. Allman, L. Wang, I. Järvinen, M. Kojo, and Y. Nishida, “A Conservative Loss Recovery Algorithm Based on Selective Acknowledgment (SACK) for TCP.” RFC 6675, Aug. 2012.

[40] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” Internet-Draft draft-ietf-quic-transport-29, Internet Engineering Task Force, Sept. 2020. Work in Progress.

[41] Apple, “About objective-c.” https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, 2014. Accessed: 23-2-2021.

[42] R. Marx, “Main logging schema for qlog,” Internet-Draft draft-marx-qlog-main-schema-02, Internet Engineering Task Force, Nov. 2020. Work in Progress.