
# Quicker:
# On the design and implementation of the QUIC protocol

Kevin Pittevils

Promotor: Prof. dr. Peter Quax
Co-promotor: Prof. dr. Wim Lamotte
Mentor: Mr. Robin Marx

Thesis proposed to achieve the degree of master in computer science.

Academic Year 2017-2018

[This page intentionally left blank]

# Abstract

In the last decade, issues were discovered with the use of HTTP/1.1. To solve this, Google developed a new protocol, called SPDY, that was later used to create and standardize HTTP/2. However, while using and optimizing SPDY, Google found other complications that it could not correct in SPDY itself. This was due the fact that SPDY (and HTTP/2) were running on top of TCP. When using TCP as a transport protocol, middleboxes tend to interfere with the connection, the connection setup is rather slow and most of all, HOL blocking occurs when there is packet loss. As a reaction to this, Google developed Google QUIC on top of UDP, to replace TCP as a transport protocol. Because of the success of Google QUIC, the IETF decided to standardize the QUIC protocol.

In this thesis, we explore the obstacles of HTTP and TCP and discuss how QUIC aims to solve these. Next we look into the various more complex features of QUIC. As part of this thesis, we have developed our own IETF QUIC server and client, called Quicker, which is up to date with draft 12 of the in-progress QUIC RFC. Consequently, when looking into the features of QUIC, we also give some insight in how we have implemented these aspects in Quicker.

Afterwards, we test the interoperability of our implementation with other IETF QUIC based libraries to verify that our server can correctly communicate with other clients/servers and vice versa. We also modify our client to test the compliance of the standard for these same libraries, where we notice clearly that not all aspects are always implemented correctly. Lastly, we measure the performance of Quicker in comparison to other implementations, where we noticed that our implementation, together with some others, currently cannot handle a lossy network very well. Only one library was able to handle the lossy network and manage 200 simultaneous connections at a time.

# Acknowledgements

This thesis was only possible because of the help and support of others. First of all, I would like to thank Robin Marx for his valuable advice throughout the development of Quicker. I would also like to express my gratitude towards him for all the suggestions, guidance and proofreading of this text. Additionally, I want to thank Prof. dr. Peter Quax and Prof. dr. Wim Lamotte for giving me the opportunity to write this thesis.

Furthermore, I appreciate all the help of the QUIC development group, especially Nick Banks for providing useful input and help during the development of Quicker and Tatsuhiro Tsujikawa for making changes available of OpenSSL which were necessary to implement QUIC.

Lastly, I am also grateful towards my girlfriend, parents, brothers, sisters and friends for their support during my studies and by giving me the motivation to finish this thesis.

# Contents

# Chapter 1

# Introduction

Since the establishment of the Transmission Control Protocol (TCP) in 1981 and the Hypertext Transfer Protocol (HTTP) in 1989, the internet has changed. During the years that these protocols were used, issues regarding their functioning in real networks (e.g., with regards to security, performance and evolvability) were discovered.

To biggest problem of HTTP/1.0 was the fact that a new TCP connection has to be established for each resource that had to be retrieved by the client. To mitigate this and other issues, HTTP/1.1 added persistent connections, which solved most practical performance issues for a couple of years. However, issues were also discovered for HTTP/1.1. An example of these issues, is head-of-line blocking (HOL blocking), which resulted in a workaround of multiple TCP connections being used to request multiple files of a webpage simultaneously. By doing this, more server and network resources need to be used to serve a single client. As one of the larger companies that make profit with the use of the internet, Google started the development of a new protocol, called SPDY, intended to solve the resource usage issues of HTTP/1.1. With SPDY, they could test their assumptions and optimize the protocol for their own use. HTTP/2 was later based on the successful ideas of SPDY. HTTP/2 solves various issues of HTTP/1.1 and makes better use of the underlying TCP connection instead of opening multiple connections, as was done in HTTP/1.1.

However, while Google was using and optimizing SPDY, they also discovered issues in both HTTP/1.1 and SPDY, mostly due to the underlying TCP connection and TLS connection setup. As was the case for HTTP/1.1, TCP also suffered from HOL blocking. Another complication was the cost of establishing a new secure connection, taking three round trips. Lastly, there is also the middlebox interference. Middleboxes tend to manipulate TCP packets to try to optimize the connection, which could lead to unpredictable behaviour. Bothered by these issues, Google started with the development of Quick UDP Internet Connections (QUIC), which is implemented on top of UDP rather than TCP and is now referred to as Google QUIC.

With the use of QUIC, the main issues that were bothering TCP, were solved. For example, with regards to HOL blocking, TCP looks at every packet on a single stream of data. Conversely, QUIC uses the concept of streams, originally used in protocols such as HTTP/2 and SCTP. When each resource is assigned an individual conceptual stream, it is possible for the transport-layer to know that, when a packet is lost, the subsequent packets could still be used if they contain data of another resource that was not in the lost packet. Other examples of issues that QUIC solves is the slow connection establishment of TLS, by making it possible to start sending encrypted data, even prior to the completion of the QUIC handshake, and the connection migration, which makes it possible for a device to change the network they are on, without the connection being dropped. To avoid deployment issues with middleboxes and to be easily changeable, QUIC is implemented on top of UDP.

With the success of Google QUIC, the Internet Engineering Task Force (IETF), decided to standardize this protocol so that the rest of the world could also benefit from this excellent work. This version of QUIC is referred to as IETF QUIC. When QUIC is used in this text, it refers to the IETF QUIC

version.

In this thesis, we will focus on implementing QUIC, while it is being standardized. The fact that the official specification is still in flux leads to frequent updates to the text and any implementations that attempt to follow along. This is useful, as the design of a protocol should be informed by real implementations, as they frequently uncover unexpected real-world issues that theoretical processes would not discover. As such, various implementers have concurrently started similar work to our own.
Our main research question is: "How complex is it to develop a new protocol based on an incomplete specification?". Related questions are: "How compliant are these types of implementations throughout their evolution?", "How much emphasis is put on performance from the get-go?" and "How is the interplay between implementations and design of QUIC?". These questions are going to be answered with first of all, a feasibility study, where we are going to implement QUIC ourselves to experience the complexity of implementing a changing protocol. The feasibility study is followed by a process evaluation, where our implementation of QUIC is tested against other QUIC implementations. By using our own codebase, we can evaluate the compliance of all efforts with the current QUIC specification draft and compare the performance of other implementations to our own.

We will begin by providing some background information about HTTP, TCP, UDP and the benefits of QUIC in section 2. This is followed by section 3, where we are going to look more closely at QUIC and the implementation decisions we made while implementing QUIC ourselves. In section 4, we are going to evaluate and compare our implementation to those of others. Finally, in section 5, we will form our conclusion and answer our research questions.

# Chapter 2

# Background

## 2.1 HTTP

### 2.1.1 Definition

The Hypertext Transfer Protocol, or HTTP, is an application-layer protocol used for data communication on the World Wide Web. It was created by Tim-Berners-Lee in 1989 during his time at CERN, where he also built the first web browser.

The first version of HTTP is HTTP/0.9 [26] and is a simple client-server communication model that uses a reliable transport protocol, namely the Transmission Control Protocol (TCP). The client requests a resource with ASCII characters, terminated by a carriage return and a line feed (CR LF). A request starts with the word GET, followed by the requesting resource without HTTP, host name, and port. The request also needs to be idempotent. Idempotent describes an operation that will result in the same state when it is executed multiple times. This is important in HTTP because a server needs to be stateless. This means that the server does not have to keep the state after the request has been processed. A response consists of a resource in HTML (Hypertext Markup Language).

### 2.1.2 HTTP/1.0

HTTP [35] was being used a lot after it was created, however the main problem with HTTP/0.9 was its various limitations. It can only serve HTML resources and there is no way to provide any metadata about the request, response, resource or either endpoint. Webdevelopers wanted a way to exchange files other than HTML files. There is also no way the browser can tell if the resource is obtained successfully or not. If a particular resource was not available, the webdeveloper had to sent a custom HTML file with an error message, but the browser itself could not detect this.

In HTTP/1.0 [27], these limitations were taken into account. Firstly, more headers were added for more functionality (e.g., content encoding, character set support, content negotiation). With these headers, it is possible for either endpoint to provide metadata about the request, response, resource or of the endpoint itself. And as a result, it is also possible to provide other resources then HTML.

Secondly, more HTTP methods were added. Instead of only the GET method in the first line of a request, POST, HEAD, PUT, DELETE, LINK, and UNLINK were also available. However, only GET, POST and HEAD were consistently implemented by most HTTP implementations. These methods make it clear what the request is intended to do (e.g., DELETE method is used to delete a particular resource).

Lastly, status codes were added to understand and satisfy the request. This made it easier for browsers to understand the response of the server (e.g., When a particular resource does not exist on the server, it sends a response with status code 404 to notify that the resource is not present on the server or when a resource has been permanently moved, the server sends a response with status code

301).

### 2.1.3 HTTP/1.0 limitations

Although HTTP/1.0 was formed by various experiments from different sources, it still has some critical issues.
One of these issues is that a new connection has to be created for every single resource that is requested as shown in Figure 2.1. At the start of HTTP, this was not a big problem, because most web pages only consisted of HTML files. However, when files other than HTML became possible to send, this was becoming a bottleneck of HTTP. A TCP connection needs one RTT (Round-Trip Time) for the connection to set up. When TLS (Transport Layer Security) is enabled, an additional two RTT are needed. This means before HTTP tells the server that it needs a particular resource, already three RTT are used to start communicating. To explain this in detail, the next example is given:

Let us assume that the index page of www.example.com consists of one HTML file, two CSS (Cascading Style Sheet) files and one JS (JavaScript) file.

1. The client starts by opening a TCP connection to the webserver of www.example.com. The server starts allocating resources for this connection. Additionally, if TLS is enabled, the client and server start negotiating secrets for the secure connection.
2. Next, the client requests index.html from the webserver. The webserver responses with a HTTP response with the resource in its body.
3. After the resource has been received by the client, the client starts closing down the TCP connection and the server frees the allocated resources for the connection.
4. The client starts to process the index.html file and notices that it needs additional resources for the webpage to load.
5. The client starts requesting the additional resources in the same way it requested index.html.

Furthermore, because a new connection has to be created for every single resource that is being requested, the connection would never use the entire available bandwidth. This is because TCP starts the connection in the slow-start stage. After every RTT, TCP doubles the amount of packets that it transmits. This is to gradually adapt to the available bandwidth of the connection. Although this manner is effective for long-lived connections, for short-lived connections, like these of HTTP, this is very ineffective. This is explained in detail in Section 2.2.4

This was also the conclusion of the analysis of Spero [54]. In the performed tests, HTTP had to spent more time waiting than it actually transfers data to the receiver. The total transaction time to request a file with a new connection was 530ms, while an already opened connection would only be 120ms.

### 2.1.4 HTTP/1.1

In 1997, the first HTTP standard was released in RFC 2068 as HTTP/1.1 by the IETF (Internet Engineering Task Force). Shortly after the release, in 1999, HTTP/1.1 got revised in RFC 2616. This new version added some optimizations [43] to the existing HTTP/1.0.

One of these optimizations, is the persistent connection. A persistent connection, is a TCP connection that is kept alive until either endpoint decides to close the connection or it is unused for an amount of time. Unlike in HTTP/1.0, the connection is not closed when the resource that has been requested is received by the client. This connection can be reused to request another resource. This optimization solves the problem that a TCP connection would never reach its full potential and removes the three RTT startup cost for new connections.

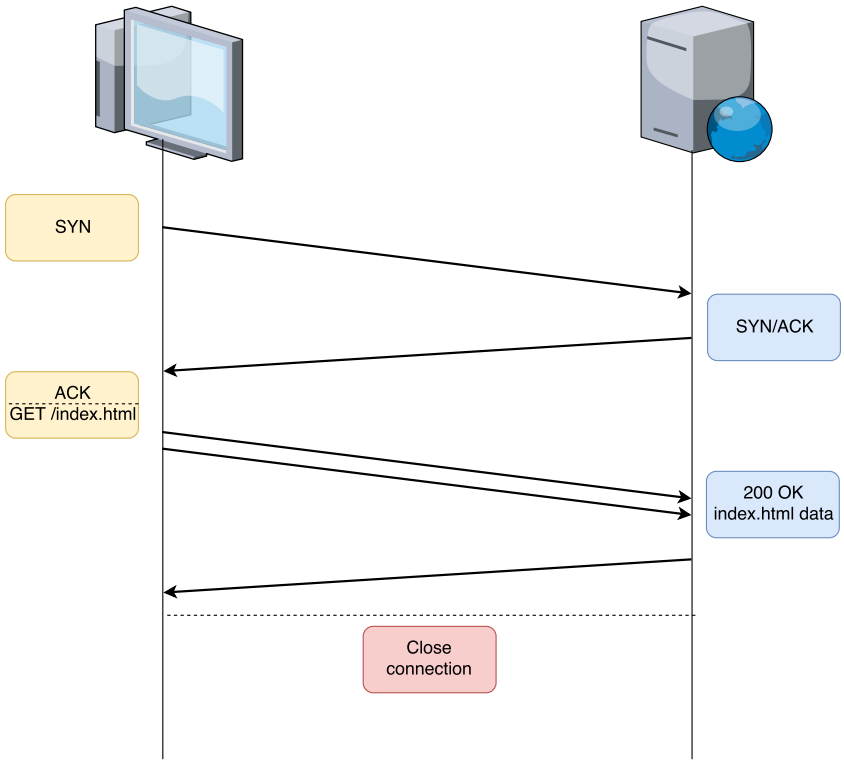



*Figure 2.1: TCP connection that gets closed after the resource is received by the client.*

To optimize this even further, the HTTP-WG (HTTP Working Group) introduced pipelining. Normally, when a resource is requested by the client, the client needs to wait until this resource has been received before requesting another resource as shown in Figure 2.2. With pipelining, the client does not have to wait for the resource to be received. The client can request multiple resources at ones. The server would then process them and send the responses with the resources back to the client. The only requirement is that it the server needs to process these request as a FIFO (First-in First-out) sequence.
Barford et al. [24] evaluated HTTP/1.1 by comparing it with HTTP/1.0. They found out that HTTP/1.1 performance is better than HTTP/1.0 under any network performance circumstances. When the CPU is the bottleneck, HTTP/1.1 performed slightly better than HTTP/1.0, which points out that HTTP/1.1 does not add or reduce CPU processing. However, HTTP/1.0 did perform slightly better when memory is the bottleneck. This is due the fact that the connection that need to be kept open consumes memory, even when they are idle for a moment.

### 2.1.5 HTTP/1.1 limitations

As previously stated, when the client sends multiple requests, the server has to process them in the same order as they are sent. E.g., if the client requests index.html first and main.css as the second resource and the server would send main.css first, the client would think that this is index.html. This behavior is caused by the inability of HTTP/1.1 has no way to correlate a particular response to its request.

To illustrate this problem even further, it is very similar to a call center with only one employee. If five people would call this particular call center and there is only one person to pick up the phone, the first caller would be helped first by the employee. The employee of the call center has no idea how long it would take to maintain this first caller, and the other four have to wait until this caller is helped. If this caller has a lot of problems, the other four have to wait a significant amount of time to get any help from the employee, even though their problem could be fixed in a matter of seconds.

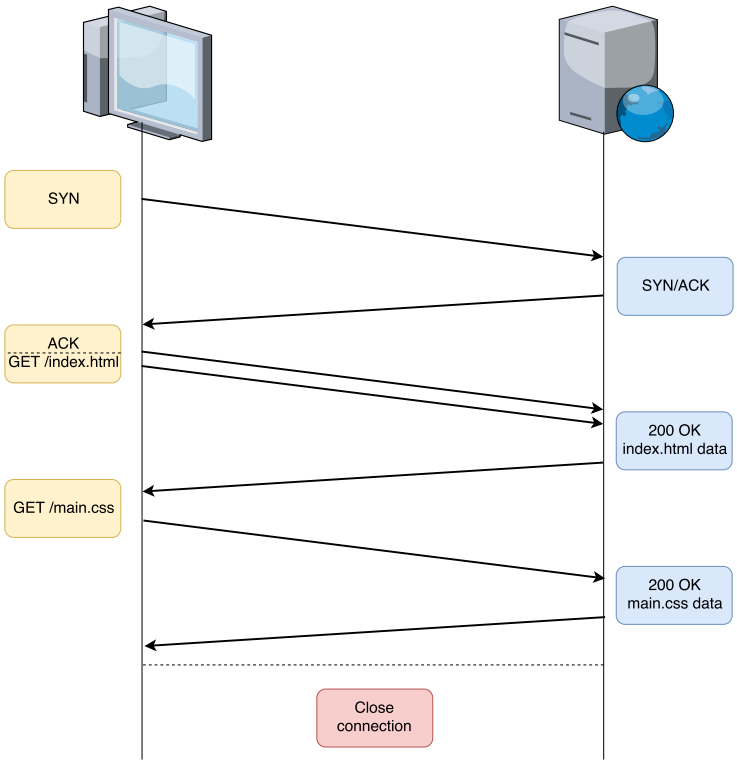



*Figure 2.2: TCP connection that gets reused by the client.*

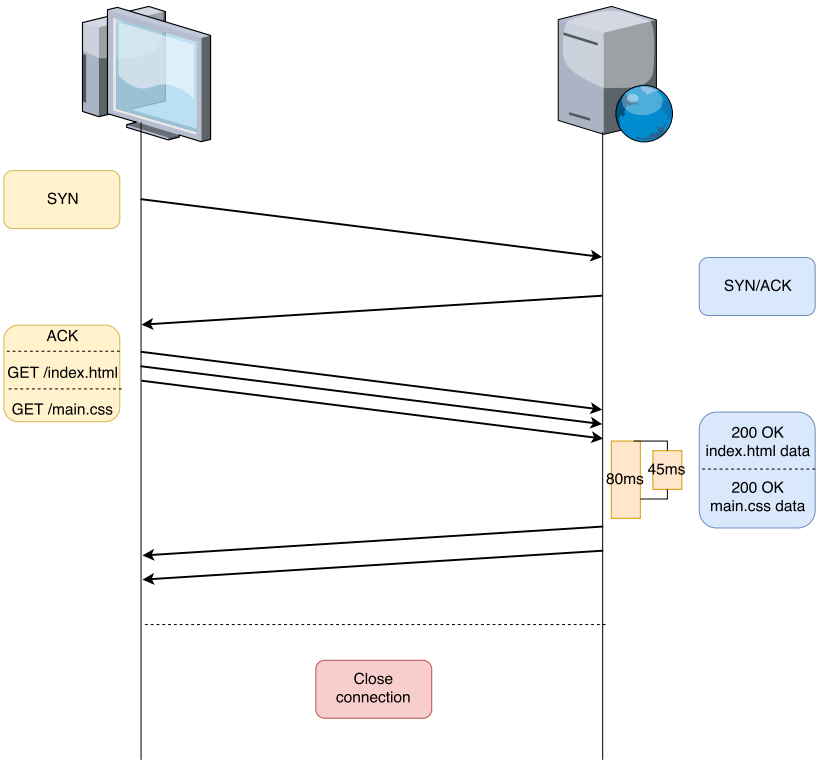



*Figure 2.3: TCP connection that gets reused by the client, with pipelining. Index.html takes the server 80ms to process and main.css 45ms. However, the server needs to wait to send main.css until index.html has been sent.*

In the same way, the client has no idea how long each resource would take to process. This problem is called HOL blocking (Head-of-line blocking). When the client starts requesting resources and the first one is the biggest file that needs the most processing, the client would not get any other resources until the first one is received. This is shown in Figure 2.3. In reality, pipelining is rarely used or even implemented. Clients wait until the first resource is received before requesting another resource over the TCP connection. For this reason, multiple TCP connections are used to request data.

With the example of the call center again. When there are two employees and nine people are calling the call center, one employee could be stuck with the first person calling, while the second employee has done the other eight at the same time. Similar to HTTP, one resource could spend more time than nine others. When a client needs to request nine resources and uses two TCP connections, it will request the first two resources with the TCP connections. When one of these requests are done, the client is going to request the next resource until all resources are obtained. If the first resource is very large, the other eight resources would be requested using the second TCP connection, thus avoiding the HOL blocking of the first resource. The client could request the headers with the HEAD operation first to determine how big the resource is. However, this also takes time in which the real resource could have been requested.

Even though multiple TCP connections solve the problem of HOL blocking, it introduces several other problems. When a client is using two applications that needs to request several resources and the first application is using one TCP connection, while the second is using six TCP connections, the second application gets an unfair share of the bandwidth. This is because HTTP uses TCP to transfer data from the server to the client. When multiple TCP connections are used, TCP's congestion control makes sure that the available bandwidth is divided equally by the amount of TCP connections that are being used. This is to provide a fairness between multiple TCP connections. If the second application uses more TCP connections than the first application, it can use more bandwidth available than the other application.

Another problem of HTTP arises from the use of HTTP headers. HTTP headers are often used to exchange information about the state of the endpoint and more information about the resource that is requested (e.g., information of the type of the data with the Content-type header). Even though this data is useful for both client and server and is often used to have a stateless server, this header information can become larger than the MSS (Maximum segment size) of TCP, which is the largest amount of data that an endpoint can receive in a single TCP packet and is provided during the handshake of TCP, which is explained in detail in section 2.2.2. This results in the fact that it needs more than one packet to send only the headers of a request or response. Typically most of the headers are sent with every request and contain the same information (e.g., Information about the server, large cookies). With this, a simple request could take more space than the MSS of TCP and results in multiple packets that need to be sent. This does not seem to be harmful, however a cookie is probably sent with every request that is made to the server. If every request needs to be sent in multiple packets, it could take more than one RTT until the server has received the entire request with a new TCP connection.

### 2.1.6 HTTP/2

HTTP/2 [35, 55] is the current version of HTTP and is released in RFC 7540[25] in 2015. HTTP/2 is based on SPDY [58] that was developed at Google. HTTP/2 introduced several important optimizations for the known problems of HTTP/1.1.

#### Multiplexing

A big change of HTTP/2 is that requests are multiplexed. To make this possible, HTTP/2 added streams. Every request and response is sent on a stream with a particular stream ID. These requests and responses are divided into frames which contain these stream ID's. This makes sure that the other endpoint can identify which frame belongs to which request or response. The union of all these frames with the same stream ID is referred to as a stream with the given stream ID.

When a client needs to request a particular resource, it creates a frame that contains the request and assigns the frame with an unused stream ID to identify the request with this chosen stream. Next, the server receives the frame with the request, it creates a new frame containing the response of the request and assigns it with the stream ID that was chosen by the client. Finally, the client receives the frame with the response inside it. In practice, the response is mostly larger than one TCP packet, which means that the response is split across multiple frames divided in multiple TCP packets. It can map the response to the request with the stream ID that is contained in the frame.

With this new feature, the HOL blocking from HTTP/1.1 with pipelining has been solved at the application-layer. When multiple resources need to be requested by the client, the client can combine multiple frames containing requests into a single TCP packet. This packet is sent to the server, which processes these requests and puts the resources in the corresponding frames with their stream ID and sends these resources back to the client. One TCP packet can contain multiple frames with multiple different resources in their frames with their respective stream ID's. This is illustrated in figure 2.4

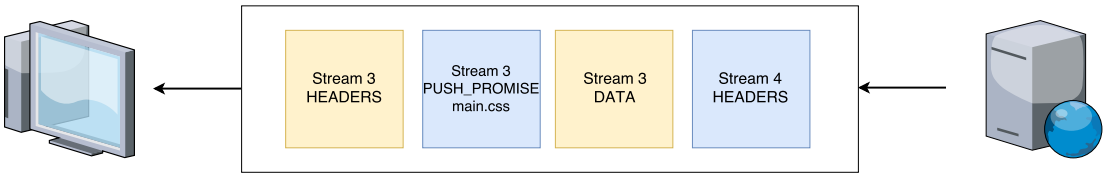



*Figure 2.4: Different frames are multiplexed over a single TCP connection.*

### Priorities and dependencies

When multiple resources are requested, there are always resources that are more important than others or are dependent of other resources. With this in mind, the HTTP-WG added priorities to HTTP/2. This is achieved by using dependencies and weights. Weights are used to indicate that a particular stream is more important than the other one. A high weight value means that the stream is very important. Values range from 1 to 256 (inclusive). Dependencies are used to specify what resources are needed before an other resource can be used.

To illustrate the use of this functionality, assume we have a webpage index.html that contains the following resources: main.css, main.js, jquery.js and banner.jpeg.

1. Client starts requesting index.html
2. While the client is processing index.html, it discovers main.css and banner.jpeg, so it starts requesting these two. Even though, main.css is more important than banner.jpeg, normally the client receives these resources simultaneously.
3. Next the client discovers main.js and jquery.js. Now the client is downloading four resources from the webserver simultaneously.

Even though the client knows that main.css is more important than all the other resouces, except index.html and that both javascript files are more important than the image, with HTTP/1.1, they were using the same amount of bandwidth. With HTTP/2, it is possible to assign weights to indicate which resources are more important than others. With the previous example, it would be a possible scenario to assign index.html a weight of 255, css files 100, javascript files 60 and images 32. So when the css, javascript and image files were downloading, main.css would use $\frac{100}{100+60+60+32}$ of the available bandwidth, while the image only uses $\frac{32}{100+60+60+32}$. The result of this dependency tree is shown in figure 2.5.
When the client wants to indicate that a particular file needs to be downloaded before another one, it can indicate that resource B has a dependency on resource A. In the previous example, the client could indicate that it first wants to receive index.html, prior to receiving any other file. It also can

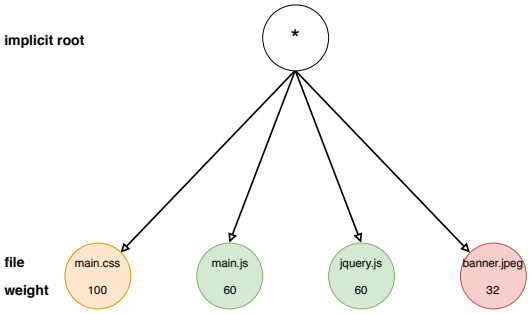



*Figure 2.5: All files are directly under the root. This indicates that all files are downloaded simultaneously. The amount of bandwidth each file receives is indicated by the weight.*

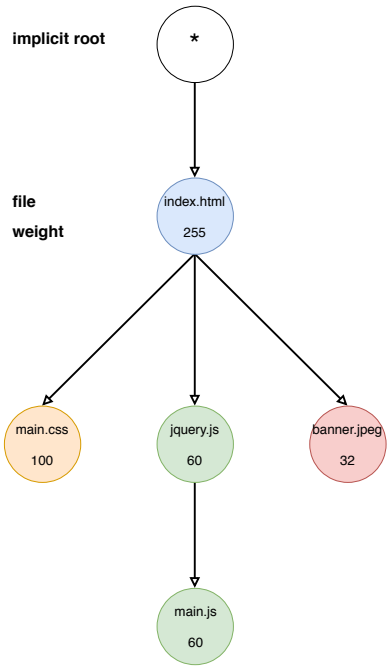



*Figure 2.6: Index.html is on top of the tree, which means it is downloaded first. Afterwards main.css, jquery.js and banner.jpeg are downloaded simultaneously. Downloading main.js starts whenever jquery.js is done.*

indicate that jquery.js needs to be downloaded before main.js. The client would add a header in the request from main.js that main.js is dependent on jquery.js. By doing this, the webserver would first sent jquery.js, together with other resources like main.css and banner.jpeg. When the client has fully received jquery.js, the webserver starts sending main.js, together with other resources that are still being sent. The result of this dependency tree is shown in figure 2.6.

### HTTP/2 push

When a resource is requested, the server could know what other resource are often requested after the first resource. However, with HTTP/1.1 the client has to request this resource before the server can sent this, even though the second resource is always requested directly after the first resource has been requested, which is illustrated in 2.7. HTTP/2 solves this problem with HTTP push. With push, the server can send a resource that is often related to another resource that is being requested by the client. By doing this, the client does not have to waste a RTT by discovering this resource from the other resource and request this particular resource from the server. This is illustrated in figure 2.8.

### Stream-level flow control

Even though TCP already implements flow control (see section 2.2.3), this only applies at the connection level. This means that a server cannot overwhelm a client with data. However this does not stop a server from using up all the data from the connection with one particular resource.
To illustrate this, assume that a client requests one large file (e.g., video file, virtual machine) and 50 smaller files. When the server responds to the multiple requests, it uses most of the connection with the large file. If the client is overwhelmed with the data of the large file, it needs to throttle the entire connection. With stream level flow control, it can throttle the particular stream, by not sending a WINDOW_UPDATE frame (WINDOW_UPDATE frame tells the other endpoint it can receive more bytes for a particular stream). By throttling the particular stream, the other requests can be handled successfully. Note that stream level flow control is only meant for data frames. This

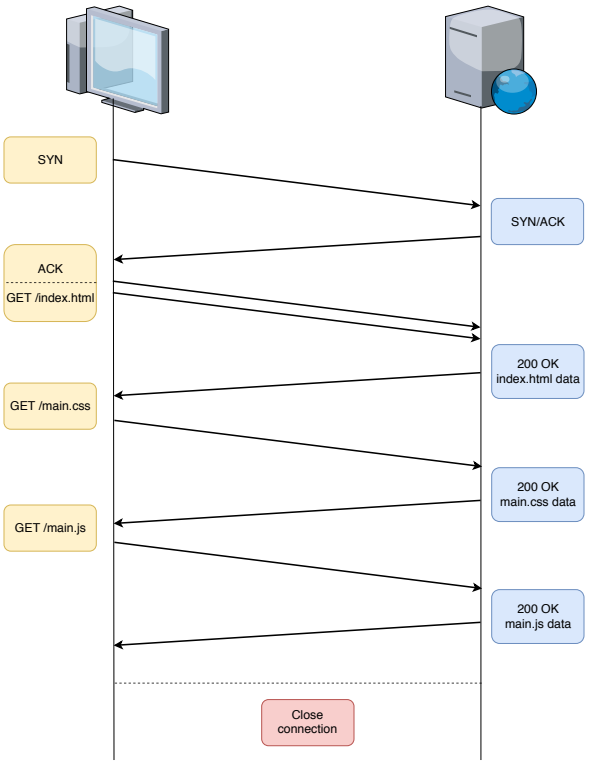



*Figure 2.7: Main.css and main.js needs to be requested after index.html is obtained.*

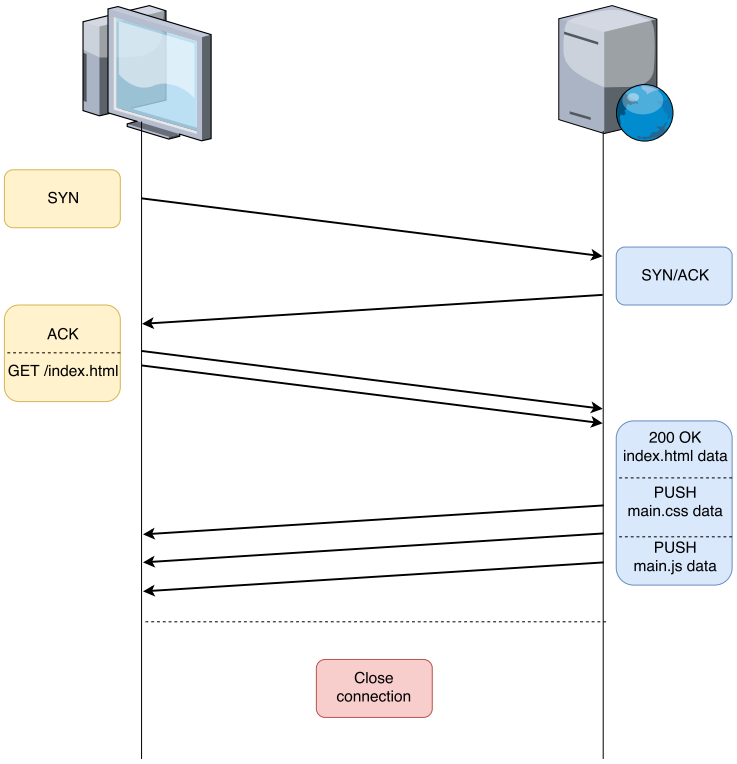



*Figure 2.8: When index.html is requested, main.js and main.css are pushed alongside index.html.*

is to make sure that control frames (e.g., WINDOW_UPDATE frame) are not blocked by DATA frames. DATA frames only contain application-specific data, while control frames contain important

information about HTTP/2 management.

It can easily be seen that HTTP/2 priorities could be implemented by the use of flow control. The client would divide the available bandwidth by the use of its credit based functionality and give highly valued resources more credits than resources that are not so valuable for the webpage. When the client wants to receive a particular resource before any other resource, it can stop sending WINDOW_UPDATE frames for all the other streams except the stream which requested that particular resource. By using flow control, the client can enforce priorities to a server which otherwise might not follow the priorities that were set by the client.

#### Other optimizations

HTTP/2 is different from the previous versions in that data is being sent in binary format instead of ASCII like the previous versions. HTTP/2 also uses header compression and can remember headers that need to be sent in every request or response. The header compression format that is used for this purpose is HPACK, which is created especially for HTTP/2. With this, large cookies only need to be sent once instead of every request or response.

### 2.1.7 HTTP/2 limitations

Even though HTTP/2 solves a lot of issues from HTTP/1.1 (e.g., HOL blocking on application-layer, large request headers), it still has some issues relating to the use of TCP. Even though HTTP/2 solves the HOL blocking of the application-layer, it encountered HOL blocking at the transport-layer. Also, the connection is limited to the network the device is on. When the device changes its network, the existing connection is terminated and a new one needs to be established. Lastly, with HTTP most requests that are made to the server are short-lived. This means that only a few RTT are spent to request and respond resources before the connection is closed. However, the connection establishment cost is at least one RTT due to TCP and an additional two RTT for TLS. This means that for HTTP/2, which are mostly short-lived connections, the connection establishment is a costly issue. The details of these issues can be read in more detail in section 2.2.5.

## 2.2 TCP

### 2.2.1 Definition

Transmission Control Protocol [45, 35] or TCP is a protocol at the transport layer and provides a service abstraction for a reliable and in-order connection between two endpoints on an unreliable network. This means that it is guaranteed to deliver the bytes that were sent and in the same order. This connection between two endpoints is identified by a tuple of size 4. This tuple contains the source IP address and port number, and destination IP address and port number. To keep the connection open, these values cannot be changed or else the connection becomes invalid. This means that, when the IP address of an endpoint changes, the TCP connections becomes invalid and a new connection needs to be established.

### 2.2.2 Three-way Handshake

TCP is connection-oriented because before two processes can start communicating with each other, they need to perform a handshake. By using this handshake, both endpoints agree on several parameters, such as the MSS and window scaling, and to let the other endpoint know of the initial sequence number that is going to be used before they can start transferring data to each other. Window scaling is added because the default receive buffer was limited to 65535 bytes. With window scaling, the value of the receive buffer is shifted by the window scale value. All these values must be known prior to starting sending any data (e.g., when the MSS is only 1200 bytes and the sender starts sending TCP packets with size 1460 bytes, all packets are dropped because they are to large.). The handshake of

TCP is also called the three-way handshake, which is illustrated in figure 2.9. The handshake starts with an endpoint sending a SYN packet which includes a random sequence number, flags and options. An example of an option in the SYN packet is the use of SACK [47] (Selective Acknowledgements) during the connection lifetime. SACK makes it possible for the receiver to inform the sender which packets were received successfully, which makes it easier for the sender to only retransmit packets that were lost. However, the receiver is limited with three ranges that it can acknowledge this way, because SACK is used as a TCP extension, which is limited to 40 bytes.

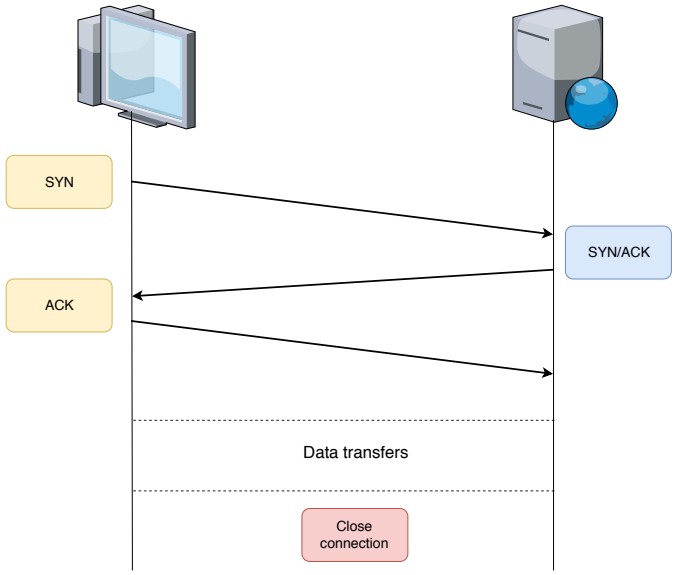



*Figure 2.9: Three-way handshake of TCP.*

When the other endpoint receives the SYN packet, it increments the sequence number given in the SYN packet, generates a random sequence number for itself and sends it back in a SYN/ACK packet. The sequence number of the first SYN packet is incremented because it is the next expected sequence number it needs to receive. This value is set on the ACK field of the packet.
When the first endpoint receives the SYN/ACK packet, it responds with an ACK packet with its own sequence number incremented by one and an ACK field that has the value of the sequence number received from the other endpoint, incremented by one. After the handshake, both endpoints know for sure that the other endpoint knows what sequence number to expect next.
This three-way handshake of TCP needs 1 RTT to be established. This delay makes new TCP connections expensive, especially with short-lived connections (e.g., HTTP).

### 2.2.3 Flow Control

Flow control is a mechanism to avoid the possibility of overwhelming the receiver with data. Each endpoint of the TCP connection keeps a receive buffer for the data that arrives from the connection. The application that uses the TCP connection reads the data from this buffer. However, this does not mean that the receiver reads the data immediately from the buffer when data has arrived. The application could be busy doing some other tasks. With this in mind, it can be seen that the buffer could overflow when the receiver does not read any data from the buffer or at a lower rate than the sender is sending data at.

To let the sender know how much data is available in the receive buffer, the receiver sends the rwnd (receive window) with its acknowledgments. The rwnd is calculated by subtracting the amount of data in the receive buffer from the total amount of data that fits in the receive buffer. When the sender receives an acknowledgement from the receiver with the rwnd, it can adjust its sending rate. When the sender has transmitted so much data that the receive window is zero, it starts transmitting

packets with one byte in the data field. These packets need to be acknowledged by the receiver, so that the sender knows when it can start transmitting packets with relevant data again. The sender sends the one data byte packets to get an update of the rwnd. If the sender did not send these packets and the last acknowledgement that was sent by the receiver had a rwnd with value zero, the sender would be blocked. This is because the sender would not know when there is space available in the receive buffer of the receiver.

### 2.2.4 Congestion Control

Congestion control is another mechanism of TCP to throttle the sending rate of the sender. This mechanism is not intended to avoid overwhelming of the receiver, congestion control is rather employed to avoid overwhelming the underlying network between the sender and the receiver. This happens when a network device (e.g., routers, switches) on the path between the sender and receiver cannot keep up with the network traffic that it needs to process. When a network device is processing a particular packet and other packets arrive, the arriving packets are buffered on the network device. When this buffer is full, the network device needs to drop some packets to keep up with the traffic that is incoming. For this reason, congestion control is added to also be aware of the devices that needs to forward these packets to the receiver. When the sender is unaware of the congestion in the underlying network and the receiver keeps advertising a large rwnd, the underlying networks get congested and more and more packets will be lost, thus the rwnd will not get any smaller. To avoid a congested network, congestion control was added to TCP. The first thing in congestion control is the congestion window or cwnd. cwnd is the amount of data the sender has transmitted to the receiver that has not been acknowledged yet by the receiver. Unlike rwnd, cwnd is not exchanged between the sender and the receiver. However, the maximum data in flight has to be the minimum of rwnd and cwnd. The cwnd variable is only kept locally by the sender. This variable is changed throughout the different phases of congestion control.

#### Slow Start

A TCP connection starts in the slow start phase of the congestion control. The initial cwnd variable was at the start of TCP set to only one packet. However, over the years this initial value has been raised a couple of times up to ten packets, which was proposed by Dukkipati et al. [32] and is an experimental RFC in RFC 6928[31]. This makes it possible to transmit ten TCP packets, which equals to approximately 15KB in the first RTT. They concluded that raising the initial cwnd value to ten, reduces latency, faster recovery from loss and allows short transfers to compete with bulk data traffic. During the slow start phase, the value of cwnd grows exponentially, it is incremented by one for each ACK that is received by the sender. In this manner, the value of the cwnd variables gets doubled each RTT. With this, the sender and receiver can find out the available bandwidth on the network. The rate at which the sender can send packets is increased by the slow start phase until one of the following three conditions are met. The first one is when a loss is detected. When a loss is detected, the sender and receiver are assumed to have reached the available bandwidth. The second condition is when the cwnd exceeds the value of rwnd and the third condition is when the cwnd reaches a threshold. This threshold is referred to as sshtresh. When one of these three conditions are met, the connection goes to the congestion avoidance phase. The slow start phase can be entered again when version dependent criteria are met, which are explained in the next paragraph for TCP NewReno.

#### Congestion Avoidance

While the slow start phase searches for the limit of the network, whenever a packet loss occurs, the algorithm assumes that the packet loss occurred because the network is congested. This means that somewhere on the network, a link could not keep up with the data that is being sent by an endpoint. The cwnd variable gets reset to a value based on the chosen algorithm. What happens next, depends on the chosen congestion avoidance algorithm.

TCP NewReno is an example of a congestion avoidance algorithm. During the congestion avoidance phase, the value of cwnd is incremented by only one MSS for each RTT until the algorithm gets feedback of any congestion on the network. The value is only incremented by one because the value is already at some point that congestion was encountered in the past. By only incrementing the cwnd by one, the algorithm tries to avoid congestion while still trying to get the most out of the available bandwidth. TCP NewReno gets feedback about the network through loss events. There are two loss events TCP NewReno looks at.
Firstly, there is the retransmission timeout. This ensures that the packets are delivered when no feedback has been received by the receiver. When a packet is sent, an alarm is set for an amount of time that is calculated by using RTT samples. The RTT samples that are used to calculate this time, are the smoothed RTT and the RTT variation. When this loss event occurs, the cwnd is reset to one MSS and slow start phase is entered again.
The second loss event is raised when three duplicate ACKs are received, which is an ACK that was already received. This happens when the receiver receives a TCP packet with a sequence number that is higher than the expected sequence number, which causes the receiver to sent the previous ACK again. When this happens, TCP NewReno does a fast retransmit and halves the cwnd variable. Next, it will set sshtresh to the value of cwnd and it will enter the fast recovery phase. During this phase, for every duplicate ACK, it will send a packet that has not been sent before.

### 2.2.5 Limitations

The first major limitation from TCP is that, like HTTP versions prior to HTTP/2, it suffers from HOL blocking. Because TCP is a transport protocol, the HOL blocking is located at the transport-layer. When ten packets are sent on a TCP connection and the first one is lost, the nine others are buffered by TCP because they cannot be received by the receiver, which is on the application-layer and expects that the data that is received, is in the same order as was sent. This is because TCP does not know what kinds of data are present in the packets. When HTTP/2 is used, they could contain data from different streams, however when HTTP/1.1 is used, it could all be data from a single response.
TCP promises in-order delivery and because it does not know what data is present in the different packets, the other nine packets need to be buffered until the first packet has arrived. For this to happen, the sender first needs to notice that the packet is lost. This can be due to an ACK from the receiver or with a timeout. Afterwards the sender retransmits the lost packet to the receiver. When the lost packet has finally arrived at the receiver, the other nine packets can also be processed.

To illustrate this even further, let us assume that a client is using HTTP/2 to request a small image logo.jpeg, main.css and main.js. Let us also assume that the congestion window is large enough to send ten packets, which is also illustrated in figure 2.10.

1. The client starts by issuing requests for the given resources. Stream 5 is used for main.css, stream 7 for main.js and stream 9 for logo.jpeg.

2. The server responds to these requests on the different streams. It first answers to the main.css request and this results in two packets. Next it sends four packets for the javascript file and lastly, the server sends five packets for the logo.jpeg file. This results in 11 packets that are sent to the client in the order that was just described.

3. Due to the lossy network, the first packet containing the first part of main.css gets lost. All the other packets arrive at the client.

4. Because TCP does not know that the missing packet contains data for main.css, all TCP packets which are sent after the missing packet are buffered, even though HTTP/2 does not need the missing packet to process logo.jpeg and main.js.

5. The server notices that the first packet was lost due the fact that the receiver has sent a couple of duplicate acks. So the server retransmits the lost packet to the client containing the first part of main.css.

6. When the first part of main.css finally arrives at the client, the other packets containing the javascript file and the image file can also be processed by the client.

This overhead of buffering could be avoided if TCP would have known that only the first two packets contains the css file, the next four contains the javascript file and the last five packet contains the image file.

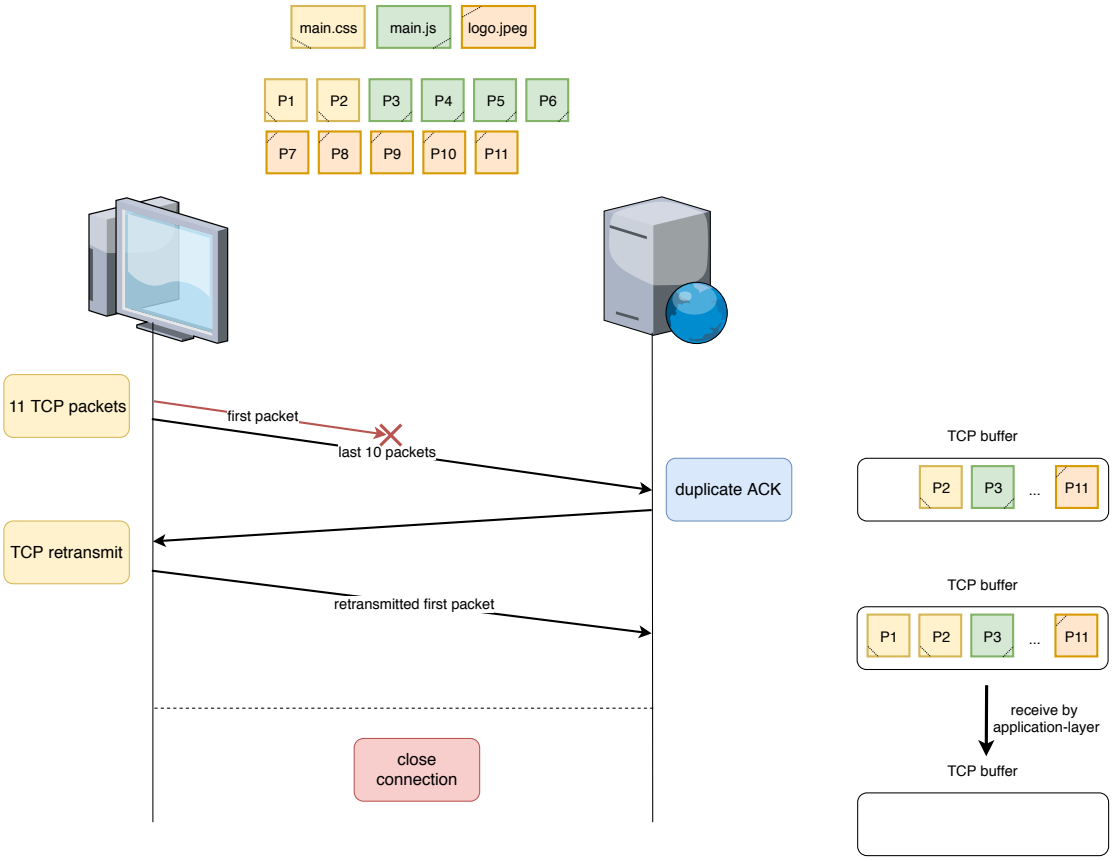



*Figure 2.10: Logo.jpeg and main.js are blocked because the first TCP packet, that contains the first part of main.css, is lost.*

Another issue from TCP is the slow connection establishment. When a client starts a TCP connection with a server, it needs one RTT before the client can start sending relevant data. When TLS is enabled, it even takes three RTT. When only a few small requests need to be made (e.g., small requests and responses that can be bundled in one or two packets), this introduces an extra overhead.

### TCP extensions as a solution

One could debate that some of these issues could also be fixed by extending TCP with the use of TCP extensions, as is the case for the slow connection establishment. For the connection establishment, TCP Fast Open (section 3.4) was added as a TCP extensions. However, extending TCP has become a nontrivial task to do [37]. The internet already exists for a couple of decades. All this knowledge let to middleboxes (e.g., NATs, firewalls, proxies) that try to further optimize connections by changing fields and options in the TCP header. However, because middleboxes change fields and options, it became a much harder task to introduce new TCP extensions. This has led to slower standardization of new TCP options because they had to take the middleboxes into account when making decisions.

Multipath TCP (MPTCP) [52] is an example of a TCP extension which had to make changes to make sure it was usable on todays internet. An example of one of the changes they had to make was during the negotiation of MPTCP in the TCP handshake. Normally, the client starts the TCP handshake and adds MP_capable option to indicate that it supports MPTCP. On receiving the SYN packet of the client, the server responds with a SYN/ACK packet with the MP_capable option to

indicate it that it also supports MPTCP. However, with middleboxes mangling with TCP options, they could disable MPTCP in the SYN or SYN/ACK packet. Disabling the MPTCP option in the SYN packet does not raise any issues. However, disabling MPTCP in the SYN/ACK packet does. When this happens, the server expects the use of MPTCP while the client thinks that the server does not support MPTCP. To mitigate this issue, the client needs to set the MP_capable option in its ACK packet. By doing this, the server is also sure that the client has received its MP_capable option from its SYN/ACK packet. This is only one of the changes that had to be made in MPTCP to support todays internet.

To illustrate the problem of some middleboxes even further is that even TLS/1.3 has seen some issues when it first was enabled by Google Chrome and Firefox [56]. Instead of implementing the spec of previous TLS versions correctly, they made some assumptions that particular fields were always present in all TLS versions. However, with TLS/1.3, some fields (e.g., session_id, ChangeCipherSpec) that were present in previous TLS versions were removed which resulted in middleboxes failing because they could not find these fields. To fix this, TLS/1.3 was changed [12, 13] rather than waiting for middleboxes to correctly implement TLS/1.3, which could take several years. They changed it by making the TLS messages look like TLS/1.2 and added a version extension where TLS/1.3 implementations know that it negotiated TLS/1.3 instead of TLS/1.2. They also introduced several fields back (e.g., session_id, ChangeCipherSpec) which were originally removed from TLS/1.3.

Honda et al. [37] examined the extensibility of TCP. They concluded that care must be taken when implementing new features to TCP, by using SYN and SYN/ACK packet to negotiate the new feature and to make sure a fallback is possible, and that middleboxes have to be taken into account. An example of this was already given with the issue with MPTCP and the SYN/ACK packet. Another example is that endpoint should not assume that sequence numbers arrive unmodfied because some middleboxes change the sequence number which is an attempt to randomize sequence numbers. Hesmans et al. [36] did a similar investigation to check whether TCP extensions are safe from middleboxes. They came to a similar conclusion that middleboxes interfere with various options in TCP packets (e.g., changing sequence numbers). They also tested how TCP behaves when middleboxes interfere, one of these were when middleboxes change sequence numbers, that the performance of the connection drops when TCP SACK is enabled. This is because sequence number randomizers do not change the value of the SACK options, which results in invalid sequence numbers. An other example is when the client sents a SYN packet containing the window scale option. When the server receives this packet, it responds with a SYN/ACK packet also containing the window scale option. When middleboxes change the value of the window scale option in the SYN/ACK packet, the server thinks that window scaling is enabled, while the client thinks that window scaling is disabled. As previous stated, this could by solved by requiring the option to also be set in the ACK packet of the handshake, as was done in MPTCP.

## 2.3 UDP

### 2.3.1 Definition

User Datagram Protocol or UDP is, like TCP, a protocol at the transport-layer. This protocol only adds a checksum to the network-layer and multiplexes packets with ports. This last functionality means that every packet is assigned a port that is being used by the socket that is sending the packet, which is also done in TCP. In contrast to TCP, UDP is an unreliable protocol that does not order the arrived packets. It is a connectionless protocol, which means that there is no prior connection establishment like the three-way handshake of TCP.

### 2.3.2 Advantages over TCP

Even though UDP is an unreliable protocol, its disadvantages are also its advantages over TCP. Firstly, because UDP does not provide any flow control or congestion control, it sends the data immediately. This gives the application that is using UDP more control when the data is being sent to the receiver. TCP cannot do this because it needs to make sure that the receiver can handle the

incoming data by looking at the rwnd and it must also look at the congestion on the network by looking at the cwnd. If one of these cannot handle the data, it is buffered at the sender until it can be sent, which introduces a delay that UDP does not have. Secondly, UDP is connectionless so there is not RTT wasted on connection establishment. As mentioned in section 2.2.2, TCP first needs to several parameters (e.g., initial sequence number, option support (e.g., window scale, SACK)). TCP must also keep connection state in memory (e.g., rwnd, cwnd), which is not needed in UDP.

Because of this bare-boned transport protocol, UDP is likely to be used more often when the data can arrive out of order or can be lost. As a result of providing almost no functionality, UDP has been being used to provide other reliable protocols, such as Real-time Transport Protocol (RTP). Even though TCP provides in-order delivery, reliability and fairness on the network, this is not always needed for some applications. It would be also possible to implement reliable protocols on top of the network-layer instead of creating one on top of UDP, which has been done for Stream Control Transmission Protocol (SCTP).

## 2.4 QUIC

### 2.4.1 Definition

QUIC is based on Google QUIC that was designed by Jim Roskind at Google [6]. QUIC is a connection-oriented transport protocol on top of UDP. In 2016 the QUIC-WG was established to work on the standardization of QUIC, which is still ongoing.
A QUIC connection is created by sending QUIC packets in UDP datagrams. Within this connection, QUIC multiplexes different requests with the use of streams, which is a borrowed concept of HTTP/2. On top of that, QUIC uses TLS/1.3 to provide a secure connection. Everything, including the congestion avoidance algorithms are implemented in user space instead of kernel space, as is the case in TCP. By implementing this in user space, it is easier to change and adept to the network where the device is on. QUIC implements best-practices learned from TCP and uses this to create a new reliable transport protocol, which solves some critical issues that were encountered in TCP, where a couple of them were addressed in section 2.2.5. The following section discusses the most important solutions for these limitations.

### 2.4.2 Solutions for TCP's limitations

The first and main issue that is solved from TCP, is the HOL blocking (Section 2.2.5). Basically, when the first packet of 11 packets is lost, the remaining ten packets need to be buffered by the receiver until the first packet is retransmitted and delivered to the receiver. QUIC solves this issue, by utilizing the concept of streams that was introduced in HTTP/2 (Section 2.1.6). HTTP/2 uses streams to send HTTP requests and responses. Every request that is made by an endpoint, is sent on a different stream.
QUIC does the same by letting an endpoint send data on a different stream. To send data from a stream, QUIC borrows another concept from HTTP/2, which is frames. The data is divided in a collection of stream frames, which is one of the types of frame that QUIC uses. These frames are combined in packets and sent to the receiving endpoint. Frames from different streams can be combined in a single packet, like in HTTP/2. This application data and TLS data is transferred using STREAM frames, other frames are used to transfer other kind of data. An example is flow control. When the sender wants to indicate it cannot sent any data anymore because the receive window is full, it can sent a STREAM_BLOCKED frame. When the receives wants to communicate to the sender that it has more room in its buffer, it can sent a MAX_STREAM_DATA frame, which contains the maximum amount of data that is allowed to be sent by the sender.

The reason that TCP has the problem of HOL blocking is that it buffers later packets when the first packet is lost. QUIC solves this issue by buffering the individual stream frames instead of packets itself. This can be illustrated by an example. Assume that a client requests five different resources. The first parts of resource one and three arrive in the first packet. The last part of resource one arrives together with resource five in the second packet. The third packet contains the last part of resource

three together with resource two and four. When the first packet is lost, it needs to be retransmitted by the sender and thus the receiver does not have the first parts of resource one and three. However, because of the multiplexing behaviour of QUIC, the other two packets can be used by the receiver without having to wait for the first packet. The client can view resource two, four and five because they were not lost. For resource one and three, the client needs to wait for the retransmitted packet. Thus QUIC does not suffer from HOL blocking because the three resources in the last two packets could be used by the receiver without having to wait for the first packet. In TCP, the receiver would have to wait for the first packet before using the next two packets. They would have to be buffered until the first packet arrived.

To illustrate this even further, take the situation from section 2.2.5. All TCP packets had to stay within the TCP buffer until the first packet, containing the first part of main.css, was retransmitted. With QUIC, the data from main.js and logo.jpeg can be delivered to the application-layer, while the second packet of main.css stays within the buffer, until the first part has arrived. This is illustrated in figure 2.11

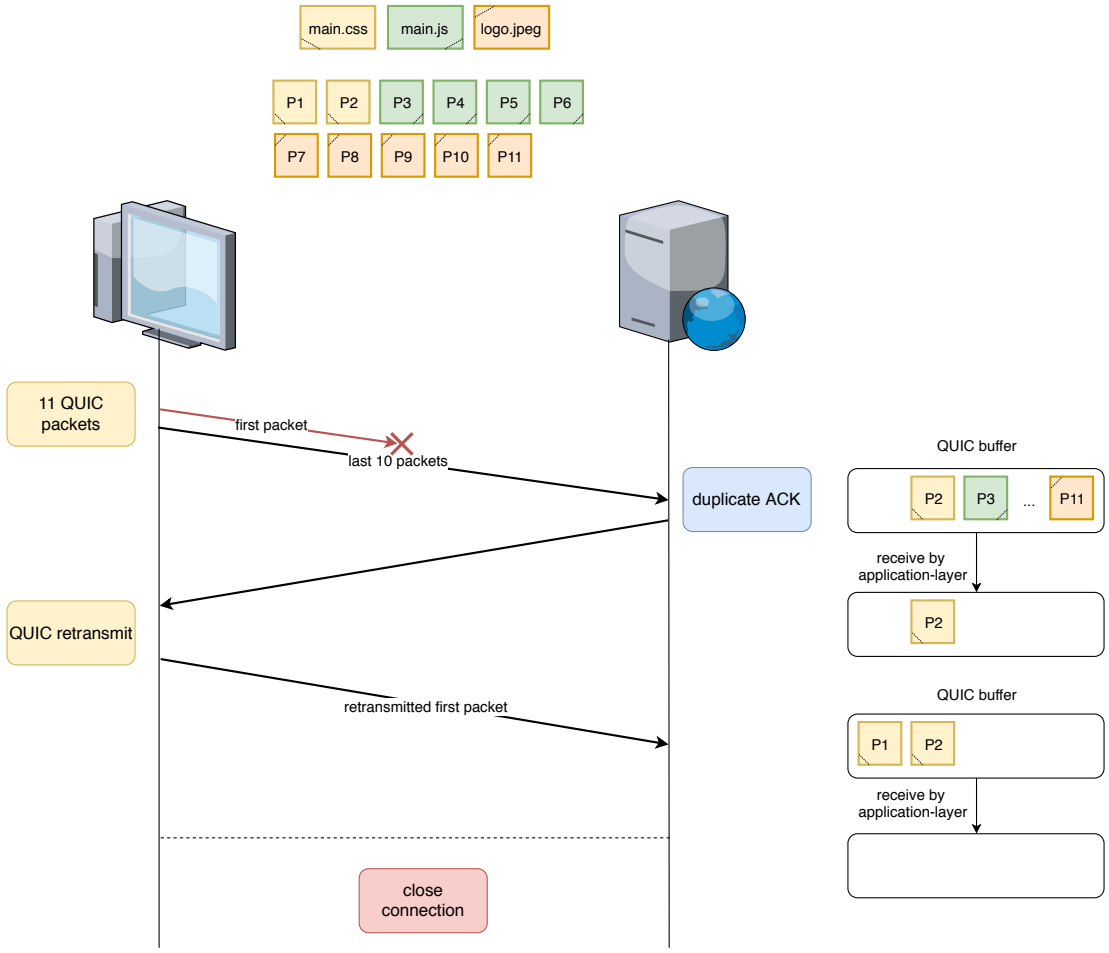



*Figure 2.11: Because QUIC buffers data from streams, rather then the packets itself, main.js and logo.jpeg can be delivered to the application-layer*

The second issue of TCP that QUIC is trying to solve is the slow connection establishment. As stated in section 2.2.5, a new TCP connection needs one RTT before either endpoint can start sending data. With TLS/1.2 enabled, it takes three RTT. QUIC solves this issue by combining the connection establishment and the TLS handshake. The minimum TLS version that QUIC supports is TLS/1.3. By combining TLS/1.3 and the handshake of QUIC, it makes it possible to have a fully encrypted initial connection in one RTT, which is two RTT less than TCP + TLS/1.2, which

is shown in figure 2.12. The handshake of QUIC is explained in more detail in section 3.3. When the client has already connected at least once with the server, the connection setup could be 0-RTT by using session resumption. The client uses the data from the previous session to secure the data that needs to be sent to the server. The workings of 0-RTT are explained in more detail in section 3.4.

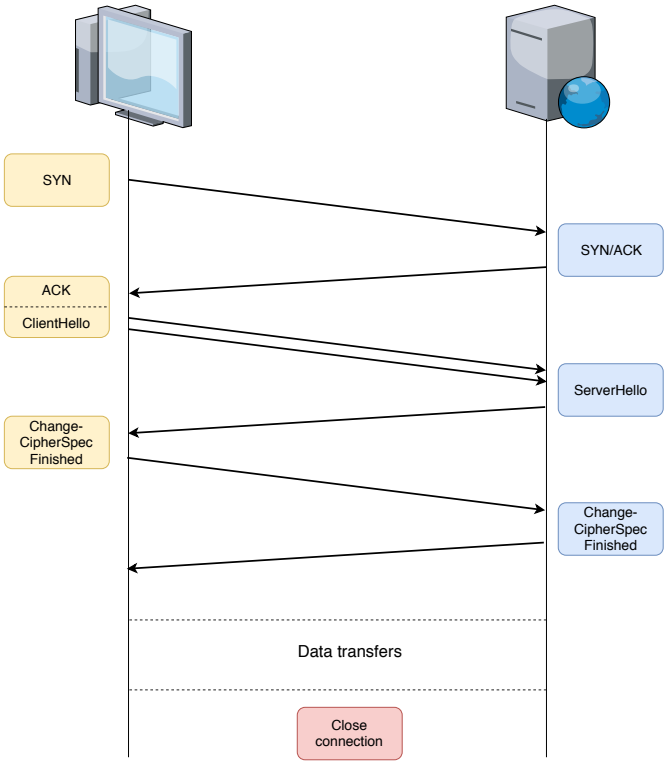



*Figure 2.12: Handshake with TCP and TLS/1.2*

# Chapter 3

# Design and Implementation

QUIC is a complex protocol which contains a lot of best practices learned from TCP. What makes it even more complex is, that it also needs to work on the internet, which is not a trivial challenge in today's internet, with middleboxes changing packets to try to optimize the network. The following section discusses the most important features that are included in QUIC that not only try to optimize the current internet, but also make it more secure by hiding as much information in the packet as possible. This is done by adding the information in encrypted frames, which are included in the payload of the packet (instead of plain text in the header of the QUIC packet) or by even encrypting parts of the header of QUIC itself.

## 3.1 Connection ID

To identify a particular endpoint, TCP uses the four-tuple containing the used ports and IP addresses from the connection, which can be found in respectively the TCP header and IP header from the packet [45, 35, 40]. A common problem for TCP is that the connection does not survive when either endpoints IP address or port changes (e.g., client switches from Cellular to WiFi which result in a new IP address and likely another port ). When this happens, the packets cannot be identified by the other endpoint because the remote information has been changed. The existing connection will get a timeout because the other endpoint does not know the new "location" of the changed device. If the connection is still wanted by either endpoint, it would need to be initiated again. The new connection needs to be established again with a handshake, possibly a TLS handshake as well. This corresponds with an overhead just because one of the endpoints was moved.

QUIC solves this issue by using a single, independent identifier for each end of the connection, namely the Connection ID [22]. This Connection ID has a size between four and eighteen bytes, that is implementation-specific created by the endpoint, which makes sure that the endpoint can identify its own created Connection ID in a QUIC packet, as the destination Connection ID is present in every QUIC packet. Either endpoint can also choose to just use the remote information as in TCP, omitting the Connection ID. However, endpoints must be careful using this method because this limits QUIC to the same issues as TCP, which were described above. This dynamic sized Connection ID makes it possible to lower the packet overhead when QUIC is used for more simplistic setups (e.g., P2P) by using a small Connection ID. However, in more complex situations (e.g., load balancing), it is possible to store some additional information in the Connection ID. In the case of load balancing, the server could encode routing information from the load balancer to the server to the Connection ID so that the load balancer knows how to route the QUIC packet to the server that is handling the connection.

As previously mentioned, both endpoints need to provide a Connection ID or choose to identify the connection using a four-tuple. The logic behind the decision [18] to let both endpoints provide a Connection ID is to resolve linkability issues. Linkability is when an on-path observer can monitor traffic coming from a client and a server, it could identify that it has a QUIC connection by using the Connection ID. However, by using a Connection ID for each endpoint, only one half of the connection is exposed instead of the entire connection. A handy side-effect is that both endpoints can now encode

additional information (e.g., routing information for load balancing) in their respective Connection ID. Because only the destination Connection ID is present in the packet, this allows the endpoints to use the additional information that they encode. An additional advantage of using this approach is that an endpoint can reject a packet which does not have a valid Connection ID following their own logic.

Before the need was identified to have both endpoints provide a Connection ID, the Connection ID was a fixed 64-bit identifier for the connection, which was only provided by the server. However, because the Connection ID could be used to store routing data or other information, the Connection ID was raised from 64-bit to 136-bit [16]. This was due the fact that 64-bit would not be sufficient to encode enough information in larger data centers and secondly that 128-bit ciphers are more available and more efficient than the 64-bit ciphers. The extra 8-bit could be random or store some additional key managing information. Although this is better for endpoints that live in large data centers, this would be an unnecessary overhead for peer-to-peer application. For this reason, the length of the Connection ID was made dynamic.

This works as follows: at the start of the connection, the client first generates a source Connection ID and a destination Connection ID. When the server receives the packet, it uses the given source Connection ID as the destination Connection ID for the next packets it is going to sent to the client. It also chooses to keep the destination Connection ID which was given by the client as its source Connection ID or it can generate a new Connection ID. Both Connection IDs need to be set in a Long header, which is the header of the packet that is used during the handshake of QUIC. When the handshake is completed, both endpoints start using Short header. This header type only contains the destination Connection ID, which is the Connection ID from the other endpoint.

#### Connection ID management in Quicker

The Connection ID is created randomly in Quicker. It encodes the length of the Connection ID in the first byte and the rest of the Connection ID is chosen randomly. Even though the Long header contains the length of the Connection ID, when the handshake is over, the Short header is used to identify packets. These headers do not contain the length of the Connection ID to keep the header as small as possible. For this reason, implementations must make sure they can identify their own Connection ID. This is possible by using a fixed length Connection ID or to encode the length of the Connection ID inside the Connection ID itself, which was the chosen method in Quicker. This is used to make all lengths possible, at the moment the Connection ID is chosen at random, however in the future it could be possible for the user to indicate how the connection is going to be used. If it is only for peer to peer connections, a small Connection ID could be used. On the other hand, when it is used by a server, longer Connection ID are preffered to make as much connections as possible.

During the connection setup, the first initial destination Connection ID in the Initial packet must be kept by both endpoints for cryptographic purposes. The server must also take care to not keep multiple connection states for the same client or two clients choose to have a zero-length Connection ID, which makes it more complex. Zero-length Connection IDs are not fully supported by Quicker yet. They can be parsed from the Long header. However, a connection cannot be created or stored in Quicker with a zero-length Connection ID. For this reason, Connection IDs that are created by Quicker are never zero-length. Each Connection ID is associated to a four-tuple to indicate that the path is validated and uses the given Connection ID. The Connection ID is associated to a four-tuple so data sent to a new path is not linkable to the previous path, which is explained into more detail in section 3.8.

## 3.2 Packet numbers

Packet numbers are used to identify an incoming and an outgoing packet. A packet number is an integer ranging from 0 to $2^{62-1}$. Each endpoint keeps a separate packet number to identify the packets. When a packet number is used, the value must be incremented by at least one.

The decision [17, 20] to increment the packet number with more than one is only interesting when an endpoint wants to test if the other endpoint is performing an opportunistic ACK attack. This attack is done by ACKing every packet number, even though the packet with the corresponding packet number has not been received. The sending endpoint would think the network can handle more packets and will keep increasing its congestion window. By skipping a packet number, the sending endpoint can test the receiving endpoint for this attack. If the receiving endpoint ACKs the skipped packet number, the sending endpoint can close the connection to stop the attack.

As previously stated in section 2.4.2, QUIC solves the issue of HOL blocking by buffering the data from different streams in the QUIC packets instead of the QUIC packets itself. Another property of packet numbers is that they are monotonically [40] increased whenever a packet is sent, which avoids the retransmission ambiguity [39] where TCP had problems with. Retransmission ambiguity [41] in TCP is when a packet has been retransmitted and an ACK has been received for that packet after retransmission. There was no way to check if the ACK was received for the initial packet or the retransmitted packet, unless the timestamp option was used. Because of this ambiguity, the RTT could be calculated wrong. This was solved by Karn's algorithm in TCP. However, to ignore this additional complexity, packet numbers in QUIC can only be used once. When a packet is lost, the data in the packet is sent in a new packet, which could also contain other data that has been sent in other packets that were lost or new data. This raises the question why a packet number is even needed in the first place.

Packet numbers are added for different reasons. Firstly, they are added to make sure the receiver of the packet has a way to indicate that it received a particular packet. To indicate that an endpoint has received a particular packet, it uses ACK frames. ACK frames contain the largest acknowledged packet number, together with ACK blocks. With these ACK blocks, it is possible to acknowledge ranges of packet number, the same way that is possible in TCP SACK. However, in contract to TCP SACK, ACK frames in QUIC can acknowledge more than four ranges [47], which makes it more efficient when more than four ranges are lost. This is because the receiver can indicate more packets that is has received. By doing this, it limits the amount of unnecessary retransmits by the sender.
Secondly, packet numbers have a cryptographic function, as they are used as nonce for the encryption of the payload of the packet. Thus, to be able to decrypt the payload, the packet number is one of the things that is needed to achieve this. The packet number is added when calculating the nonce so that two packets that are identical (e.g., same header, same payload), do not look the same on the wire [38].

However, the packet number was added in the beginning to the header in plain text, which resulted in a linkability concern that on-path observers could see that multiple connections belong together. QUIC provides a way to migrate the connection to a new network so that it survives when an endpoint's IP address or port changes. This process is explained in section 3.8. When a migration happens, the endpoint can change its Connection ID to remove linkability from all packets that were sent with the previous Connection ID. However, with a plain text packet number, it would be possible for a smart observer to link packets from the old Connection ID with the new Connection ID because the packet number follow-up on each other. This is due the fact that packet number, as previously stated, are monotonically increased.

With the knowledge that the packet numbers are used as a nonce for encrypting the payload, it would not be wise to just reset the packet numbers when a connection migration occurs. To solve the linkability issue [38], QUIC encrypts the packet number. With this, on-path observers would not be able to link different packets together which have different Connection IDs. Packet number encryption also has the advantage that it helps against ossification of the packet number, because the packet number is not easily retrieved from the packet and it cannot be changed because the packet number is used for the calculation of the nonce to decrypt the payload of the packet. The disadvantage of this, is that it adds an additional cpu cost with every packet that is sent and received.

#### Packet number management in Quicker

Packet numbers raised a challenge in Quicker. Packet numbers can go up to $2^{62}-1$. However, Quicker is implemented in NodeJS. In JavaScript, the number type is limited to a value of $2^{53}$. Recently [29],

the BigInt type was added to JavaScript. However, at the time of writing, this type was not added yet to NodeJS. To make it possible for numbers larger than $2^{53}$, the Bignum class was added to Quicker, which internally uses a library, currently bn.js [1], that can hold numbers of at least $2^{62} - 1$. Instead of using the library directly, the Bignum class was created to act as a wrapper. In case the library needs to be changed in the future, only the Bignum class needs to be refactored.

The second challenge of packet numbers is that packet numbers are relatively encoded in QUIC packets. To get the real value of the packet number, the largest packet number must be used and compared to the relative value in the packet to determine the real value. This is done by finding the packet number which is closest to the expected packet number, which is the current number incremented by one. As an example, when the current packet number is 0x0e12 and the received relative packet number in the header is 0x13, which only uses seven bits, the packet number of the received packet is 0x0e13. A more complex example is when the current packet number is 0xaa82f30e and the packet that is received has encoded 14 bits with the following value: 0x1f94. The packet number of the received packet is 0xaa831f94, because 0xaa821f94 would be smaller than the current packet number, which is not allowed in QUIC. That is why QUIC implementations must be careful that there is no ambiguity between different values. As an example , this can occur when a sender transmits more than 256 packet at once while only using one byte to encode the relative packet number. By doing this, ambiguity occurs because the first and the 257th number are encoded with the same value even though they represent different values. The real value is determined when the header has been parsed and the packet number is decrypted. The real value is needed to decrypt the payload of the QUIC packet. The entire flow of a QUIC packet in Quicker is explained in more detail in section 3.11

## 3.3 Handshake

To exchange the necessary information about each endpoint and to negotiate the secrets that are going to be used, QUIC also uses a handshake. All the TLS data is sent in stream frames using stream ID 0 and QUIC specific data (e.g., initial max stream id for unidirectional and bidirectional streams, amount of time the connection can stay idle before a timeout occurs), which is also referred to as transport parameters, are sent using an extension in TLS to add it to the TLS data. By doing this, only TLS data needs to be sent and the recipient of the data can parse it using the parse callback from its TLS stack. Adding the transport parameters to the TLS data is probably done to avoid creating a new frame type especially for transport parameters which could only be used during the handshake of QUIC and which would also add even more complexity to the existing logic.

The handshake of QUIC starts by a client sending a Client Hello, which is in an Initial packet that is only used to send Client Hello data to the other endpoint. In this Initial packet, the client chooses its Connection IDs and the QUIC version to start the connection with. In the payload of the packet is data that is used by TLS to negotiate the cipher suite and other secrets. This payload is encrypted by using the default encryption algorithm of QUIC version one, aes-128-gcm [53]. All packets, except for Version negotiation and Stateless reset, are encrypted by an Authentication Encryption with Associated Data [48] (AEAD) algorithm. As for the keys for this encryption algorithm, the version, version-specific salt, Connection ID, packet number and the header in its entirety are used to generate them. All this information can be found in the header of the packet so it is clear that the encryption is not provided to be secure for on-path attackers because they can access the necessary data to decrypt the payload of the packet. However, it provides some protection for off-path attackers because they cannot inject any packet as the keys are generated connection and version specific.

Next, when the server receives the Client Hello packet, it first checks the used version in the header of the packet. The version field is one of the fields of the headers in QUIC that are described in the invariants of QUIC [59]. Thus, these fields will be present in the headers in every version of QUIC. If the given version is not in its own list of supported versions, the server creates a version negotiation packet and sends it to the client. If the version is supported, the server answers the packet with a Handshake packet, which is used to sent handshake related data, containing the Server Hello. By receiving the Client Hello and generating the Server Hello, the server has finished the handshake.

However, the client still needs to finish the handshake by receiving the Server Hello. When the client receives this Handshake packet, the handshake is complete at the client and it can derive keys from its TLS stack to protect the payload of all the packets transmitted. The packets which are encrypted with these new keys are called Protected 1-RTT packets.

The client still needs to indicate to the server that it has finished the handshake. This data, coming from its TLS stack, is sent with a Handshake packet. All the data after this packet is sent with a Protected 1-RTT packet. After receiving the last Handshake packet from the client, the server knows that the client has finished the handshake. The server could also send a session ticket to the client to make session resumption possible in the next connection. The ticket is sent in a Protected 1-RTT packet because the client already finished the handshake. The handshake of QUIC is illustrated in figure 3.1. When one of these first packets is lost, QUIC uses a timed based loss detection to retransmit these key Handshake packets. This is explained into more detail in section 3.7.1

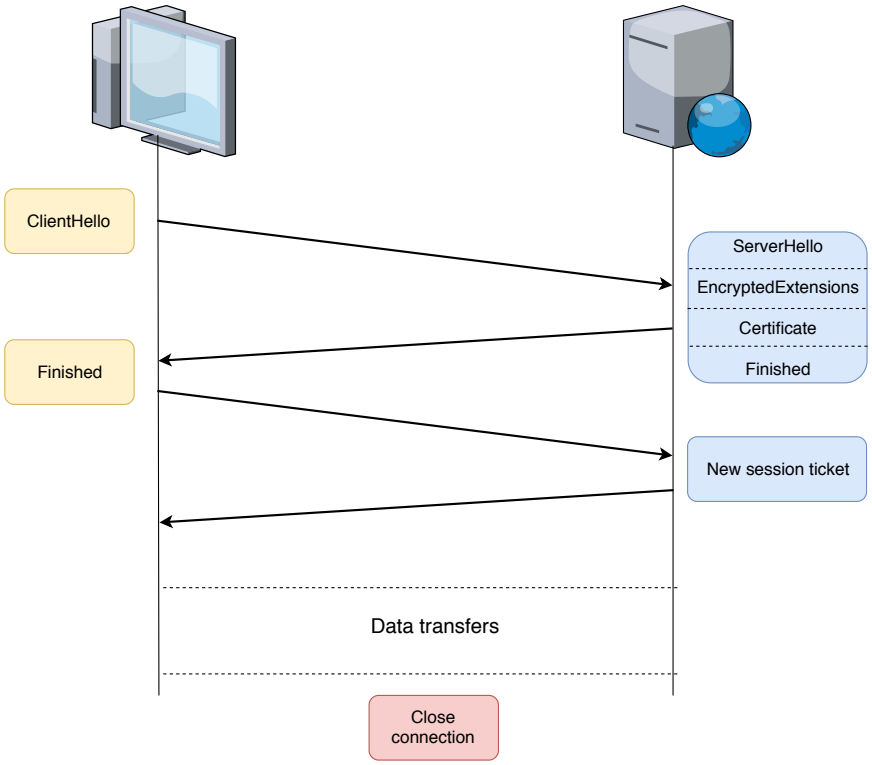



*Figure 3.1: Handshake with QUIC which only takes one RTT for a secure connection*

A server can choose to remain stateless until it is sure that the packet comes from the one it claims to be. This is to make sure the server does not need to keep state if it receives a large number of Initial packets by an attacker. This is done by the Retry packet. When the server receives an Initial packet, it passes the TLS data to its TLS stack, which returns a HelloRetryRequest. This HelloRetryRequest is passed back to the client in a Retry packet. On receiving this packet, the client resets its transport and connection state and sends a new Initial packet with the additional data that provides enough information to the server to know that the client is at the address it claims to be. This is because the HelloRetryRequest contains the cookie extension, which is used to validate the client. On a session resumption, an implementation can add this cookie in the NewSessionTicket to avoid an unnecessary stateless retry. For this to work, the cookie that is provided must be hard to guess by a client to make sure that an attacker could not use this for a possible attack.

To prevent an amplification attack with QUIC, the Initial packet that is sent by the client, must be padded to make the QUIC packet at least 1200 bytes. When the server responds to the Initial packet, it can only send up to three packets until the path must be validated. This is done by sending a

PATH_CHALLENGE frame along with the Server Hello data. This frame contains a 8-byte random payload that must be hard to guess. When the client receives the PATH_CHALLENGE frame, it needs to reply to it with a PATH_RESPONSE frame, which contains the exact same data which was in the PATH_CHALLENGE frame. A legitimate endpoint has the cryptographic keys to decrypt the packet and encrypt a new packet containing the PATH_RESPONSE frame. Thus, the highest amplification factor that can be achieved is three, because the Handshake packets which contain the Server Hello, can only be 1200 bytes. Because the Initial packet contains at least 1200 bytes and the three Handshake packet together contain at most 3600 bytes, the amplification factor is only three.

**Handshake in Quicker**

To be able to do the handshake, Quicker uses OpenSSL to provide the TLS data. This data is made available with a wrapper class that we have created in NodeJS itself to provide the functionalities of OpenSSL in TypeScript. See section 3.12 for more details of this wrapper class. By making the functions available from OpenSSL, we created a handshake handler in TypeScript that receives the handshake data from stream 0 and passes it to OpenSSL, which processes this data. To make the transport parameters available for Quicker, we have created a function which OpenSSL calls when extension data needs to be added or parsed.

The transport parameters are added by the wrapper class when the ClientHello and the EncryptedExtensions are constructed. This is done by passing the transport parameters from TypeScript to C++, which keeps the transport parameters in a member variable. When OpenSSL calls the callback function to add the transport parameters, the value of the member variable is simply passed to OpenSSL. During the processing of the ClientHello and the EncryptedExtensions by the other recipient, OpenSSL calls the parse callback function from the wrapper class. Inside the wrapper class, the data of the extension is kept in a member variable, which can be accessed later in TypeScript. The wrapper class itself does not parse the data, it simply keeps the data which makes it possible to parse it in TypeScript. Thus, both the constructing and parsing of the extension data is done in TypeScript to make it more clear what happens with the transport parameters in TypeScript itself.

While the handshake is not complete yet, Long headers are used to wrap the frames. These headers contain a payload length field, that indicates the size of the packet number and the payload itself. This makes it possible to combine multiple QUIC packets together into a single UDP datagram, when the combination is still under the MTU. This is referred to as coalescing of packets. This makes it possible to combine the Initial packet with a 0-RTT packet, which is explained in section 3.4. Normally, the Initial packet needs to be padded until it has a size of 1200 bytes. By coalescing the packet with a 0-RTT packet, the 0-RTT packet can be used as padding. Instead of using useless padding frames, the bytes are better utilized with a 0-RTT packet containing request. Coalescing of packets is not fully supported by Quicker. It can handle incoming coalesced packets, however Quicker cannot construct it by itself.

## 3.4 0-RTT

QUIC provides a way for connections to start exchanging data without waiting for a single RTT for the handshake. This is possible when the client starts a connection with a server that he already contacted before, thus the connection is started with additional session information from the previous connection. Using this information, it is possible to export encryption keys that can be used to encrypt the payload. The server that is receiving this payload, can export the same encryption keys from the session information that is provided in the Client Hello packet, which is already explained in section 3.3. Next, the server can decrypt the payload using the early data keys. This 0-RTT handshake is illustrated in figure 3.2.
This mechanism is possible because QUIC uses TLS/1.3 for the encryption of data. With TLS/1.3 it is possible, when using session resumption, to encrypt data using early data keys. However, this early data encryption is less secure than the encryption with keys exported after the handshake. The early data keys are susceptible to replay attacks [44]. For this reason, the data sent with early data needs to be idempotent and should not be sensitive data.

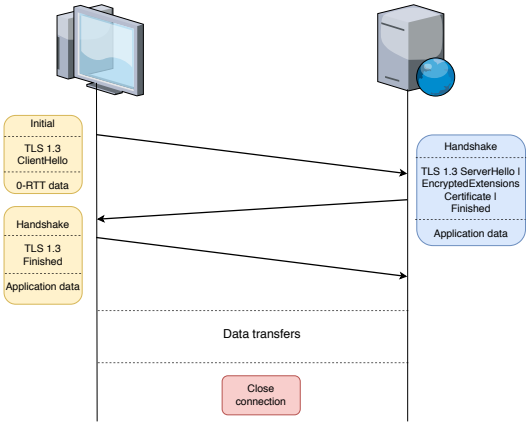



*Figure 3.2: 0-RTT handshake of QUIC. Application data is sent along with the Client Hello.*

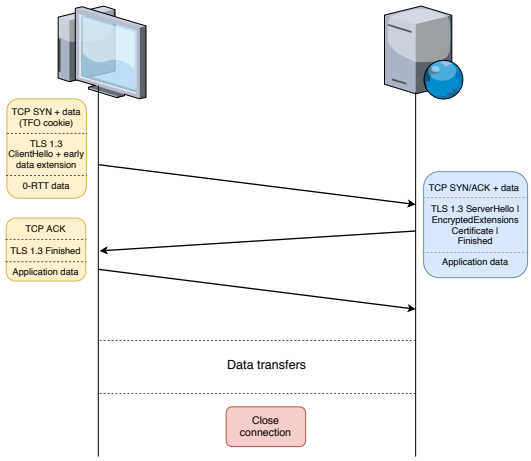



*Figure 3.3: 0-RTT by using TCP with TCP Fast Open and TLS/1.3. This is only possible with resumption, the initial connection still needs at least two RTT (one RTT from TCP and one RTT from TLS/1.3)*

A possible replay attack, which is also illustrated in figure 3.4, could be the following: imagine a client that already contacted a webserver once before and has kept the session data.

1. Client creates a Client Hello packet and makes a POST request to transfer €500 using 0-RTT because it uses session resumption.
2. An on-path attacker notices the Client Hello packet and the 0-RTT data and saves it for later use.
3. The webserver receives the Client Hello packet and the 0-RTT data. It next passes the Client Hello data to its TLS stack, generates the necessary keys and reads the 0-RTT data and processes the 0-RTT data, thus transferring the money from the one bank account to the other.
4. Next it generates the Server Hello, together with the response to the 0-RTT data.
5. When the client finally has received all the data it needed from the webserver, it closes the connection.
6. When the on-path attacker is sure that the connection between the client and the webserver has been shutdown, it starts sending the Client Hello and 0-RTT data, containing the POST request to transfer the money, again to the webserver.

When the data that is sent from the client to the server is idempotent, the client can choose to close the connection and make use of the 0-RTT functionality to request data or keep the connection alive and use the warmed up connection. Both approaches have their advantages and disadvantages. When the client uses the 0-RTT functionality to request data and close the connection, he does not need to keep the connection open and thus resources are freed after receiving the data from the server. When the client needs another resource from the server, it could use the previous session again to reopen the connection and request data using 0-RTT again. However, this approach also has its disadvantages, when the connection needs to reopen every time the client needs a resource, the connection does not know how many bandwidth there is available on the network. This means that the connection needs to go through the slow-start phase from congestion control, which is explained in more detail in section 3.7.2 for QUIC. Keeping the connection alive benefits from the fact that the connection is already warmed-up to use the available bandwidth for new requests for resources. This is at the cost of keeping the resources for the connection reserved at both the client and the server.

With a server that is responding to 0-RTT requests, this introduces another possible attack, namely an amplification attack. This amplification attack is illustrated in figure 3.5.
The amplification attack could go as follows:

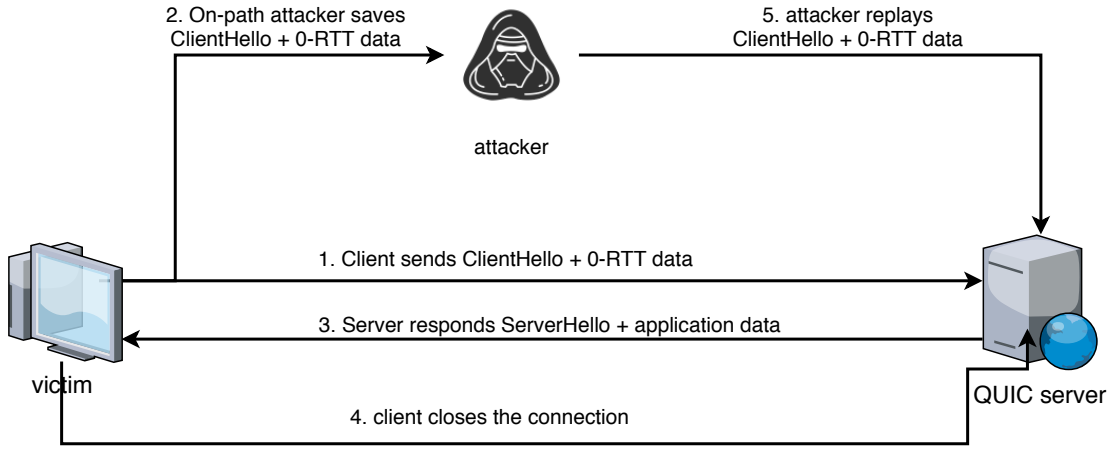



*Figure 3.4: Possible replay attack with 0-RTT. The attacker copies the Client Hello and the 0-RTT. After the victim closes the connection with the server, the attacker starts sending the copied data to the server.*

1. Attacker creates a Client Hello packet and requests a large file using 0-RTT because it uses session resumption. He also sets the source IP address of the packet to the one of the victim.
2. Server responds to the Client Hello packet with a Handshake packet containing the Server Hello. It also starts responding to the 0-RTT request which was made by the attacker.
3. Instead of the attacker receiving all this data, the network of the victim gets overwhelmed with the data that is sent by the server.

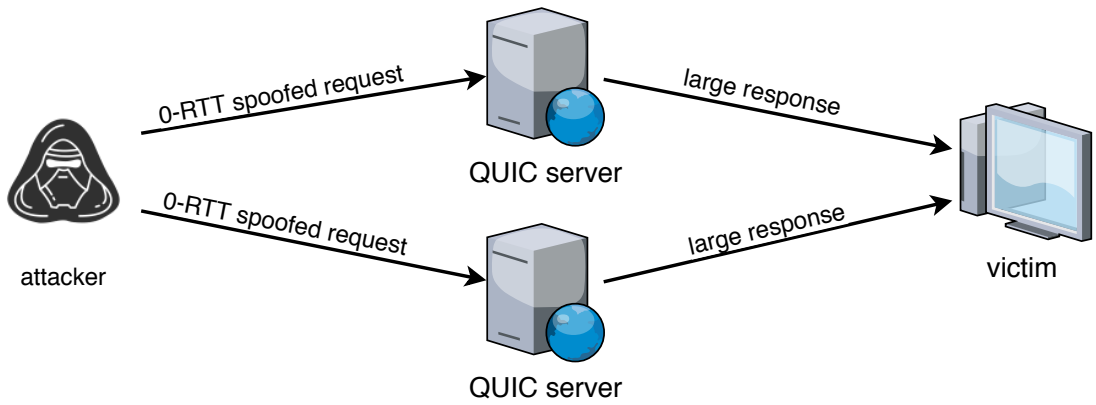



*Figure 3.5: Possible amplification attack with 0-RTT. The attacker creates a 0-RTT request that requests a large resource with a session that was made from a previous connection. When the server receives the request, it starts sending the data from the resource to the victim.*

As mentioned in section 3.3, during the handshake a path validation must be done to prevent an amplification attack. However, because with 0-RTT requests, this path validation method cannot be used because it would diminish the use of 0-RTT. To prevent an amplification attack, the server adds an address validation token in the new session ticket, which was obtained in the initial connection and that was used to resume the connection. When a server receives a session ticket, it can check the address validation token to make sure the path is valid. Therefore, the token must be a hard to guess token and its integrity protected to prevent the usage of false tokens.

With TCP, it is also possible to obtain 0-RTT. This can be done by combining TCP with TLS/1.3 and TCP Fast Open [51], which is illustrated in figure 3.3. TCP Fast Open makes it possible to send

data in the SYN packet during the handshake. Next, the server processes the SYN packet, sends the SYN/ACK packet together with the response to the request which was sent along with the SYN packet. This can only be done when the client already connected with the server prior to using the TCP Fast Open. During the first connection, the client adds the option of TCP Fast Open to let the server know that it supports TCP Fast Open. When the server also supports TCP Fast Open, it adds a cookie, which is constructed by using the clients IP address. The second time that the client wants to connect to the server, it adds the cookie to its SYN packet. The server verifies that the cookie is valid and when it is valid, it processes the data and returns the response to the client. To enable 0-RTT for TLS/1.3, the client has to use the session ticket which was provided by the server in the first connection. Thus, to use 0-RTT in TCP, the client needs to keep the cookie for TCP Fast Open and the session for TLS/1.3.

As was already mentioned in 2.2.5, TCP Fast Open also has seen some usage issues, even when both client and server support it, since its deployment, as it is a TCP option. Paasch et al. [49] tested this for TCP Fast Open and have come to the conclusion that, a small fraction of devices tend to suppress TCP Fast Open by disabling them in the SYN packet, when the device does not know TCP Fast Open or when the packet contains data in its syn packet,they drop the packet or even drop the entire connection. Another limitation of TCP Fast Open, is the fact that it can only be added to a single packet, which is the SYN packet. Thus, this limits the data that can be added to the MSS that was advertised in the previous connection of the server. If this value was not kept, the client needs to use default MSS for IPv4, which is 536 bytes, or the default value for IPv6, which is 1240 bytes. For these reason, it was one of the goals to make it possible for QUIC to perform 0-RTT before the standardization of the protocol. With QUIC, it is possible to send multiple QUIC packets containing 0-RTT data, as long as the receive window and the network allows it.

#### 0-RTT in Quicker

To provide a way to perform 0-RTT, Quicker added an extra parameter to the connect function to pass possible 0-RTT data. Even though, it is possible to request multiple resources using 0-RTT, Quicker is currently limited by only one. Furthermore, it is only possible to perform HTTP/0.9 GET requests. So normally, these requests are always idempotent. For this rason, Quicker does not have a way to check if a request is idempotent or not. The client needs to make sure of this.
To make 0-RTT possible, the NewSessionTicket is necessary. To be safe from amplification attacks, the NewSessionTicket needs to contain a token, which the server checks to see if the client is the owner of the NewSessionTicket. Quicker does not implement this, because this functionality has been changed since draft 13 and the way to provide such a token was discovered by us when draft 13 was already released. Because it would only be useful for draft 12 and not for later draft versions, it was not added to Quicker.

## 3.5 Multiplexing with Streams

As done by HTTP/2, QUIC adds the use of streams. With these streams, it is possible to request multiple resources concurrently. In contrast to TCP, which suffers from HOL blocking, QUIC can use these streams to overcome this issue, that was explained in section 2.4.1. Each stream has a stream identifier to identify the stream. This stream ID is an unsigned 62-bit integer where the two least significant bits are used to identify the type of stream. A stream can be unidirectional or bidirectional. The second least significant bit identifies this type. When the bit is set, it is a unidirectional stream, if not it is a bidirectional stream. The least significant bit is to identify which endpoint initiated the stream. When this bit is set, the server initiated the stream, otherwise, it is the client. By using fixed bits for identifying these flags, the need to send extra meta data about the stream becomes unnecessary.

If these bits were not added, streams would be always bidirectional and an endpoint could choose what stream ID it would use. However, there could be a race condition that both endpoints start requesting data on the same stream ID. This would make it harder to start a request without the other endpoint interfering by using the same stream ID. A way to make these dynamic without having any race conditions is to agree what streams are unidirectional and bidirectional and what streams

are client initiated and server initiated. This could be absolute or by agreeing what bits are for which endpoint and direction. However, to make things not more complex than it already is, the bits are fixed in QUIC .

When a stream is created, resources can be exchanged between both endpoints. The data from these streams are exchanged in stream frames. Each stream frame has a stream identifier, together with some flags to let the other endpoint know what the size is of the frame and if it is the last frame of the stream. To make sure that data is received by the receiver in the same order as it was sent, each stream frame contains an offset that is used to sort the data at the receiving side of the stream. This offset is also used for stream flow control, which is explained in detail in 3.6

#### Stream management in Quicker

Quicker manages different streams in a connection to provide multiplexing. How this works is, when a request (e.g., HTTP request) is made, the data which needs to be sent to the receiving endpoint is added to the Stream class, which keeps it in a buffer until the data is gathered for further processing. When Quicker starts building packets, the data which is contained in all streams are gathered and divided in STREAM frames, which are then divided in QUIC packets and transmitted to the receiver. Currently, Quicker does not provide any way to prioritize streams over other streams. Also internally, it iterates over the streams which are still open and divides the data that has been put into the stream over stream frames. This could be viewed as a first come, first serve at the level of streams, because streams which are created first, are handled first.

When the receiver receives STREAM frames, the data is emitted to the application that is using Quicker. When an out-of-order STREAM frame arrives, the STREAM frame, together with the offset of the data that is contained in the STREAM frame are buffered in the Stream class until all prior data is received. The Stream class also keeps track of the last received offset of a stream to make the decision to buffer incoming out-of-order STREAM frames, emit the data to the application when the frame is in-order or to drop the frame when it was already received, which is the case when the offset is larger than the offset that is given in the STREAM frame.

## 3.6 Flow Control

QUIC implements flow control at the level of the connection, stream and stream ID. Having flow control on different levels has some advantages. In TCP, there only exists flow control on the level of the connection. With connection flow control, TCP makes sure that the receiving side of the connection does not get overwhelmed by the sending side of the connection as stated in 2.2.3. However, QUIC also works with multiple streams. To avoid one stream taking over the connection, flow control is also added to the level of streams. This means that a sender cannot send more data on a stream than there is available by the flow control at both stream and connection level. Note that both connection and stream level flow control, only applies to stream frames. The control frames (e.g., Max Data frame, Ack frame) are not affected by flow control.
This can be illustrated by an example. Let's say that a student is watching a video of a lecture. At the same time, he is downloading the resources that are accompanying the video. Both the resources and the video are transmitted over the same QUIC connection. Because the video needs to play real-time, the stream that is carrying data for the video is prioritized over the resources. When the student decides to pause the video to wait until the resources are downloaded for the lecture, the client could throttle the stream that is used to transmit the video to avoid having to buffer a large amount of data. When the resources for the lecture are downloaded, the student unpauses the video of the lecture, which results in the client sending MAX_STREAM_DATA frame to start receiving data again for the video.

In addition, QUIC also implements flow control on the level of stream ID's. This is to avoid that either endpoint opens a lot of streams without checking if the other endpoint can even handle this amount of streams. This is to avoid having situations where the other recipient opens a tremendous amount of streams. When a client opens 5000 streams to request various resources and the maximum

allowed data on the connection cannot exceed 50.000 bytes, the server can only transmit 10 bytes on each stream if it wants to divide the available memory equally. Additionally, the server needs to keep a state for each stream, which uses memory that it could be using for other purposes. For this reason, a recipient can set the maximum amount of streams it wants to keep open. As previously mentioned in section 3.5, there are multiple types of stream ID's. When an endpoint receives a MAX_STREAM_ID frame, it must first check if the given stream ID is a unidirectional stream ID or a bidirectional frame. An endpoint must also check if the given stream ID is larger than the current maximum stream ID, if not the frame must be ignored. Neither connection, stream and stream ID flow control can decrease its maximum given value by a new frame. When these checks are done, the maximum stream ID that the receiver of the frame can use is adjusted.

**Flow control in Quicker**

Quicker implements flow control for both streams and connections as for stream IDs. To implement flow control for streams and connections, both keep variables of the current local and remote offset, which is actually the amount of data that is already sent or received, the maximum local and remote data that is allowed and the amount of buffer space that is available. A stream keeps track of its own buffer space, while the connection keeps track of the used space of all streams. When Quicker notices that a stream's or a connection's offset is almost exceeding, which is currently calculated by adding 10% from the maximum data to the current offset and checking if this value exceeds the maximum data, the maximum amount of data that is allowed is recalculated by incrementing the current offset by the amount of buffer space that is available. The value of 10% is chosen arbitrarily. This results in the fact that the amount of data that can be received by the receiver is not larger than the available buffer space, for both streams and connections.
To make sure that the sender does not exceed the remote receive buffer, it checks the remote offset prior to taking the data off the stream to put it into frames. When it notices that it cannot send any STREAM frames containing data, it keeps the data in the stream and the sender sends either a STREAM_BLOCKED frame or a BLOCKED frame when respectively the stream is blocked or the connection is blocked.

Flow control on the level of stream IDs is not fully supported in Quicker. When it notices that the remote endpoint is running out of available streams, Quicker sends a MAX_STREAM_ID frame without checking how many streams are still open. Thus, it simply makes sure that the remote endpoint does not run out of available streams to use. In the future, this could be modified by checking at the stream manager, how many streams are open and by setting a maximum amount of streams that can be open. When the other endpoint runs out of streams, the amount of streams that can be opened is calculated and a MAX_STREAM_ID frame is transmitted if it is possible.

## 3.7 Recovery

QUIC implements recovery [39] by adding loss detection and congestion control with the knowledge obtained from TCP and by taking into account the differences between TCP and QUIC.
Firstly, as previously mentioned in section 3.2, packet numbers are monotonically increasing, which solves the retransmission ambiguity, as was shortly mentioned in section 3.2.

Secondly, TCP packets that are transmitted, need to be delivered at the receiver, no matter what the content of these packets are. QUIC divides packets into packets that are retransmittable and ones that are not. Retransmittable packets are packets that contain frames which are retransmittable. This is because the packets itself are not retransmitted, rather the frames inside a packet are retransmitted, if these frames are retransmittable. STREAM frames are an example of frame which are retransmittable, they need to be delivered because the application needs this data. PADDING frames are frames which are not retransmittable. This is because PADDING frames are used to make a certain packet bigger (e.g., Client Hello packet) if necessary, however the frame itself does not contain any information. For this reason, this type of frame is not retransmittable. Another example of a frame that is not retransmittable, is the ACK frame. This is because, whether or not an ACK frame is lost, the next ACK frame shows to same or even updated values of the previous ACK frame. When the first ACK frame acks a packet with packet number one and two, the second ACK frame

acks the same packets, plus additional packets that may have arrived in the meantime, because the first ACK frame has not been acked (because it went lost).

Thirdly, reneging, which is only possible when the SACK option is available and the receiver selectively acks data and later discards it from its receive buffer before delivering the data to the application-layer, is not allowed in QUIC. Reneging is valid in TCP and is done when the receiver has limited memory and discards the packet to free some memory. However, because it is allowed, senders need to keep the acknowledged out-of-order packets in their send buffer until all previous packets are acked. By not allowing reneging, the sender would not need to keep packets in their buffer and thus, senders can use this free memory for new packets. Ekiz et al. [34] created a model to detect reneging and analyzed TCP traces from different domains to check if reneging still occurs. The result was that in only roughly 0.05% of the traces reneging occurred and thus reneging does only rarely occur these days. They suggest that reneging should not be allowed in newer reliable transport protocols and when a situation occurs that the receiver does not have enough memory, it should reset the connection instead of reneging some packets.

Lastly, ACK frames encode the delay between when the packet was received by the receiver and when the corresponding ACK frame has been sent. This is to calculate a more accurate RTT by taking into account the delay which occurred by the receiver when it was processing the packet, which is used for loss detection (section 3.7.1) This is because QUIC implementations do not need to respond immediately with an ACK frame. They could wait until a couple of packets can be acknowledged. The value of the delay is based on the last received packet that is being acknowledged by the ACK frame.

### 3.7.1 Loss Detection

QUIC adds loss detection to make sure it is reliable and all QUIC frames are delivered, by identifying which QUIC packets are lost and retransmit the frames within these QUIC packets in new QUIC packets, as was mentioned in 3.7. To detect a loss, QUIC uses ACKs and timeouts.

To implement ACK based loss detection, QUIC uses fast-retransmit that is used to retransmit the data of a packet when a fixed amount of packets are acknowledged that were transmitted after the packet, which is default set to three, early-retransmit, SACK (section 2.2.2) and forward acknowledgement (FACK). Early-retransmit is used when fast-retransmit cannot be used. When the amount of outstanding packets is less than the value of the packets that needs to be acknowledged for fast-retransmit. This happens when the last packet that was transmitted was acknowledged by the receiver. All the packets that were transmitted prior to the last packets and are still unacknowledged, are retransmitted. FACK [46] uses additional information, provided by SACK, to measure the total bytes outstanding. This helps to discover multiple losses faster than by waiting for three duplicate ACKs to occur. By using these algorithms,a lost packet can be discovered faster than simply waiting for three duplicate ACKs to occur.

As was already mentioned, the ack delay is included in the ACK frame to get an estimation of the RTT. This is because timer based loss detection heavily relies on a good estimated RTT to perform better. When the RTT is to low, retransmissions occur to fast and when the RTT is calculated to high, lost packets are detected much later.
To implement timer based loss detection, QUIC uses three alarms, which are for handshake retransmissions, (tail loss probe) TLP and (retransmission timeouts) RTO. Firstly, handshake retransmissions are added because various reasons that [14] could happen when the first QUIC packet, containing the Client Hello is lost. ACK based loss detection cannot be used here due the fact that this is the first packet. The packet could be lost because of the network (e.g., Congestion on the path, malfunction of a device on the network ). It could be because the path of the network does not support the size of the packet due the fact that the MTU of the path does not support packets of that size, which for Client Hello packet is 1200. In this case, QUIC cannot be used because it needs a minimum MTU of size 1200. Client Hello packets smaller than 1200 are not allowed to be accepted by a QUIC server. A Third reason for the packet loss is because a middlebox did not like the packet and dropped it. To solve these various reasons, a QUIC packet is retransmitted after two times of the initial RTT, which

is default 100ms. When the packet is lost again, the previous timer value is doubled until the idle timeout occurs, which default is 30 seconds. When this timeout occurs, the QUIC connection cannot be established over the used network path.

The second timer, which is TLP, is used to detect a packet that is lost at the tail, which is one of the last packets that were sent. Normally, ACK based loss detection can identify a packet that was lost, however because these packets are at the tail, recovery using duplicate ACKs is not possible, which is when three duplicate ACKs are received to indicate that a packet was lost and the data needs to be retransmitted. For this reason, TLP is proposed for TCP and is also used for QUIC.
The time of TLP is set when the last retransmittable packet has been sent prior to being idle for some unknown time. When this alarm fires, a TLP packet is sent to try to force the receiver to send an ACK. When there is new data available, a TLP packet contains this new data, otherwise it should just retransmit data that has not been acknowledged yet. When the TLP alarm is fired for the first time, a second TLP alarm is set prior to setting the RTO as is done in TCP. This is done to reduce latency because the RTO alarm has a longer time than a TLP alarm. Another variation of the QUIC version of TLP, is that it calculates the MaxAckDelay dynamically. In TCP, this is a constant which is used for all connections, QUIC incorporates an ACK Delay field in its ACK frame. With this field, the MaxAckDelay is reconsidered on each ACK frame that is received. By making this dynamically, each time that is set for the TLP and RTO has a realistic view of the possible delay that the other endpoint has while processing the packet and sending the ACK frame.

The third alarm that is used, which is also used in TCP, is the RTO alarm. This alarm is set when the TLP alarm has been fired twice. When the RTO alarm fires, two packets are sent to make sure that an ACK could be retrieved, which contain new data or retransmitted data as is the case of a TLP packet, and a new RTO alarm is set with the RTO period doubled as the previous RTO alarm. This is because the probability that the chance that two packets are lost is smaller than when only one packet is sent. As the TLP period, the RTO alarm also adds the MaxAckDelay to incorporate a better estimation of the delay that was introduced by the other endpoint, as is the case of the TLP packet. QUIC RTO differs from the TCP version in that the firing of the RTO alarm does not change the congestion window because the RTO is not considered an indication of packet loss, as it could also mean that the RTT has changed. On top of that, a packet that is sent because of the RTO alarm cannot be blocked by the senders congestion control. However, it does count for the bytes in flight of the senders congestion control because this packet adds an additional load to the used network without an occurrence of packet loss.

### 3.7.2 Congestion Control

QUIC adds congestion control to detect congestion, which happens when a network device cannot handle all of the incoming traffic, resulting in that device starting to drop packets on the path between both endpoints. By detecting congestion, the sender can change the pace it is sending. As is the case in TCP, QUIC also keeps a congestion window, which is the amount of data that is in flight which is not acknowledged yet. The congestion control algorithm described in the QUIC draft is based on TCP NewReno, which was explained in section 2.2.4 as an example of a congestion avoidance algorithm in TCP.

QUIC is implemented in user space, which makes it easier to switch between different congestion control algorithms. This benefits the endpoint which uses QUIC to adjust the congestion control algorithm to the situation. With TCP, the congestion control algorithm is provided in kernel space. This makes it hard to make changes to the current network situations, because each TCP congestion control algorithm has an ideal situation. When an endpoint notices that the congestion control algorithm that is being used is not ideal on its current network, it could just change the algorithm with one that is more ideal on its network. As of today, Google QUIC implements both TCP CUBIC and TCP BBR. Thus, a user of Google QUIC could just switch the algorithm.

Even though the recovery draft [39] explains how to implement TCP NewReno in QUIC, it is not mandatory to use TCP NewReno as the congestion control algorithm [21]. This was added as an

example and as a default. Implementors can use whatever congestion control algorithm they see fit to implement in QUIC

**Recovery in Quicker**

To manage all the ACKs that needs to be transmitted, a separate AckHandler class was created. This handler keeps information of received packets, which is the following: the packet number, the time when the packet was received and if it only carried acknowledgements. These need to be kept until a packet has been acked by the other endpoint which contained an ACK frame where these packets of the other endpoints were acked. This is because ACK frames are not retransmittable. So if this information was removed immediately after the creation of the ACK frame and the packet containing the ACK frame was lost, the other endpoint would think that the packets it has sent, were lost. To determine the ack blocks, the packet numbers are first sorted when the handler wants to create an ACK frame. Next, for each gap between two packet numbers, a new ack block is created. When all packet numbers have been considered, the ACK frame is constructed and queued for transmission. The creation of an ACK frame can be triggered in two ways. Firstly, when a batch of packets are created for transmission, it first checks if one of the queued frames is already an ACK frame. If not, the getAckFrame method is called to check if it can add an ACK frame. When there are still packets that are not acked, an ACK frame is added to the queue and transmitted with the batch of packets. The second way of triggering the creation of an ACK frame is with a timeout. When a packet has been received and no packet needs to be sent, the AckHandler constructs the ACK frame and adds it to the queue of frames of the connection, after the timeout went off.

Quicker implements loss detection with the given pseudo code from the recovery draft [39]. However, a minor difference between the draft and Quicker is that when the TLP or RTO alarm fires, it just retransmits the data of one packet, or two in the case of RTO, which was sent before. This decision was made to reduce the time that the data of the packet needs to be retransmitted. Even though the RTO is not considered to be a strong enough indication of packet loss, by retransmitting data, we avoid the situation where the reason was packet loss. We do have seen other QUIC implementations (e.g., quicly) to transmit new data instead of retransmitting data.

Congestion control is also implemented in Quicker, using the pseudo code given from the recovery draft [39]. On top of that, we added a queue for packets and a sendPackets method that is triggered when a packet is acked, lost or queued. The method is called when a packet is queued for the obvious reason to check if it can transmit the packet immediately or keep it in the queue because the congestion window does not allow. When a packet is acked, the method is called to check if there are any packets in the queue as the congestion window is larger because of the acked packet. Finally, the queue is also checked when a packet is lost because the bytes that are in flight, which means the amount of bytes that is outstanding, is decreased because a packet is lost.

## 3.8 Connection Migration

Unlike TCP, QUIC allows an endpoint to migrate to a new location (e.g., IP address or port changes). With TCP, when the four-tuple of an endpoint changes the connection drops because the connection is identified by the four-tuple. One of these value changes when an endpoint changes its network and thus, TCP has no means to indicate that the endpoint has a new location. QUIC uses the Connection ID (section 3.1) to identify a connection. This value does not change when an endpoint changes one of the values of the four-tuple and thus, the connection can still be identified by both endpoints.

A change of the four-tuple could occur intentionally (e.g., moved from a Cellular network to a WiFi network) or unintentionally (e.g., NAT rebinding). When this occurs, the receiving endpoint would receive a packet from an unknown source with a Connection ID it recognizes and a packet it can decrypt. However, the packet could be sent by a malicious attacker. This could be a peer or an on-path attacker that is performing address spoofing. A peer could spoof the address to perform a Denial of Service (DoS) to its victim. This is done by first finishing the handshake, then requesting a large file and immediately after, sent a packet (e.g., containing an ACK frame) with the spoofed

address. The receiving peer would think the sending peer has migrated to another location and sent the large file to the new address. The victims network could possibly not handle the amount of traffic that it is suddenly receiving and results in a DoS.

This attack is mitigated in two ways. The first and most important is, when a endpoint receives a packet from an unknown source, it must respond with a PATH_CHALLENGE frame, which was previously mentioned in section 3.3. When and endpoint receives this type of frame, it must respond to it by sending a packet containing a PATH_RESPONSE frame. In the case of the attack, the endpoint would not receive the PATH_CHALLENGE frame, which is an indication that the new address is not owned by the endpoint from the previous address. An endpoint could go back to the previous address for subsequent packets or abort the connection. The second way which mitigates the given attack is by limiting the rate that packets are sent. This is done to avoid immediate congestion on the new path. Because the endpoint is on a different network, the resources available on the new network could be less than the previous one, unless it is sure that the new path is the same as the old one (e.g., when client's port number has changed, it could be the cause of NAT rebinding). However, until the new path is validated by the first mitigation, the rate at which packets are sent, must be equal to the minimum congestion window size.

The second type of attack is when an on-path attacker copies a packet on the network, spoofs the address and sends it to the receiving peer. When the spoofed packet arrives before the legitimate one, the receiving peer would get an indication that the sending peer has migrated to a new network and the legitimate packet is dropped because it already received a packet with the given packet number. However, because the attacker does not have the cryptographic keys, it cannot respond to the path validation. The receiving peer could go back to the previous address or when this is not possible anymore, abort the connection silently and discard all state of the connection. Implementations should keep the previous address of the peer and not transmit any data to the new address until the new address has been validated. This is to avoid a possible amplification attack on the new address. When the peer had to close the connection and discard all state, it can respond to new incoming packets with a stateless reset that uses a stateless reset token of 16 byte, which is provided during the handshake in the transport parameters. With a stateless reset, it is possible to communicate to the other endpoint that an error occurred and the peer does not have any state anymore of the connection. This is why the stateless reset token must be a calculated value to be able to generate it based on a field it can use, such as the four-tuple or the Connection ID in the header of the received packet.

When an endpoint migrates to an other connection, it is still vulnerable for linkability issues. If an endpoint uses the same Connection ID, an on-path observer could link this connection with the connection from the other source and know that the endpoint just migrated to an other network. To mitigate this issue, an endpoint that supports that the other endpoint can migrate, must issue a Connection ID in a NEW_CONNECTION_ID frame. This is an indication to the other endpoint that it can use the new Connection ID when it is migrating to a new network, thus when the endpoint migrates to an other network, it uses the new Connection ID, which the other endpoint recognizes.However, an on-path observer would think this is a new connection or it recognizes that it is a migrated connection, however it cannot verify what the previous connection was. The on-path observer cannot see the NEW_CONNECTION_ID frame, as it is transmitted in a 1-RTT protected packet that cannot be viewed by the on-path attacker.

To following example makes this more clear:

1. The client finishes the handshake with the server. The server supports that the client migrates to an other network, so it sends NEW_CONNECTION_ID frames to present the client new Connection IDs for possible migrations.

2. An on-path observer notices traffic going out from the client and going to the server, so one half of the connection is exposed to the observer.

3. The client migrates to a different network, so it takes one of the given Connection IDs that were provided by the server for the subsequent packets.

4. The on-path observer does not see any traffic anymore with the previous Connection ID going from the client to the server. Because Connection ID and IP address (and possibly port) has changed, the observer cannot link traffic prior to the migration with the new traffic.

**Connection migration in Quicker**

Quicker currently does not implement connection migration. This is due to the fact that connection migration still changes frequently in the current drafts. Also, the current use cases are not clear yet on when an implementation needs to decide to perform connection migration or when the Connection ID needs to change. With our current view on connection migration, we assume that the endpoint must check the IP address and port that is used to change the Connection ID. This means that, when an endpoint does a connection migration, the other endpoint needs to associate the Connection ID with the four-tuple that the first endpoint uses. The challenge is, knowing when to change the destination Connection ID to use. It is easy to check if the local IP address or port has been changed to change the Connection ID accordingly. However, with external changes (e.g., public IP address changes, NAT rebinding) it is not clear how to detect this and change the Connection ID to prevent linkability. Thus, the endpoint only knows for sure that it needs to change its Connection ID when it is changed locally. If NAT rebinding has occured, the client does not have any control over this, nor it has a way of identifying this.

Additionally, parts of connection migration are not included in the first version of QUIC. Multipath QUIC [10] and simultaneous connection migrations [15] are examples of migration that are left out for the next version of QUIC due to its complexity and to make sure that the core of the first version of QUIC is constructed to make additional features in the next versions easy to incorporate. Multipath QUIC is the possibility to transmit and receive data over multiple network paths and simultaneous connection migration is the ability to let both endpoints migrate at the same time, where one of the endpoints initiates this process by using a different address (e.g., other IP or port) to transmit data to a different address from the other endpoint, which has communicated this address to the initiator of the simultaneous connection migration.

## 3.9 HTTP/QUIC

HTTP/QUIC [28] is the mapping of HTTP over the QUIC transport protocol. Because QUIC implements features which are also implemented in HTTP/2, it would not be beneficial to simply run HTTP/2 over QUIC as it would run over TCP. For this reason, HTTP/QUIC was created. HTTP/QUIC uses the features, such as stream multiplexing, from QUIC instead of implementing it over again, which is done in HTTP/2. As such, the HTTP/QUIC specification is a slimmed down version of HTTP/2 that describes how to map semantics of HTTP over QUIC. Because HTTP/QUIC uses QUIC, it does not suffer from the limitations (Section 2.2.5) that HTTP/2 has by using TCP.

Still, differences exist between HTTP/2 and HTTP/QUIC. As previously mentioned, QUIC implements features from HTTP/2. Therefore, frames that were used for these features (e.g., stream multiplexing) are not needed anymore, as such they were removed from HTTP/QUIC.
Another difference is that HTTP/QUIC uses QPACK [42], which is a variation of HPACK [50], which is the header compression format designed for HTTP/2. QPACK is mostly the same as HPACK, however if HPACK would be used together with HTTP/QUIC, inconsistencies would occur between the headers from the sending endpoint with the receiving endpoint. This is due the fact that HTTP/2 uses TCP as its transport protocol. Because everything that is encoded and transmitted with TCP arrives in the exact same order, HPACK could just build its header tables with the knowledge that whatever arrives first, was transmitted first by the sender. However, as QUIC implements in-order delivery in a single stream, it does not provide in-order delivery between streams. Because of this, headers that were sent on different streams, could arrive out of order, which results in a state where the sending endpoint has different headers in his table than the receiving endpoint. To solve this issue, QPACK was created.

Each header block that is sent from the encoder to the decoder, contains a Largest Reference value. This contains the largest index in the dynamic table, which is the table that contains headers that

are not used by default (e.g., custom headers from applications). When a header block is received, which has a larger reference value than is present in the current dynamic table of the decoder, the header block is buffered and the stream on which the block has been sent, is marked as blocked, until the largest index is greater than or equal the Largest Reference value from the header block. This is done to make sure that headers that are referenced by index in the header block, are present in the dynamic table. If this check did not exist, a header block containing a header by reference that was received before the actual header block that maps the header to the index, an error would occur. Header blocks that can be processed without the need to wait for a previous header block, can set the Largest Reference value to zero. This means that the header block can be processed immediately. This is the case when a header block contains only new mappings and thus, does not contain any references to the dynamic table.

Even though, this results in the fact that a stream can block another stream when a stream has transmitted a header block before the second stream and the header block of the second stream arrives before the first stream. It makes sure that the headers that are used on any given time by the receiver, are the same when the sender has sent them, along with the corresponding data. This means that they have introduced HOL blocking by using this approach. This is because one stream needs to wait until the headers of the other streams are obtained.

To make sure that the encoder and decoder are kept in sync, the decoder emits events to the encoder when a particular header block has been processed, when a stream has been abandoned and when new dynamic table entries are received. This is done to make sure that the encoder does not send unnecessary header blocks and can check if sending a particular header block can block the stream at the decoder, thus this is an additional way to check if a header block would be blocked.

#### HTTP/QUIC in Quicker

At the time of writing, Quicker has no support for HTTP/QUIC. This is because the development of Quicker was done in parallel with others that are implementing QUIC. To test interoperability, which is one of the tests in section 4, Quicker followed the newest drafts up to draft 12. Because other implementations focused on the transport draft of QUIC, HTTP/QUIC has not been implemented. To test the different servers, HTTP/0.9 was used to focus more on the transport draft. Also, since draft 10, QPACK was added to QUIC which does not have a full implementation yet. Due to the complexity and the short amount of time, this was not doable for this thesis and as such is ideal for future work.

## 3.10 Forward Error Correction

Forward error correction[57] (FEC) was added in the early days of Google QUIC. It is a way to recover a single packet from a group of packets based on the information provided by the rest of the group and a FEC packet. FEC in QUIC is based on XOR operations. For every x amount of packets that are sent, a FEC packet is sent along with them. This FEC packet contains the result of the XOR operation on the x amount of previous packets that are sent. When a single packet is lost, this packet can be recovered by using the FEC packet with the x - 1 packets that did arrive at the receiver. The group was identified by a field in the header of a QUIC packet.

One of the main advantages of FEC was that it could recover a packet without retransmitting any data. The operation that is used, namely XOR, also has a low computational overhead. Before the launch of Google QUIC, Google performed experiments with FEC and Chrome. Chrome sent 20 packets in burst and FEC was able to recover the packet losses in 65% of the time. However after the launch, new experiments resulted in only 28% of the cases that FEC was useful. This is due the fact that FEC was not useful in the situations where no packet was lost or where more than one packet was lost. Studies have shown that when FEC is enabled, QUIC happens to be less efficient. Carlucci et al.[30] compared TCP with Google QUIC with and without FEC. The parameters that are compared are loss ratio and channel utilization. They found that, when FEC is enabled, it utilizes 33% of the available bandwidth in the Google QUIC version they were testing. They also found out that when a congestion loss occurs, which is a loss that occurs when there is congestion on the network between

the two endpoints, more than one packet is lost which means that FEC cannot be used. On top of that, they also found out that when FEC is enabled, the loss is even greater. This is because, when a packet is lost, this is most of the time due the fact that congestion occurred on the network. However, when a receiver can reconstruct the packet, the congestion window is not altered and it keeps increasing until more packets are lost. At the moment of writing, FEC was not included in the current IETF QUIC version. According to the QUIC charter [10], one of the next versions of QUIC will include FEC, probably with better FEC schemes than the one described here.

## 3.11 Packet flow in Quicker

In this section, we are going to explain how QUIC packets are handled when they arrive at the server or the client in Quicker. We discuss both how the QUIC packet is parsed and how it is processed later. Next, we also explain how and when new frames are created and how these are transmitted to the recipient.

#### Incoming QUIC packets

There are two situations when packets are handled in Quicker. When there is an incoming packet and when there is an outgoing packet. Because Quicker can be used as a library, both client and server have similar packet flows, they only differ how TLS data is handled and also that a client can initiate the connection and the server cannot, or it would act as a client for another server. In the following section, the situations are explained from the point of view of a server, unless otherwise stated, the client behaves the same.

When a UDP datagram arrives at the server, the headers of all the QUIC packets are parsed. A UDP datagram can contain multiple QUIC packets. This is what is called coalescing of packets. QUIC packets which have a Long header, contain the field Length in the header. This field contains the length of the encrypted packet number + the length of the payload, which was done because otherwise it was not possible to determine the entire length of coalesced packets, which are multiple QUIC packets in a single UDP datagram. This is due to the fact that packet numbers are encrypted using the length of the packet and to determine the length of the packet, the packet number needs to be decrypted. The Short header does not contain a Length field because when the handshake is completed, all frames can be placed into a single QUIC packet.

When all the headers are parsed and their corresponding encrypted packet number and encrypted payload are identified, the header handler is called. The header handler does tasks which need to be done, prior to decrypting and parsing the payload of the packet. First thing which need to be done, is to check the version of the header, in case it is a Long header, because a Short header does not contain a Version field. In case the server identifies that this is an unknown version, the server transmits a Version Negotiation packet, which contains all the QUIC versions that the server supports. When the packet contains a known version, the header handler starts to decrypt the packet number and sets the correct packet number in the header object.

Next, when the packet number is decrypted, the payload of the QUIC packet can be decrypted and eventually parsed by the packet parser. The payload of these packets contain frames, unless it was a Version Negotiation packet. Thus, when it is known what the type of packet the packet parser is processing, the frame handler is called to parse the frames inside the packet. After the packet and the frames are parsed, the parser first checks if the packet contains invalid frames. As an example, a flow control frame cannot be carried by a Handshake packet. When an invalid frame is detected, an error is raised and the connection is closed. Next, the packet handler is called. The packet handler calls the handle function of the identified packet type. Unless the packet is a Version Negotiation packet, which only the client can receive and results in checking if one of the given versions is supported by the client, if the version is not supported, the connection is closed. All the other packet types contain frames. To call the appropriate methods for the corresponding frames, the frame handler is called. The frame handler makes sure that the correct method is called for the given frame, which is also the end of handling incoming packets.

When an ACK frame is received, the loss detector is called. When loss detection receives the ACK frame from the frame handler, it starts by calculating which packet numbers were received by the other endpoint. With this, it can determine what packets went possibly lost and emits these lost packets. The AckHandler listens to the event of loss detection when a packet is acknowledged. The AckHandler needs to verify if a ACK frame was present in the acknowledged packet. When an ACK frame was present, it knows that the other recipient knows that we have received their packets. As such, we do not need to add the packet numbers in future ACK frames. Next the congestion controller both listens to the event of a lost packet and an acked packet from loss detection. When a packet is lost, it changes the bytes in flight and lowers its congestion window. When a packet is acknowledged, it can raise the congestion window and subtract the size of the packet from the bytes in flight variable. Finally, the connection listens to the event of packets that needs to be retransmitted, which is also emitted by loss detection. When this happens, the connection determines which frames need to be retransmitted and queues them for retransmission.

When one of the flow control frames are received, the connection or the stream that is associated by this frame is managed. As an example, when a MAX_STREAM_DATA frame has been received, the maximum allowed offset of the particular stream is raised to the value that is inside this frame. The stream is obtained by using the stream ID within the frame and by using this value to get the corresponding stream from the StreamManager. This StreamManager creates and keeps the streams that are created. It also emits an event when a new stream is created. This is done to create a QuicStream object for the library user or, when the stream ID is zero, to hook the stream to the HandshakeHandler. The connection objects also hooks to the events of each stream for flow control. When data is received by a stream, it emits the amount of bytes it has received to make sure the connection flow control also receives this amount.

#### Outgoing QUIC packets

However, by handling incoming packets, new events occur, in which case Quicker also needs to send QUIC packets. As previously mentioned,a QUIC packet contains one or more frames. These frames are created when the user of the library wants to sent application data, which result of the creation of one or more STREAM frames, or when the handling of incoming frames results in the creation of new frames (e.g., Incoming PATH_CHALLENGE frame results in an outgoing PATH_RESPONSE frame, Incoming packets result in ACK frames). These frames are buffered until all frames are handled from an incoming packet or when a timeout occurs, which is a timer that is set when the first frame is buffered. When one of these events occur, all frames are placed in the correct type of packet. When packets are constructed from the buffered frames, it also checks if an ACK frame can be added to a packet if this is needed, as was mentioned in section 3.7.

Next, we also iterate over all the opened streams from the StreamManager to check if there is any data available. If there is data available on a receive-only stream, the data is dropped, as these cannot transmit data. Next, we try to put data into STREAM frames, as it also needs to check if the receiver can receive these amounts of data. If the receive window is full, a STREAM_BLOCKED frame is added to the packet, if not, the STREAM frames with the corresponding data are added to the packets. We also check if there are any streams or the connection itself, are blocked or almost blocked. If this is the case, we check how many buffer space is available and add a MAX_(STREAM_)DATA frame to update the buffer space at the other endpoint.

When all frames are placed inside packets, they are moved to the queue of the congestion controller, which in turn sends the packets, when the congestion window allows it. When a packet is transmitted, it emits the packet sent event, where both the congestion controller itself, as the loss detection listens to. The congestion controller changes the amount of bytes in flight. Loss detection listens to this event to keep the packet in memory until it has been acknowledged or when it is verified to be lost.

## 3.12 NodeJS

Quicker is implemented in NodeJS. However, because QUIC uses TLS/1.3 and handles its own encryption of packets, extra functions were needed from OpenSSL. A second challenge that was encountered at the start of the development was that NodeJS used OpenSSL version 1.0.2 and TLS/1.3 is added to OpenSSL in version 1.1.1. This needed a couple of changes because OpenSSL changed their API since version 1.1.0. These changes were made with the help of existing pull requests on github that were not merged. After these changes were applied, NodeJS could be built with OpenSSL version 1.1.1.

The first exercise of changing the OpenSSL version using the pull requests was a start of understanding the basic workings of NodeJS. To get an even better understanding, we also ran the existing tests where a couple of them failed because of the new OpenSSL version which added TLS/1.3. NodeJS takes the highest possible TLS version, which results that TLS tests failed. By examining the tests that were failing and trying to make an effort to fix these tests, helped us gain an even better understanding of the existing codebase of NodeJS. This revealed how NodeJS uses OpenSSL in their flow to use TLS.

The reason why these tests failed was because TLS/1.3 only needs one RTT to perform the TLS handshake, while TLS/1.2 needs two. On top of that TLS/1.3 made a change to the exchange of the NewSessionTicket, which is used to perform a session resumption, as was mentioned in section 3.4. While in TLS/1.2, the NewSessionTicket is exchanged during the handshake, TLS/1.3 changed this by sending the NewSessionTicket after the TLS handshake. As the current tests of NodeJS check the session ticket immediately after the handshake is done, the client has not been able to receive the NewSessionTicket from the server and thus, the tests fail. Another example of a test which failed in NodeJS because of TLS/1.3 was a test that counted the amount of times the TLSWrap class was called, which is the class in NodeJS that manages the TLS communication. With one RTT less than TLS/1.2, the TLSWrap was not called as many times when using TLS/1.3.

With the knowledge received by looking at the issues of the tests, we created a wrapper for the functions in C++ that are needed for Quicker. These functions contain calls to OpenSSL are needed by Quicker to perform the handshake and derive the encryption keys to encrypt and decrypt QUIC packets. The functions of OpenSSL are almost directly mapped to the function of the wrapper class. This decision is made to have all the logic in TypeScript, instead of having some parts in C++ and some parts in TypeScript. The downside of this approach is the overhead. To handle incoming handshake data, first the C++ function is called to write the data to OpenSSL.

When the handler of the handshake data is the server, a call to read early data is used. This function is needed to be able to generate the keys to decrypt the 0-RTT QUIC packets and to be able to read the value set in the early_data extension, which needs to be set to 0xffffffff. Server must check this value, however this is not added yet to Quicker. Next, the endpoint call the readHandshake function to check if OpenSSL has data that needs to be transmitted to the other endpoint. Lastly, a call is made to the readSSL function of OpenSSL. This needs to be done to be able to process the NewSessionTicket, readHandshake does not work in this case as NewSessionTicket is processed after the handshake. So, instead of combining these four functions, we kept them separated to maintain the logic of the handshake in TypeScript.

# Chapter 4

# Evaluation

In this section, we are going to evaluate Quicker with other implementations with various different tests. These implementations include ngtcp2 and quant. As was discussed in section 3 the implementations, including Quicker, implemented draft 12. However, because draft 12 was not clear enough about the Length field in the header, it was made more clear in pull request (PR) 1389 [19]. This PR also made the change that the Length field includes the length of the packet number. With this in mind, Quicker and the other implementations that are going to be used also implemented this pull request.

The tests include an interoperability test, to test if Quicker can communicate with the other implementations, a performance test between all the implementations to check how many requests each implementation can handle and a client is going to test the mentioned implementations to check the protocol compliance of each implementation and how the implementation behaves with a badly implemented client. For each test that is performed, we first discuss how the test is going to be taken, how it is evaluated, what the expected results are and discuss the expected results with the real results.

For all the tests, an index.html file is used with a size of 177KB. The implementations and the test clients are run inside docker containers. The tests are taken as followed:

1. The containers for both client and server are created.
2. The server on one of the containers is started, together with tcpdump to capture all network traffic. The logs that each implementation creates are written to files.
3. After the server is started, the client starts tcpdump, creates a connection with the server and requests a file. The logs that each implementation creates are also written to a file.
4. After the connection closes because of an idle timeout, which is default 30 seconds, both containers are removed.

A slightly different approach is used for the performance test, where we do not wait for the idle timeout to close the connection. Instead, we close the connection as soon as we obtain the resource that has been requested, which is also index.html.

## 4.1 Quicker interoperability

The first test is the interoperability test. Quicker is tested against other implementations to check if it is implemented correctly. During the development of Quicker, there were days were all implementations tested against each other with the given implementation draft. Implementation drafts [33] are goals which are set for the implementations to implement various features (e.g., Packet number encryption, handshake, 0-RTT) and to test them for future decisions. This is to make sure that QUIC behaves as expected and when not, take calculated decisions to change the functionalities. To keep track which implementation has issues with particular features with other implementations, the working features are kept in a table in Google sheets [11]. Because these sheets are clear and are already used during the development, they are used in this thesis to show the interoperability of

Quicker against the tested implementations.

The features that are going to be tested are the following:

1. Version negotiation (V): if a client includes a version in the Initial packet that the server does not support, the server must send a Version Negotiation packet. Next, when the client receives a Version Negotiation packet, it either chooses one of the versions if they are supported by the client or the client terminates the connection when there is no supported version.

2. Handshake (H): The client and server can successfully perform a handshake.

3. Stream data (D): The client and server can both successfully send, receive and process data which is sent on one of the streams. This test does not include data that is coming from stream 0, which is used for the handshake. This is already tested with the Handshake test.

4. Connection close (C): Both endpoints can successfully close a connection when one of the endpoints indicates that it wants to close the connection.

5. Resumption (R): The client, which already communicated before with the server, can use the session data to perform session resumption.

6. 0-RTT (Z): The client can successfully send 0-RTT data and the server can receive and process it accordingly.

7. Stateless Retry (S): The server can send a Retry packet to the client, which is used by the server to process the Initial packet without keeping any state and makes it also possible for the server to perform address validation because only the legitimate client can respond to the Retry packet. The client can process the Retry packet and sent a new Initial packet accordingly.

8. Conenection migration (M): The migrating endpoint obtains one or more NEW_CONNECTION_ID frames from the other endpoint prior to starting the migration. When the migrating endpoint wants to migrate to an other four-tuple, it uses one of the Connection IDs to send data to the other endpoint.

The expected results for the interoperability of Quicker is that it can use all these features against other clients and servers from other implementations, except for the Retry and Migration tests. The server of Quicker is not able to perform a stateless retry. This is because a call to SSL_read_early_data() results in an error when using a stateless server, which uses SSL_stateless() function. More details about this issue can be found in issue 5235 [23] on github in the OpenSSL repository. However, in newer versions of OpenSSL, it is expected to be solved. Quicker does not support connection migration yet, as was indicated in section 3.8. For this reason, 'M' was not added to any of the interoperability results of Quicker.

The results for implementation draft 6 can be found in table 4.1 and 4.2. First of all, the implementations mvfst, winquic, Pandora, ngx_quic and AppleQUIC are closed source, where AppleQUIC does not even have a public endpoint to test its server. For the case of mvfst, ngx_quic and Pandora, the results were not added in the table because they did not implemented draft 12 + PR #1389 on their public server.

The public endpoint of winquic does implement draft 12 + PR #1389. However, its current TLS stack only supports TLS draft version 28 and there is something wrong with the version negotiation of the TLS stack, which choses the wrong TLS draft version that it does not support, which results in a failed handshake. To be able to still test winquic, we have disabled the draft versions 26 and 27 in the OpenSSL code inside NodeJS so that only TLS/1.3 draft version 28 is available in the supported versions list. After making this change, the handshake between Quicker and winquic succeeds. However, after the Quicker client sents the Finished message in a STREAM frame, it receives the NewSessionTicket in a Protected 1-RTT packet. The client of Quicker fails to decrypt this packet and closes the connection. The connection was closed prior to receiving any data, thus the data interoperability test failed. Also, because it did not receive the NewSessionTicket, session resumption is not possible and thus the resumption and 0-RTT, which needs the resumption, both fail. Finally,

| Interoperability matrix (1) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| client↓ \|server → | Minq | Mozquic | Quicly | quant | ngtcp2 | mvfst | picoQUIC | winquic |
| Minq | - | - | - | - | - | - | - | - |
| Mozquic | - | - | - | - | - | - | - | - |
| Quicly | - | - | VHDRZS | - | VHDRZ | - | VHDRZS | - |
| quant | - | - | VHDCRZ | VHDCRZSM | - | - | - | - |
| ngtcp2 | - | - | VH | - | VHDCRZ | - | - | - |
| mvfst | - | - | - | - | - | - | - | - |
| picoQUIC | - | - | VHDCRZS | - | VHDCRZ | - | VHDCRZS | - |
| winquic | - | - | - | VHDCRZS | VHDCRZ | - | - | - |
| f5 | - | - | - | - | - | - | - | - |
| ATS | - | - | - | - | - | - | - | - |
| Pandora | - | - | - | - | - | - | - | - |
| AppleQUIC | - | - | - | VH | - | - | V | VH |
| ngx_quic | - | - | - | - | - | - | - | - |
| !gquic | - | - | - | - | - | - | - | - |
| quic-go | - | - | - | - | - | - | - | - |
| quicker | - | - | V | VHDCRZS | VHDCRZ | - | - | VHC |

*Table 4.1: First table containing the results of the interoperability tests from implementation draft 6, where implementors need to implement QUIC draft 12 + PR #1389*

when the client of Quicker tests the stateless retry endpoint of winquic, it receives a Retry packet. The client of Quicker processes the Retry packet and responds with a new Initial packet. However, winquic does not respond to this new Initial packet and thus, the Retry test also fails.

Minq, Mozquic and picoQUIC are open source, which also provide a public endpoint. However, the public endpoints do not support draft 12 + PR #1389. By looking at the commits for Minq, it looks like they somehow skipped draft 12 or they are implementing draft 12 and 13 together. Because the handshake mechanism of draft 13 is completely different, we could not test Minq by using the source in github.
Mozquic just recently implemented draft 12, however it is not clear that PR #1389 has been implemented. Lastly, picoQUIC was setup locally. However, we tested picoQUIC with both ngtcp2 and Quicker and both could not perform the handshake without getting any errors (e.g., PROTOCOL_VIOLATION, enable to decrypt packet).
When testing quicly, version negotiation succeeds, however when the chosen version is used, Quicker fails to decrypt the first Handshake packet that is received from the server. We have not been able to determine what is going wrong, because ngtcp2 has no issues with both Quicker and quicly.
Testing against the servers of quant and ngtcp2 both succeeded with all the features that are available. Quant does implement connection migration, however, Quicker does not support it so VHDCRZS is everything that is possible between Quicker and quant. The server of ngtcp2 does not implement stateless retry because it also uses OpenSSL as its TLS stack.
F5 is the next implementation that has been tested by the Quicker client. The client succeeded to perform version negotiation, do a handshake and request data from the f5 server. Next it properly closes the connection. We believe that this is all that is possible with f5, because it corresponds with an entire ServerHello when we try to perform session resumption. Because f5 is closed source, we could not validate this assumption, however in prior implementation draft versions, it other implementations also did not succeed in getting more than VHDC.

Similar to f5, the result of the interoperability with ATS resulted in VHDC, which means that version negotiation, handshake, requesting data and closing the connection succeeded when testing the Quicker client against the ATS server.
Finally, !gquic and quic-go cannot be tested because they are based on the implementation of Google QUIC instead of IETF QUIC. The wire format is completely different at the time of writing, Google QUIC v44 only uses the invariant header of IETF QUIC.

This current interoperability table of implementation draft 6, which implements QUIC draft 12 is a bit empty in comparison with previous tables. If we look at the table of the previous implementation

| Interoperability matrix (2) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| client↓ \|server → | f5 | ATS | Pandora | AppleQUIC | ngx_quic | !gquic | quic-go | quicker |
| Minq | - | - | - | - | - | - | - | - |
| Mozquic | - | - | - | - | - | - | - | - |
| Quicly | - | - | - | - | VHD | - | - | - |
| quant | VHDC | VHDCM | - | - | - | - | - | VHDCRZ |
| ngtcp2 | - | - | - | - | VHDC | - | - | VHDCRZ |
| mvfst | - | - | - | - | - | - | - | - |
| picoQUIC | - | - | - | - | VHDC | - | - | - |
| winquic | - | - | - | - | VHDC | - | - | - |
| f5 | - | - | - | - | - | - | - | - |
| ATS | - | - | - | - | - | - | - | - |
| Pandora | - | - | - | - | - | - | - | - |
| AppleQUIC | - | VH | - | - | - | - | - | - |
| ngx_quic | - | - | - | - | - | - | - | - |
| !gquic | - | - | - | - | - | - | - | - |
| quic-go | - | - | - | - | - | - | - | - |
| quicker | VHDC | VHDC | - | - | - | - | - | VHDCRZ |

*Table 4.2: Second table containing the results of the interoperability tests from implementation draft 6, where implementors need to implement QUIC draft 12 + PR #1389*

draft, where implementations had to implement draft 11, we see that a lot more implementations were developing more active in comparison to the table of draft 12. The results of the interoperability tests of implementation draft 5 can be found in table 4.3 and 4.4. We can conclude that, even though the implementation drafts were used to immediatly test the current draft for possible issues, the usefulness of it has gone lost because of the way that QUIC evolves. Thus, all the major changes in each new draft results in more and more implementors that skip a draft to avoid implementing features that are possible to change in the next upcoming draft. This results in the fact that issues are discovered later, which by then, could be harder to change again.

| Interoperability matrix (1) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| client↓ \|server → | Minq | Mozquic | Quicly | quant | ngtcp2 | mvfst | picoQUIC | winquic |
| Minq | - | - | - | - | - | - | - | - |
| Mozquic | - | - | - | - | - | - | - | - |
| Quicly | - | - | - | - | - | - | - | - |
| quant | - | VHDCRZS | - | VHDCRZS | VHDCRZ | - | VHDCRZS | VHDCRZS |
| ngtcp2 | - | VHDCRS | - | VHDCRZS | VHDCRZ | - | VHDCRZS | VHDCRZS |
| mvfst | - | - | - | - | - | - | - | - |
| picoQUIC | - | VHDCRZS | - | VHDCRZS | VHDCRZ | - | VHDCRZS | VHDCRZS |
| winquic | - | VHDCRZS | - | VHDCRS | VHDCRZ | - | VHDCRZS | VHDCRZS |
| f5 | - | - | - | - | - | - | - | - |
| ATS | - | VHDC | - | VHDC | VHDC | - | VHDC | VHDC |
| Pandora | - | - | - | - | - | - | - | - |
| AppleQUIC | - | - | - | VH | - | - | V | VH |
| ngx_quic | - | - | - | - | - | - | - | - |
| !gquic | - | - | - | - | - | - | - | - |
| quic-go | - | - | - | - | - | - | - | - |
| quicker | - | VHDCR | - | VHDCRZS | VHDCRZ | - | VHDCS | VHDCRZS |

*Table 4.3: First table containing the results of the interoperability tests from implementation draft 5, where implementors need to implement QUIC draft 11*

## 4.2 Performance

The second test is the performance test. The performance of Quicker is tested against the other implementations to determine which implementation can handle more requests at the same time. We are also going to look at both the memory and CPU usage of the libraries during these tests.

| Interoperability matrix (2) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| client↓ \|server → | f5 | ATS | Pandora | AppleQUIC | ngx_quic | !gquic | quic-go | quicker |
| Minq | - | - | - | - | - | - | - | - |
| Mozquic | - | - | - | - | - | - | - | - |
| Quicly | - | - | - | - | - | - | - | - |
| quant | VHDC | VHDCRZ | - | - | VHDC | - | - | VHDCRZ |
| ngtcp2 | VHDC | VHDCRZ | - | - | VHDC | - | - | VHDCRZ |
| mvfst | - | - | - | - | - | - | - | - |
| picoQUIC | VHDC | VHD | - | - | VHDCS | - | - | VHDCRZ |
| winquic | VHDC | VHDCRZ | - | - | VHDC | - | - | VHDCR |
| f5 | - | - | - | - | - | - | - | - |
| ATS | VHDC | VHDC | - | - | - | - | - | VHDC |
| Pandora | - | - | - | - | - | - | - | - |
| AppleQUIC | - | - | - | - | - | - | - | - |
| ngx_quic | - | - | - | - | - | - | - | - |
| !gquic | - | - | - | - | - | - | - | - |
| quic-go | - | - | - | - | - | - | - | - |
| quicker | - | VHDC | - | - | VH | - | - | VHDCRZ |

*Table 4.4: Second table containing the results of the interoperability tests from implementation draft 5, where implementors need to implement QUIC draft 11*

Normally, the tested QUIC servers were ngtcp2 [8], quant [9] and Quicker. However, quant is at the time of writing not able to handle more than one connection that requests a resource and respond to this request at a time. We expect that the cause of this issue is the following: When the quant server is started, it binds to the interface and ports that were given as arguments. When there is an incoming connection, the q_accept function returns a connection. Within the main function, the request that is made by the other recipient is handled for this connection. When the server is finished handling the resource and the connection is closed, q_accept is called again to wait for a new connection. Because the handling of the connection does not happen on a different thread, the server can only serve resources to a single connection. However, we still found it odd that the handshake was performed of subsequent connections. When looking more closely to the code of quant, we noticed that the handshake is done, prior to the return of q_accept. We have informed the implementors of quant of this, however, they have not verified our assumption at the time of writing. In the meantime, we are going to use another implementation to do the performance test, which is quicly [7].

Prior to the tests, we look at the statistics of each container, which have 2000MB of RAM memory, using docker stats command to see the memory usage and the CPU load of each server when they are idle, prior to any incoming connections. The results of the tested implementations are shown in table 4.5.

| Memory usage implementations | | | |
|---|---|---|---|
| | ngtcp2 | quicly | quicker |
| Memory | 5.148MB | 4.797MB | 17.72MB |
| Memory percentage | 0.26% | 0.24% | 0.89% |
| CPU load percentage | 0.00% | 0.00% | 0.00% |

*Table 4.5: Table with the base memory usage of all implementations.*

Based on these results, the client of quicly could be chosen as the client to perform the performance tests based on the idle memory usage of quicly. We suspect that the client of quicly is the most performant implementation because the client and server share most of the codebase (except for the client/server specific code). However, as was mentioned in section 4.1, the client of quicly does not have interoperability with Quicker. For this reason, we have chosen ngtcp2 to be the client to perform the test with, as it has interoperability with both Quicker and quicly. Using this client, we are incrementally increasing the amount of connections to the server to test how many connection each

implementation can handle. The client of ngtcp2 that we are going to use is slightly modified [5]. Originally, the client performs the handshake, requests the resource and waits until the connection times out to close the process. We have modified it by closing the QUIC connection and exit the process immediately after receiving the resource that is requested. We close the connection immediately as we only want to measure how many short-lived connections the server can handle. We suspect that this could be a real situation as the use of 0-RTT makes it possible to request a new resource without the cost of the connection establishment. In the meantime, the client and server can free their resources for other usages. We also printed a statement when the resource of stream ID 4 has been received to verify that the request has been received by the client.

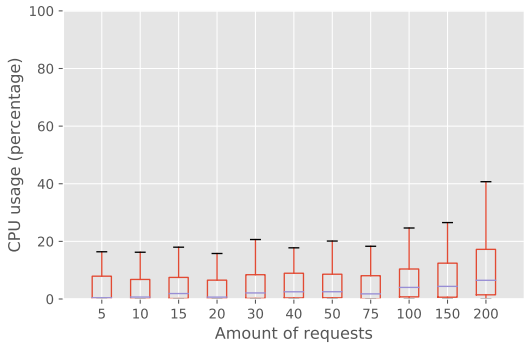
We expect that Quicker is going to be one of the slower implementations because Quicker is implemented in NodeJS, which uses the V8 engine [2] to translate JavaScript to machine code. Because Quicker is implemented in TypeScript, which is compiled to JavaScript, its performance is less than an implementation written entirely in C++. By using NodeJS, Quicker often also needs to swap between JavaScript code and C++ code for NodeJS itself and the OpenSSL functions that are needed for Quicker itself, which adds an additional delay. For this reason, we expect that Quicker is not going to handle as many requests or is not going to respond as fast as the C++ implementations (e.g., ngtcp2).

The tests are done on a Digital Ocean cloud server, located in Amsterdam, which has 1 vCPU with 2.3 Ghz, 2GB memory and 50GB SSD [4]. On this server, we start the servers of every implementation. We execute the performance tests five times on each implementation. To be able to request our own resources, we also had to modify quicly, because the resources that are available were hardcoded in the codebase of quicly. We added our own resources to this list to perform our tests.

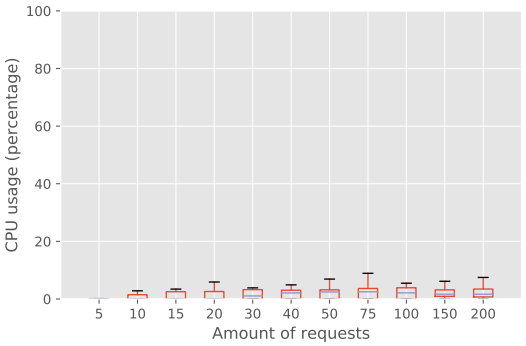
When we were taking the tests on the three implementations, we noticed that when the amount of requests was 20 or higher, quicly resulted in an abort, which means that an assert failed, or segmentation fault happened. On top of that, when quicly did not seem to have any issues, as it was not crashing, the communication just stopped suddenly until all the active connections had a timeout. When we looked more closely at the monitored results of quicly, we noticed that the median CPU usage went up from less than 2 percent at 5 request to 100 percent at 10 requests, which is illustrated in figure 4.1c. The median memory usage (figure 4.2c) with 15 requests and lower are approximately the same as ngtcp2. We have not identified the exact problem that causes these issues. We suspect that it entered a loop that it cannot exit and that this only occurs when there is loss on the path. We think this is due to loss because of the assert that was failed, as it cannot go further when the offset of the stream is greater than the start offset of the frame that needs to be transmitted. However, we have not been able to verify this with the implementors of quicly that this is in fact the issue.

When we compare the median CPU usages of Quicker and ngtcp2, we notice a huge difference, as ngtcp2 only uses 10-15 percent of the CPU, whereas Quicker goes up to 40 percent. When we look at the network I/O results of Quicker and compared them with ngtcp2, we see that the amount of data that was transmitted and received is less in Quicker when the amount of requests are 30 or higher. This would mean that Quicker cannot keep up with the amount of packets that are incoming and starts to drop packets. This would mean that the limit of Quicker is around 30 simultaneous connections and when more connections are opened, connections simply timeout. To verify this, we examined the logs of the clients to make sure all the connections had been closed successfully. While examining these logs, we saw that the amount of connections that Quicker can handle was even less than we first thought. Quicker started having issues when the amount of simultaneous connections was 15. With these connections, not the entire resource was served and the connection timed out. The results of ngtcp2 are more positive, as it can keep up with the amount of connections that are opened, which could also be verified by examining the logs of ngtcp2.

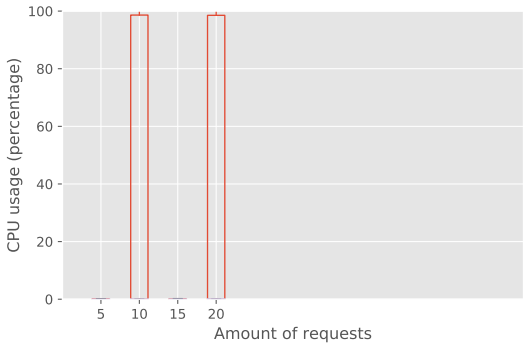
When we look at the memory usage of Quicker and ngtcp2 in figure 4.2, we see that the amount of memory that Quicker uses is ten times higher than the amount that ngtcp2 needs. We believe that the reason for the higher memory usage in Quicker is that it opens the entire file and immediately puts all the data on the streams, instead of maintaining a write buffer and reading the file block by block. This would result in less memory usage on any given moment, as only the size of a buffer is read into memory instead of reading the entire file, which is afterwards copied to the stream.




(a) CPU usage over various request counts of Quicker.

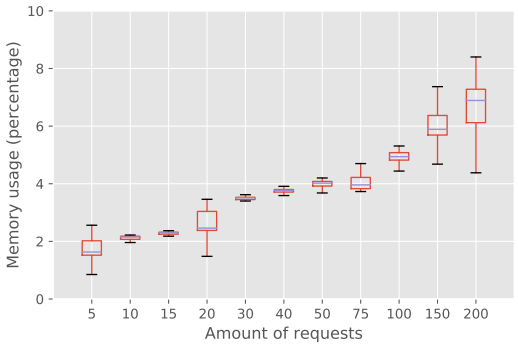



(b) CPU usage over various request counts of ngtcp2.

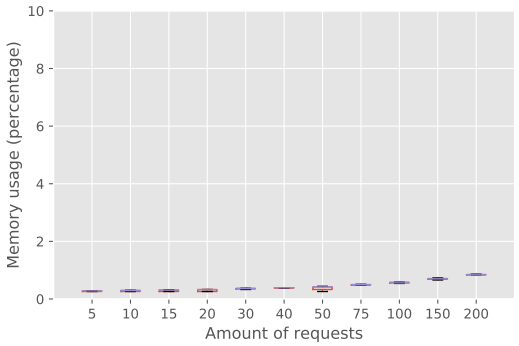



(c) CPU usage over various request counts of quicly.

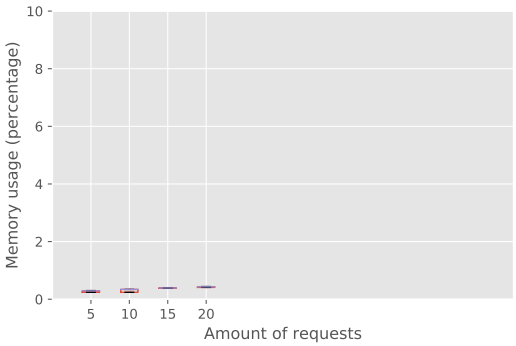
*Figure 4.1: Boxplots containing the median cpu usage results of the performance tests*




(a) Memory usage over various request counts of Quicker.




(b) Memory usage over various request counts of ngtcp2.




(c) Memory usage over various request counts of quicly.

*Figure 4.2: Boxplots containing the median memory usage results of the performance tests*

Not only quicly, but Quicker also suffered from lost packets. When we look at the logs of Quicker, we notice a couple of things. We see that sometimes an error occurs in loss detection because a variable was not initialized yet and which is only called when a packet is lost. A second error in Quicker is that it sometimes cannot decrypt a packet. This is the due fact that a packet number was not decrypted correctly, which results in that the server cannot decrypt the payload of the QUIC packet.

While searching for the bad results of Quicker, we found out when we were taking the logs to determine why it was dropping packets, it was due the fact that the client could not keep up with the server by logging all the messages instead of just the relevant ones. When we started logging only the relevant logs, which were the errors, we discovered yet another issue. The first reason why errors occurred with Quicker, happened when a packet was lost from the client to the server. The server constructed an ACK frame with an ackblock to show the gap of the missing packets. However, it seems that Quicker does not construct the ackblock correctly as the client of ngtcp2 raised an error that the frame was incorrect. As such, we could not determine the amount of packets that Quicker can process because the connections that lost a packet coming from the client to the server were invalid.
We can conclude from these results that ngtcp2 uses its resources better than Quicker, which is what we expected at the beginning. On top of that, we discovered a new issue in Quicker which is why we could not determine how many requests Quicker can handle. We also found out that quant currently cannot serve resources to more than one connection simultaneously. Finally, we suspect that quicly is not optimized yet to perform well when there is loss on the path. However, both our assumptions of quant and quicly were not verified yet by the implementors of quant and quicly.

## 4.3 Compliance

The last tests we are going to do is about protocol compliance. We are going to tweak some parts of the Quicker client to test how compliant each implementation is. First we are going to test each implementation with features that need to work, by checking if the servers are taken into account the limitations of the transport parameters (e.g., Set max stream data low, which should immediately result in a stream blocked) and we are going to intentionally send packets which should result in a protocol violation:

1. Random bytes: When random bytes are sent, the server must ignore this as it is a malformed packet and should not crash.
2. Client request at stream 3: The server should close the connection because stream 3 is a unidirectional stream for the server. The client is not allowed to start sending data or even sent any data on this stream, as it is a unidirectional stream for the server.
3. Start connection with Handshake packet: Server should ignore this packet, because connection must start with an Initial packet.
4. Badly formed frame: When an endpoint receives a badly formed frame within a packet, it needs to respond with a frame format error.
5. STOP_SENDING frame on client unidirectional stream: The server should respond with a protocol violation because a server cannot sent data on a client unidirectional stream.
6. Data after fin bit was set: With this test, the client sents data on a stream which was already closed. The server should respond to this with a final offset error because the final offset is the offset of the last STREAM frame incremented by the length of this frame.
7. UDP datagram with a lot of small QUIC packets: With this test, a client sents a UDP datagram containing a lot of small QUIC packets (e.g., Packets containing a single padding frame of size one) to check how various implementations react to this.
8. Duplicate packets: This test checks how an implementation behaves when a duplicate packet is received. Normally packet numbers cannot be used twice. However, on todays internet, it could

happen that a middlebox caches the QUIC packet and retransmit it when it thinks it was lost which results in the same QUIC packet that arrives twice. Implementations should ignore this packet as it is already received by the client. As was already mentioned in section 3.2, packet number ambiguity can occur because packet numbers are relative in a QUIC packet. When this occurs, the packet is still ignored as the correct packet number is needed to decrypt the payload of the packet.

9. Out-of-order packets: When an out-of-order packet is received by the server, frames other than STREAM frames can be used immediatly. The data from STREAM frames are buffered if the previous packet contains data from the same stream, as the offset is larger than the offset that is expected. When the previous packet does not contain data from the same stream, which results that the offset is the offset value that is expected by the server, the data can also be used immediatly by the server.

10. Flow control: With this test, the flow control functionality of the server is tested. The initial_max_stream_data is set low to force the stream to get blocked when a large file is requested, which is done with HTTP/0.9 because most implementations have implemented the basic HTTP functionalities. The server should not exceed the max_stream_data from the client until a MAX_STREAM_DATA frame has been received from the client.

We expect that each implementation takes into account the transport parameters which are set by the client and that they can handle packets which are not compliant to the QUIC protocol and respond to it correctly.

#### Random bytes

The Quicker client has been modified to send random bytes instead of the normal Initial packet. The first tested server was Quicker. Quicker ignored the random bytes because it could not parse the packet. The parsing of the header threw an exception because some values did not pass validation and because it is not an existing connection, it ignored the packet and continued serving other clients. The logs of ngtcp2 are a bit more useful than the logs of Quicker. These really show that the header could not be parsed and was ignored. Quant also succeeded the test by ignoring the random bytes. The logs show that it could not find any connection with the given Connection ID or that the packet was invalid.

#### Client request at stream 3

Streams can be unidirectional or bidirectional and can be started by a server and a client. However, as was mentioned in section 3.5, the ID of the stream indicates if the stream is unidirectional or bidirectional and started by the server or the client. Normally, the client cannot use stream three because it is unidirectional stream for the server. The client of Quicker has been modified to use stream three instead of stream four to request data.
The server of Quicker checks the ID prior to serving any data to the client. This validation failed for the request and the packet was ignored. Also, because it is not allowed to request data on stream three by the client, the server terminated the connection by sending a CONNECTION_CLOSE frame with the PROTOCOL_VIOLATION error. Similar to Quicker, ngtcp2 and quant also terminated the connection with PROTOCOL_VIOLATION, which mean that all implementations behave properly when a client tries to request data from the wrong stream.

#### Start connection with Handshake packet

The client needs to initiate the connection using a Initial packet, when other packet types are used, the server must ignore these packets. The other packet type that could be used is the Handshake packet, which resulted in the Quicker client that has been modified to initiate the connection using a Handshake packet instead of an Initial packet.
The Quicker server was tested first and it accepted the Handshake packet instead of ignoring it. This is because the Quicker server just creates a new connection when it cannot find the Connection ID that was used in the header of the packet. Ofcourse this can result in unexpected behaviour. To

resolve this issue, the packet type must first be checked prior to creating a new connection. When the packet type is not an Initial packet, the incoming packet is ignored. The ngtcp2 server did ignore the incoming Handshake packet and thus, is compliant to the specification of QUIC. As is the case with ngtcp2, quant also checked the Connection ID if there was already an existing connection. Because it could find any, the packet was also ingored in quant.

#### Badly formed frame

For the next test, the Quicker client is modified by changing the way a buffer is created for a STREAM frame. Instead of properly created the buffer, random bytes are generated and put into the packet instead to verify how an implementation would react to a badly formed frame.
Quicker did parse and tried to handle the incoming packet with the badly formed STREAM frame. Because it contained bad data, the length and offset were parsed incorrect. However, when trying to handle the badly formed frame, Quicker noticed the bad data and raised a PROTOCOL_VIOLATION error. Even though Quicker noticed something was wrong with the frame, it should have noticed it when parsing the frame. That way, unnecessary work does not need to be done. Ngtcp2 did notice the badly formed frame and responded correctly to it. Quant did also notice the badly formed frame, which is indicated in the log by the unknown frame type, which is probably the same way as ngtcp2 noticed it. Quicker did not because, when parsing the type of the frame, it checks if the type is bigger than some value. However, it does not check if the type is smaller than the upper bound and because of this, Quicker did not see the badly formed frame until it was trying to process it.

#### STOP_SENDING frame on client unidirectional stream

The Quicker client has been modified again to add a STOP_SENDING frame to each packet with a stream ID of a client unidirectional stream. On this type of stream, the server can never send any data, thus receiving a STOP_SENDING frame on a receive only stream should be considered as a PROTOCOL_VIOLATION error.
All implementations responded the same, by ignoring the incoming packet and closing the connection with the PROTOCOL_VIOLATION error.

#### Data after fin bit was set

When a stream receives a fin bit, it means that all the data has been sent from the sender on that stream. An example of this is when a client wants to request some resource with HTTP. The request would only contain the HTTP request and the stream can be closed on the end of the client. When the server receives the request, it notices that the fin bit is set and thus, can start processing the HTTP request and respond to it with a HTTP response. When a stream has been closed, data cannot be sent anymore because the data has already been processed by the receiving end. With this test, we have modified the Quicker client to request a second resource on a stream that has already been closed.
Quicker, ngtcp2 and quant notices that the stream has already been closed when new data arrives. All of them respond to it with a CONNECTION_CLOSE frame containing the error FINAL_OFFSET_ERROR, which means that the sender has sent data with an offset that exceeds the final offset and thus, all servers are compliant to the specification with this test.

#### UDP datagram with a lot of small QUIC packets

To perform this test, the Quicker client has been modified to send a coalesced packet, containing as many Handshake packet that are possible for a single UDP datagram that is valid for QUIC. Quicker sets this value to 1280 bytes, thus Handshake packet containing a single PADDING frame, with a size of 1, are added until the UDP datagram is full.
Quicker seems to have issues when parsing the coalesced packets. When it tries to decrypt the first packet, it raises an error that it cannot decrypt the packet and it closes the connection. This seems to indicate that the parsing of a coalesced packet still has some bugs in it. Ngtcp2 successfully parsed the coalesced packets and all of them are shown in the logfile of ngtcp2. The results of quant seems that it also failed parsing the coalesced packets. Quant managed to take Handshake packets out of

the UDP datagram, however the decoded packet numbers seems wrong a lot of times, which results that quant cannot decrypt the payload as it needs the packet number to do so.

#### Duplicate packets

With the following test, the client of Quicker has been modified again. This time, it sents every packet twice to see how implementations react when a packet is received with a packet number that was already used.
The server of Quicker still opens the packet and processed the content of the packet. Even though, the content is in this test the same, which result in a no-op because offsets are checked with STREAM frames and flow control is idempotent, an attacker could have inject the packet with other content and the server of Quicker would still process the packet. Quicker should check if the packet number has already been received. By looking at the logs of ngtcp2, it looks like the server does ignore duplicate packet numbers, which is the correct way to respond to a duplicate packet number. By looking at the logs of quant, it seems that its behaviour is similar to that from Quicker. It also decrypt the packets and handles them with whatever actions that needs to be done.

#### Out-of-order packets

For our next test, we are going to check if servers can handle out-of-order packets, by reordering the packet just before they are transmitted to the server. Servers should be able to process out-of-order packets when the packet number is not to large compared to the largest received packet number, otherwise there could be an ambiguous packet number, as was explained in section sec:quic-packet-numbers.
All the servers succeed with handling the out-of-order packet. The client has first transmitted the HTTP request before sending the Finished message of the handshake. Quicker, quant and ngtcp2 could handle the request and responded to it and thus, they can handle out-of-order packets when the packet number is not ambiguous.

#### Flow control

Finally, we are going to test flow control by setting the maximum allowed data on a stream to 10 bytes. This results in the fact that servers can only send 10 bytes before having to wait for a MAX_STREAM_DATA frame.
The Quicker server did not respond correctly to this constraint. Even though, flow control is not applicable to stream 0, the Quicker server sents a STREAM_BLOCKED frame after the handshake. The client still needs to receive the NewSessionTicket of the server and that is why the Quicker server sents a STREAM_BLOCKED frame. Thus, for flow control on stream 0, the Quicker server is not compliant to the specification of QUIC. Ngtcp2 did respond correctly by waiting until a new MAX_STREAM_DATA frame was received and by not sending a STREAM_BLOCKED frame for stream 0 after finishing the handshake with the client. Quant responded to the low maximum allowed data by sending a STREAM_BLOCKED frame for the stream where the resource was requested. The Quicker client responded by sending a MAX_STREAM_DATA frame, however the quant server did not react to this. It looks like, when the maximum allowed data on a stream is to low, the quant server does not send any data.

# Chapter 5

# Conclusion and future work

Despite the fact that there was already some knowledge about the features of QUIC, because of Google QUIC, it still changes frequently. Examples of these changes are dual Connection IDs instead of using one single Connection ID for both endpoints, which was introduced in draft 11 and packet number encryption, which was added in draft 12. To be able to adept to these changes, we had to make sure that the implementation was written with the knowledge that things could change. However, at the start of this thesis, it was not clear that these things could change as fast as sometimes happened between different drafts. This made it hard for us, with only base knowledge about transport protocols and with new design decisions in every new draft, it was not always clear how to implement or change functionalities of QUIC.

That is why only the more important features of the transport protocol of QUIC were implemented in Quicker instead of implementing the entire protocol and HTTP/QUIC. Examples of these that were left out of Quicker are the ability to migrate the connection, as it was a challenge to determine when to change Connection ID (e.g., NAT rebinding, change of network) and the ability to coalesce packets to transmit multiple QUIC packets in a single UDP packet. Also features, such as 0-RTT, multiplexing of streams and packet number encryption were added because these were one of the core reasons of the QUIC protocol. Others, such as flow control and the handling of ACKs were added because these were more important and were necessary to be able to evaluate the interoperability of Quicker against other libraries. However, these were kept relatively simple to be able to focus more on the correctness of the core features.

According to the milestones of the datatracker [3], the due date of the QUIC protocol, including the HTTP/QUIC and QPACK specifications, is November 2018. However, the implementation drafts and the interoperability of QUIC libraries has been focused on the core transport of QUIC. HTTP/QUIC and QPACK which are based on HTTP/2 have only seen some limited testing and are not even fully implemented by anyone. As such, there has been no opportunity yet to discover limitations and design issues in one of these specifications. On top of that, at the time of writing, there were still 94 open issues on github for both HTTP/QUIC as for the transport part of protocol, even though this part has already been tested by various implementors in the last months. We can only conclude from this that it is not feasible to finish the first version of QUIC in November 2018 as it will probably still contain issues which were not discovered yet.

Also, while the implementation drafts were made available to evaluate the design of QUIC and make calculated decisions for possible changes, the rapid evolution of QUIC seem to scare off some implementors to wait for the next draft to start implementing features to avoid refactoring these when a new draft comes out, as was shown in 4.1 where less implementors tested interoperability of draft 12 in comparison to draft 11. While this is understandable, issues in decisions are discovered later one in the process because of this and as such, the implementation drafts lose some of their usefulness.

Additionally, we discovered how important it is to experiment with outliers of situations. During the development of Quicker, we mostly looked at the more common situations to test our implementation. However, as was seen in section 4, testing the outliers can be important to find possible vulnerabili-

ties, as was the case for quicker, which would process a packet with the same packet number twice.

Lastly, we have evaluated Quicker against other implementations to see how performant Quicker is, when comparing it with others. We have seen that the base memory and cpu usage is significant more than the these of other implementations because it is implemented in NodeJS. However, we knew that the decision to use NodeJS was not to make the most performant implementation. It was to make the QUIC protocol more readable for others who would like to understand QUIC by looking at source files instead of the specifications. However, we were not able to succeed with this goal, as some parts, such as flow control, are implemented to complex to be readable for people that do not understand QUIC.

**Future work**

The first and most obvious possibility is to continue the development of Quicker to have a working QUIC implementation in NodeJS by the time the drafts of QUIC are released. However, it would probably need some refactoring of the current structure because the next following draft versions significantly change the mechanisms of QUIC, where the current structure is not ideal. Also, during the compliance evaluations in section 4.3, we have seen that some features (e.g., flow control), need some refactoring to be fully compliant to the draft version that was used in this thesis, which was draft 12.

A second possibility is, besides the further development of QUIC, to create outlier situations to test not only our, but also other implementations so they can test if their implementation follows the specification of QUIC. This would mean that we expand the amount of tests that we already did in the compliance tests of section 4.3.

A final possibility is to create a C++ module for the wrapper class which is now directly implemented in NodeJS. By using a module, it would be possible to use Quicker in more recent versions of NodeJS, instead of using our modified NodeJS version. We already looked at the possibility of doing this and noticed that we could remove the dependency of having OpenSSL. By created a separate module, we could make it possible to use other TLS stacks, such as PicoTLS and BoringSSL.

# List of Figures

# List of Tables

# Bibliography


[1] Bn.js. https://github.com/indutny/bn.js/. Accessed: 15/08/2018.

[2] Chrome v8 engine. https://developers.google.com/v8/. Accessed: 22/08/2018.

[3] Datatracker quic. https://datatracker.ietf.org/wg/quic/about/. Accessed: 22/08/2018.

[4] Digital ocean droplet. https://www.digitalocean.com/products/linux-distribution/ubuntu/. Accessed: 22/08/2018.

[5] fork from ngtcp2/ngtcp2. https://github.com/kevin-kp/ngtcp2. Accessed: 23/08/2018.

[6] Google quic wire layout specification. https://docs.google.com/document/d/1WJvyZflAO2pq77yOLbp9NsGjC1CHetAXV8I0fQe-B_U/edit. Accessed: 16/03/2018.

[7] h2o/quicly. https://github.com/h2o/quicly. Accessed: 23/08/2018.

[8] ngtcp2/ngtcp2. https://github.com/ngtcp2/ngtcp2. Accessed: 23/08/2018.

[9] Ntap/quant. https://github.com/NTAP/quant. Accessed: 23/08/2018.

[10] Quic charter. https://datatracker.ietf.org/doc/charter-ietf-quic/. Version: 01, Accessed: 08/04/2018.

[11] Quic interop matrix. https://docs.google.com/spreadsheets/d/1D0tW89vOoaScs3IY9RGC0UesWGAwE6xyLk0l4JtvTVg/edit#gid=1900425901. Accessed: 15/08/2018.

[12] Versions in extensions, pr #632. https://github.com/tlswg/tls13-spec/pull/632, 2016.

[13] Middlebox changes2 hrr, pr #1091. https://github.com/tlswg/tls13-spec/pull/1091, 2017.

[14] Response to lost handshake packets, issue #169. https://github.com/quicwg/base-drafts/issues/169, 2017.

[15] Simultaneous connection migration, issue #490. https://github.com/quicwg/base-drafts/issues/490, 2017.

[16] A 17 octet connection id, pr #1088. https://github.com/quicwg/base-drafts/pull/1088, 2018.

[17] Are random packet number skips still relevant for opportunistic ack protection? , issue #1030. https://github.com/quicwg/base-drafts/issues/1030, 2018.

[18] Asymmetric connection id, issue #1089. https://github.com/quicwg/base-drafts/issues/1089, 2018.

[19] Make the packet number encryption sampling clearer, pr #1389. https://github.com/quicwg/base-drafts/pull/1389, 2018.

[20] Move opportunistic ack defense to security considerations, pr #1185. https://github.com/quicwg/base-drafts/pull/1185, 2018.

[21] Other congestion controllers, pr #1633. https://github.com/quicwg/base-drafts/pull/1633, 2018.

[22] Symmetric connection id, pr #1151. https://github.com/quicwg/base-drafts/pull/1151, 2018.

[23] Tls1.3: Stateless retry and ccs, issue #5235. https://github.com/openssl/openssl/issues/5235, 2018.

[24] Barford, P., and Crovella, M. A performance evaluation of hyper text transfer protocols. Tech. rep., Boston, MA, USA, 1998.

[25] Belshe, M., Peon, R., and Thomson, M. Hypertext transfer protocol version 2 (http/2). RFC 7540, RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7540.txt.

[26] Berners-Lee, T. The original http as defined in 1991. https://www.w3.org/Protocols/HTTP/AsImplemented.html, 1991. Accessed: 16/03/2018.

[27] Berners-Lee, T., Fielding, R. T., and Nielsen, H. F. Hypertext Transfer Protocol – HTTP/1.0. RFC 1945, RFC Editor, May 1996. http://www.rfc-editor.org/rfc/rfc1945.txt.

[28] Bishop, M. Hypertext Transfer Protocol (HTTP) over QUIC. Internet-Draft draft-ietf-quic-http-12, IETF Secretariat, May 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-http-12.txt.

[29] Bynens, M. Bigint: arbitrary-precision integers in javascript. https://developers.google.com/web/updates/2018/05/bigint, may 2018. Accessed: 15/08/2018.

[30] Carlucci, G., Cicco, L. D., and Mascolo, S. Http over udp: an experimental investigation of quic. In *SAC* (2015).

[31] Chu, J., Dukkipati, N., Cheng, Y., and Mathis, M. Increasing TCP's Initial Window. RFC 6928, RFC Editor, April 2013. http://www.rfc-editor.org/rfc/rfc6928.txt.

[32] Dukkipati, N., Refice, T., Cheng, Y., Chu, J., Herbert, T., Agarwal, A., Jain, A., and Sutin, N. An argument for increasing tcp's initial congestion window. *SIGCOMM Comput. Commun. Rev. 40*, 3 (June 2010), 26–33.

[33] Eggert, L. 1st implementation draft. https://github.com/quicwg/base-drafts/wiki/1st-Implementation-Draft. Accessed: 15/08/2018.

[34] Ekiz, N., and Amer, P. D. Transport layer reneging. *Computer Communications 52* (2014), 82 – 88.

[35] Grigorik, I. High performance browser networking. https://hpbn.co. Accessed: 16/03/2018.

[36] Hesmans, B., Duchene, F., Paasch, C., Detal, G., and Bonaventure, O. Are tcp extensions middlebox-proof? In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (New York, NY, USA, 2013), HotMiddlebox '13, ACM, pp. 37–42.

[37] Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., and Tokuda, H. Is it still possible to extend tcp? In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2011), IMC '11, ACM, pp. 181–194.

[38] Huitema, C. What was yesterday's packet number decision? https://mailarchive.ietf.org/arch/msg/quic/L50RJhmPeHN8z8qe1VHFUGxIt6E, January 2018. Accessed: 15/08/2018.

[39] Iyengar, J., and Swett, I. QUIC Loss Detection and Congestion Control. Internet-Draft draft-ietf-quic-recovery-12, IETF Secretariat, May 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-recovery-12.txt.

[40] Iyengar, J., and Thomson, M. QUIC: A UDP-Based Multiplexed and Secure Transport. Internet-Draft draft-ietf-quic-transport-12, IETF Secretariat, May 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-12.txt.

[41] Karn, P., and Partridge, C. Improving round-trip time estimates in reliable transport protocols. *ACM Trans. Comput. Syst. 9*, 4 (Nov. 1991), 364–373.

[42] Krasic, C., Bishop, M., and Frindell, A. QPACK: Header Compression for HTTP over QUIC. Internet-Draft draft-ietf-quic-qpack-01, IETF Secretariat, June 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-qpack-01.txt.

[43] Krishnamurphy, B., Mogul, J. C., and Kristol, D. M. Key differences between http/1.0 and http/1.1. *Comput. Netw. 31*, 11-16 (May 1999), 1737–1751.

[44] Kuehlewind, M., and Trammell, B. Applicability of the QUIC Transport Protocol. Internet-Draft draft-ietf-quic-applicability-01, IETF Secretariat, October 2017. http://www.ietf.org/internet-drafts/draft-ietf-quic-applicability-01.txt.

[45] Kurose, J. F., and Ross, K. W. *Computer Networking: A Top-Down Approach (6th Edition)*, 6th ed. Pearson, 2012.

[46] Mathis, M., and Mahdavi, J. Forward acknowledgement: Refining tcp congestion control. *SIGCOMM Comput. Commun. Rev. 26*, 4 (Aug. 1996), 281–291.

[47] Mathis, M., Mahdavi, J., Floyd, S., and Romanow, A. TCP Selective Acknowledgment Options. RFC 2018, RFC Editor, October 1996.

[48] McGrew, D. An Interface and Algorithms for Authenticated Encryption. RFC 5116, RFC Editor, January 2008. http://www.rfc-editor.org/rfc/rfc5116.txt.

[49] Paasch, C. Network support for tcp fast open (nanog 67 presentation). https://www.nanog.org/sites/default/files/Paasch_Network_Support.pdf, June 2016. Accessed: 30/07/2018.

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

[51] Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., and Raghavan, B. Tcp fast open. In *Proceedings of the 7th International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2011).

[52] Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchene, F., Bonaventure, O., and Handley, M. How hard can it be? designing and implementing a deployable multipath TCP. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX Association, pp. 399–412.

[53] Salowey, J., Choudhury, A., and McGrew, D. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288, RFC Editor, August 2008. http://www.rfc-editor.org/rfc/rfc5288.txt.

[54] Spero, S. E. Analysis of http performance problems. https://www.w3.org/Protocols/HTTP-NG/http-prob.html, 1994.

[55] Stenberg, D. Http2 explained. *SIGCOMM Comput. Commun. Rev. 44*, 3 (July 2014), 120–128.

[56] Sullivan, N. Why tls 1.3 isn't in browsers yet. https://blog.cloudflare.com/why-tls-1-3-isnt-in-browsers-yet/, December 2017. Accessed: 04/08/2018.

[57] Swett, I. Quic fec v1. https://docs.google.com/document/d/1Hg1SaLEl6T4rEU9j-isovCo8VEjjnuCPTcLNJewj7Nk/edit. Accessed: 08/04/2018.

[58] The Chromium Projects. Spdy protocol. https://www.chromium.org/spdy/spdy-protocol. Accessed: 02/07/2018.

[59] Thomson, M. Version-Independent Properties of QUIC. Internet-Draft draft-ietf-quic-invariants-01, IETF Secretariat, March 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-invariants-01.txt.