Bachelor's Thesis

Development Of A Minimal QUIC Implementation In Rust: An Introduction To Next Generation Networking

submitted by

Christoph Matthias Britsch

from Berlin

Department Operating Systems
Prof. Dr. Michael Schöttner
Heinrich Heine University Düsseldorf

18. März 2024

Primary Reviewer:	Prof. Dr. Michael Schöttner
Secondary Reviewer:	Prof. Dr. Martin Mauve
Supervisor:	M.Sc. Fabian Ruhland

Contents

Chapter 1

Introduction

Over the past four decades, the reliability of the Transmission Control Protocol (TCP) has been the cornerstone of typical, day-to-day user traffic on the Internet. TCP, renowned for its ability to provide reliable, ordered, and error-checked data transmission between two applications, has been the foundation in sustaining the connectivity that underpins our digital experiences. However, the ever-accelerating evolution of network landscapes, coupled with an intensifying demand for heightened flexibility, security, and superior performance, has catalyzed in the exploration of alternatives. In response to these demands, the QUIC protocol has emerged as a solution to next-generation networking, with its roots tracing back to the laboratories of Google. In a contemporary environment where the static nature of traffic from predominantly home networks has given way to a dynamic mix of devices, networks and technologies, QUIC represents the much needed next step in networking. Unlike its predecessor, QUIC is designed to meet the challenges of a modern digital ecosystem, where seamless and secure connections are imperative across a diverse array of devices and usage scenarios. As our lives become increasingly interconnected and reliant on a myriad of digital interfaces, QUIC stands as the widely adopted next step, ensuring not only reliability but also security.

1.1 Motivation

The current TCP/IP stack faces limitations in latency and connection establishment times due to its three-way handshake mechanism. QUIC, built on top of UDP, offers significant improvements ocer TCP. Firstly, it utilizes a single handshake for establishing a secure connection, leading to faster connection establishment. Secondly, QUIC integrates a high performance stream control system combined with a robust error detection system, allowing for reliable data transmission even with packet loss, improving application performance.

QUIC provides an excellent opportunity to utilise a systems programming language for an implementation in order to maximise performance while managing resources at byte level. The programming language Rust has a number of benefits as it is currently gaining traction due to its focus on memory safety, performance, and concurrency. Its ownership system eliminates dangling pointers and memory leaks, crucial for low-level network programming. Additionally, Rust's ability to express concurrency efficiently makes it well-suited for handling the asynchronous nature of network communication.

This thesis aims to develop a basic QUIC implementation in Rust that demonstrates basic connection handling, data transfer, and the library design. While not a full-fledged production-ready implementation, this prototype will provide valuable insights into QUICs packet and header designs, its handshake mechanisms, its security measures and its performace benefits over TCP.

1.2 Objective

This bachelor's thesis objective is to implement a standalone version of the QUIC transport protocol in Rust, utilizing version rustc 1.72.0 and the UDP-Socket from the Rust standard library as its foundation. The implementation aims to comprise several features crucial for a functional QUIC server endpoint. These include the ability to send and receive QUIC packets, parsing, and encryption/decryption of both header and payload data. Additionally, the system aims to support a fully executed 1-RTT QUIC handshake between two endpoints, employing TLS 1.3 through the Rustls crate.
In developing this implementation, reference is exclusively made to RFCs 8999-9001, chosen due to a lack of comparable open-source implementations in Rust. A focus is set on handling all frames within the payload, with special attention given to Cryptoframes and Dataframes.
The implementation makes use of existing tools, such as the Rustls crate with the "quic" feature for a complete TLS 1.3 implementation and the Octets crate, serving as a Rust zero-copy mutable Byte Buffer to facilitate parsing.
Certain features are deliberately omitted due to time constraints. These include 0-RTT packets, version negotiation between different QUIC versions, connection migration, a client endpoint, asynchronous processing, and advanced flow control.
The primary communication goal is to establish seamless communication with a Python-implemented client. This client initiates the handshake and subsequently sends individual data packets as payloads, awaiting acknowledgment from the Rust-based server. Notably, the Python QUIC client is implemented using the open-source implementation aioquic.
Furthermore this bachelor's thesis aims to provide a comprehensive exploration of the QUIC protocol, exploring its intricacies and explaining its functionalities. By examining QUIC's features, particularly its transport, encryption, multiplexing, and error-handling mechanisms, one can better discern how QUIC stands out in terms of resilience, efficiency, and security when compared to TCP.

1.3 Overview

Starting with the subsequent chapter, "Fundamentals", the thesis establishes a foundation for understanding the principles of QUIC. Key terms and definitions are clarified, followed by an introduction into networking protocol fundamentals. Furthermore, both TCP and UDP are introduced in more detail.
Chapter three forms the core of the thesis, in which QUIC is examined in detail. It starts by exploring QUIC's historical background, clarifying its goals in addressing problems faced by TCP and other protocols, and continues with a detailed breakdown of its architecture. Moving into the more practical part of QUIC, the following subsections break down its connections and the ways in which QUIC deals with different errors and how it recovers from them. Finally, we examine the security features integrated into QUIC.

Moving into the implementation chapter, the rationale behind choosing Rust as the implementation language is explained and the scope of the QUIC implementation is set, followed by the development environment specifics and all external libraries the implementation makes use of. The main part of chapter four deals with the library layout and dives deep into specifics of connection establishment, packet handling and stream management. Lastly both protocol and implementation error handling mechanisms will be examined before concluding the chapter with an overview of the strategies used to test and validate parts of the implementation.
The "Evaluation" chapter critically assesses the implemented QUIC protocol. It addresses encountered challenges, acknowledges inherent limitations, evaluates performance considerations, provides considerations for future improvements and modifications, and conducts a comparative analysis with other existing implementations.
In the concluding chapter, key findings are laid out and a conclusion is drawn. The end of this thesis provides an outlook into the future of the QUIC implementation.

Chapter 2

Fundamentals

2.1 Terms and Definitions

- **Endpoint:** An entity that can participate in a QUIC connection by generating, receiving, and processing QUIC packets. There are only two types of endpoints in QUIC: client and server.

- **QUIC packet:** A complete processable unit of QUIC that can be encapsulated in a UDP datagram. One or more QUIC packets can be encapsulated in a single UDP datagram.

- **Variable Sized Integer** QUIC uses two most significant bits of the first byte to encode the base-2 logarithm of the integer encoding length in bytes. The remaining bits store the integer value itself, in network byte order. This allows encoding integers in 1, 2, 4, or 8 bytes, representing 6-, 14-, 30-, or 62-bit values.

- **Explicit Congestion Notification (ECN)**: ECN is an optional mechanism within the Internet Protocol (IP) that allows routers to signal incipient congestion to data senders before packet drops occur. This is achieved by routers setting a dedicated ECN bit in the IP header of packets traversing congested paths.

- **ACK Range:** An ACK range specifies a continuous sequence of packets that have been successfully received by the receiver. When a receiver acknowledges data, it doesn't necessarily need to acknowledge each packet individually. Instead, it can efficiently transmit an ACK with a range, indicating that it has received all packets within that range, from a starting point to an ending point.

- **Connection ID:** An identifier that is used to identify a QUIC connection at an endpoint. Each endpoint selects one or more connection IDs for its peer to include in packets sent towards the endpoint. This value is opaque to the peer.

- **Authenticated Encryption with Associated Data (AEAD):** AEAD is a cryptographic mode that builds upon Authenticated Encryption (AE). AEAD allows messages to include Associated Data (AD) alongside the actual message to be encrypted. This associated data is additional information that is not confidential but requires authenticity and integrity protection. In essence, AEAD authenticates the confidentiality of the encrypted message and the integrity of both the message and the associated data. This makes AEAD suitable for scenarios like network packets, where the header containing routing information needs to be integrity-protected and authenticated, while the payload carrying the actual content is encrypted for confidentiality.

Chapter 3

QUIC In Detail

QUIC is a transport layer network protocol. Although initially proposed to be an acronym for "Quick UDP Internet Connections", QUIC is just the name[1, p. 10]. It represents the next step in offering reliable, ordered, secure, and error checked data transport for connection-oriented web applications that traditionally relied on TCP. Improvements include enhanced performance, better security through the direct integration of TLS 1.3 and more flexibility by decoupling an endpoint from the actual interface address through the use of seperate Connection Identifiers.
Initially conceptualized by Jim Roskind at Google in 2012 and publicly disclosed in 2013 as an experiment, QUIC formally materialized in 2016 at an IETF meeting. QUIC version 1.0 was officially standardised in May 2021 across RFC's 8999[2], 9000[1], 9001[3] and 9002[4]. Its design aims to prevent protocol ossification, ensuring its continued evolution, unlike TCP, which has experienced notable ossification.
Today, QUIC plays a pivotal role in internet communication, being utilized in over half of the connections from the Chrome web browser to Google's servers. Notably, major web browsers such as Microsoft Edge (post-version series 1.x, a descendant of the open-source Chromium browser), Firefox, and Safari have extended support for QUIC.

3.1 Historic Background

In May 1974, 50 years ago, Vint Cerf and Bob Kahn outlined a protocol for network resource sharing via packet switching between network nodes. The resulting protocol, detailed in RFC 675 (Specification of Internet Transmission Control Program), was authored by Vint Cerf, Yogen Dalal, and Carl Sunshine, and released in December 1974. At the core of this model was the Transmission Control Program, encompassing both connection-oriented links and datagram services between hosts. Over time, the monolithic Transmission Control Program underwent a transformation into a modular architecture, giving rise to the Transmission Control Protocol (TCP), Internet Control Message Protocol (ICMP) and the Internet Protocol (IP). This architectural shift led to the emergence of a networking model informally known as TCP/IP, although it was formally referred to as the Internet Protocol Suite. Over time, a number of changes have been made to the original TCP specification in RFC 793 to adapt to the increasingly changing nature of the internet. Updates to RFC 793 have been made in, but are not limited to RFCs 879, 2873, 6093, 6429, 6528, and 6691.

3.1.1 An Outline of TCP

As a transport protocol, TCP roughly performs the following three basic functions

(1) reliable and ordered byte stream transmission with window-based flow control

(2) multiplexed data transmission through the use of port numbers

(3) congestion control to prevent network overload

First(1), in TCP, each data unit, that is every byte, is assigned a unique sequence number. Multiple, sequential data units are called a segment. A segment is identified by its highest sequence number. This sequential numbering allows both the sender and receiver to keep track of the data exchanged during communication. Upon receiving data, the receiver acknowledges the receipt of each segment by sending back an ACK packet containing the next expected sequence number. This acknowledgment mechanism ensures that the sender knows which data segments have been successfully received by the receiver. Additionally, TCP implements a window-based flow control to regulate the rate of data transmission between the sender and receiver. A window is defined as the maximum possible number of in-flight data segments. The sender maintains a sliding window that represents the maximum number of unacknowledged bytes that can be sent at any given time. As the receiver processes data and acknowledges its receipt, the sender adjusts the size of the window dynamically, allowing for efficient data transfer without overwhelming the receiver or causing high rates of packet loss. By combining sequential sequence numbers and ACKs, TCP achieves reliable and ordered byte stream transmission. Sequence numbers ensure that data segments are transmitted in the correct order, while ACKs confirm the successful receipt of each segment. Furthermore, window-based flow control optimizes the transmission rate, ensuring efficient data exchange between the communicating endpoints. This combination of mechanisms enables TCP a robust way to provide dependable and organized data transmission over networks.
Second(2), each TCP segment contains source and destination port numbers in its header, enabling the receiving device to match incoming data to the appropriate application or process based on the destination port number. Port numbers serve as identifiers for a specific process, allowing for the simultaneous handling of multiple connections on the same network interface. When a TCP connection is established, both the client and server use specific port numbers either chosen or set by standard. The combination of the source IP address, source port number, destination IP address, and destination port number forms a unique socket, enabling the identification of each connection on the network. By utilizing port numbers, TCP, in combination with IP, enables multiplexing, allowing multiple applications or services to communicate independently over a single network connection. This capability was revolutionary 50 years ago and provided the foundation for what we know today where a single device establishes dozens of connections at any given time.
Third(3), Congestion Control (CC) aims to to alleviate congestion and prevent further deterioration of network performance once packet loss is detected. TCP's CC operates by dynamically adjusting the rate at which data is transmitted based on network conditions. When packet loss is detected, TCP reduces its transmission rate to alleviate congestion and prevent further deterioration of network performance. One of the primary congestion control algorithms used in TCP is known from TCP Tahoe, a specialised TCP version

which employs a conservative approach to congestion avoidance. When packet loss is detected, TCP Tahoe enters a congestion avoidance phase where it reduces its transmission rate by halving the congestion window size. This cautious approach helps TCP adapt to changing network conditions while minimizing the risk of exacerbating congestion.

3.1.2 Problems of TCP

After its emergence, TCP quickly became the backbone for most of the worlds HTTP traffic. Estimates from 2010 approximated the worldwide TCP traffic at roughly 85% [5]. While being a highly deployed protocol, TCP does not come without issues. Particularly issues that stem from the exponential increase in internet traffic and therefore could not have been anticipated by its inventors. Hence, there have been a lot of additions to the original RFC, many of which try to fix these problems or which expand upon the original specification, for example TCP Options and Maximum Segment Size (MSS) in RFC 6691[6] which updates suggestions for what value to use for the MSS option. That has resulted in significant protocol ossification and, in addition, the major issues remain unfixed to this day. There have been a significant number of attempts to adapt TCP into different versions to address these issues but none of them succeeded in providing a combined solution for all major problems. One example is named Multipath TCP (MPTCP), which allows TCP to facilitate multiple, concurrent data streams, but it was ultimately constrained by middlebox behaviour. Further attempts include but are not limited to TCP Fast Open, Stream TCP and Transaction TCP.
All of these represent the attempt to fix TCPs long standing problems.

1. Head-Of-Line blocking, where a single lost or delayed packet can halt the delivery of subsequent packets, even if they are unrelated. This occurs because TCP guarantees in-order delivery of data, so if a packet is lost or delayed, subsequent packets must wait until it is retransmitted or delivered. This can lead to inefficiencies and delays, particularly in scenarios where real-time communication is crucial.

2. Long delay incurred during connection setup. TCP with TLS requires a three-way handshake process to establish a secure connection, which typically involves three round-trip exchanges between the client and server, plus an additional round trip for acknowledgment. This can result in significant latency, particularly in situations where rapid connection establishment is essential.

3. A fixed IP address per connection endpoint. This means that if a device's IP address changes, existing TCP connections must be terminated and reestablished with the new address. This can introduce disruptions and additional overhead, particularly in mobile or dynamic networking environments where IP addresses frequently change.

4. TCP's congestion control and reliable data delivery mechanisms are tightly coupled, which can sometimes lead to inefficiencies. For example, TCP's congestion control algorithms may throttle data transmission rates in response to perceived network congestion, even in situations where packet loss is minimal or unrelated to network congestion, leading to suboptimal performance.

3.1.3 Present Day

Today, the internet has widely outgrown the original design goals of TCP both in performance and complexity and transformed into a complex network of different devices and subnetworks, some of which are not physically connected anymore but through the use of radio technology. LTE and 5G allow for seemingly uninterrupted internet access while moving but due to their physical properties, they have an inherent tendency for losing connections. The original protocol design could not have possibly considered that one day mobile devices would leave and join network access points within milliseconds. These factors contribute to frequent termination and re-establishment of sessions, resulting in time and resource intense re-establishment from scratch. Said issues have become more and more prevalent with time and resulted in countless small fixes that combine into a vastly complex set of protocols, standards and routines that differ from carrier to carrier, e.g. running transmission sessions over separately established encrypted tunnels or keeping content related state transfer on top of transmission protocol states. The need to run session establishment of each involved protocol separately in sequence further contributes to time and resource consumption.

The growing complexity of services, employing modern design paradigms such as meshing, service-based architecture and microservices, enhances the impact of these design shortcomings, as the number of sessions and the number of endpoints to which they are established is further increased. To increase efficiency in general, but specifically for mobile devices making use of less powerful hardware and their dependency on battery power, as well as enabling new architectural concepts on server side, while maintaining customer experience when growing service meshes, a new, more efficient approach on the network stack level was needed.

Big providers, namely Google, identified these problems years ago and started development on a radically rethought transmission protocol which addresses todays challenges and aims to facilitate continuous development by integrating the idea that the protocol itself will evolve beyond the current scope. In 2016 the IETF officially assigned a working group to the QUIC project[7]. RFC's 8999[2], 9000[1], 9001[3] and 9002[4], standardising QUIC version 1, were officially released in May 2021. One month later, the RFC for HTTP/3 was released, marking the first application layer protocol which makes full use of QUIC[8]. In May 2023, QUIC version 2 officially released, bringing only very minor changes[9]. Its objective is to counteract diverse ossification vectors and utilize the version negotiation framework. Additionally, version 2 functions as a blueprint for the minimal alterations in potential future versions of QUIC.

Since 2021, the adoption of QUIC made substantial progress. All major browsers for all major platforms support QUIC and HTTP/3 by default, that is Google Chrome[1], Safari[2], Firefox[3] and their respective mobile apps. In October 2020, Facebook declared the successful transition of its applications, including Instagram, and server infrastructure to QUIC, with 75% of its Internet traffic already utilizing QUIC[4]. All mobile applications

[1]`https://www.zdnet.com/article/http-over-quic-to-be-renamed-http3/` - 2024-02-15

[2]`https://developer.apple.com/documentation/safari-release-notes/safari-14-release-notes` - 2024-02-15

[3]`https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox-nightly-and-beta/` - 2024-02-15

[4]`https://engineering.fb.com/2020/10/21/networking-traffic/how-facebook-is-bringing-quic-to-billio` - 2024-02-15

from Google, including YouTube and Gmail, are compatible with QUIC. Additionally there are a number of open-source implementations, include Cloudflares "quiche" and Amazons "s2n-quic" both of which are written in Rust. According to Cloudflare, the current rate of QUIC traffic worldwide is at about 30% [5].

3.2 Architecture

QUIC's architecture is designed to address the limitations of traditional transport protocols like TCP, offering improved performance, security, and flexibility. At its core, QUIC operates as a transport layer protocol, facilitating communication between endpoints over the unreliable transport protocol UDP. One of the key features of QUIC's architecture is its integration of transport and security layers, providing encryption and authentication by default. This helps safeguard data transmitted over the network, mitigating security threats such as eavesdropping and tampering. QUIC employs a connection-oriented model, establishing and maintaining connections between endpoints for reliable data transfer. Unlike TCP, QUIC connections are established more efficiently, requiring fewer round trips for setup. Additionally, QUIC supports multiplexed streams within a single connection, allowing for concurrent transmission of multiple data streams. This feature enhances efficiency and reduces latency by eliminating the need for multiple connections for different types of data. Another notable aspect of QUIC's architecture is its support for connection migration, enabling seamless transfer of connections between network interfaces or IP addresses without disrupting ongoing communication. Additionally, QUIC incorporates mechanisms for congestion control and flow control to optimize network performance and prevent network congestion. These mechanisms dynamically adjust transmission rates based on network conditions and receiver capabilities, ensuring efficient data transfer while minimizing the risk of congestion-related issues.

3.2.1 Multiplexing with Streams

Streams in QUIC offer a lightweight byte-wise transmission of data. They represent a continuous flow of bytes, with strict byte-level ordering within each stream. Streams can cater to various communication patterns. Unidirectional streams, akin to simplex connections, handle data transfer in one direction, like video streaming. Bidirectional streams, similar to full-duplex connections, enable interactive communication like web browsing where both sides are capable of receiving and sending data. Unlike TCP, one QUIC connection can serve as many streams as both endpoints negotiated on during the handshake. Streams are not pre-established entities. They are dynamically created upon data transmission or implicitly opened when data arrives, ensuring efficient resource utilization. Encapsulating stream information, STREAM frames contained within the QUIC payload function as the data packets of QUIC. A single frame can encompass stream opening, data payload, and closure, streamlining communication efficiency. To facilitate multiplexing, each stream is identified by a unique stream identifier. This identifier is held in the header of every stream frame and contains information about the stream initiator and its type. The least significant bit (0x01) of the stream ID designates the initiator of the stream while the second least significant bit (0x02) of the stream ID distinguishes

[5]`https://radar.cloudflare.com/adoption-and-usage` - 2024-02-14

between bidirectional streams (with the bit set to 0) and unidirectional streams (with the bit set to 1) as seen in Table 3.1.

Bits	Stream Type
0x00	Client-Initiated, Bidirectional
0x01	Server-Initiated, Bidirectional
0x02	Client-Initiated, Unidirectional
0x03	Server-Initiated, Unidirectional

Table 3.1: Stream ID Types [1, Tab. 1]

Streams are managed on a per-stream basis. QUIC guarantees in-order byte delivery within a stream. This might involve temporarily buffering out-of-order data until its rightful position arrives, ensuring reliable data sequencing. To prevent data overload, QUIC adheres to flow control (3.2.5) limits advertised by the receiver both on stream and connection level. This mechanism, separated by congestion control (3.2.6), ensures smooth data transmission without buffer overflows. In case of packet loss, retransmission of lost data is handled within QUIC, not UDP. However, QUIC does not provide a specific way to handle prioritization of streams, yet it states

> A QUIC implementation **SHOULD** provide ways in which an application can indicate the relative priority of streams.[1, p. 14]

Applications can interact with streams by either writing, reading or both, depending on its type. Streams can be either closed gracefully by marking the last STREAM frame with the FIN bit or abruptly by sending a RESET_STREAM frame in case of unexpected errors which results in immediate termination. When being on the receiving end of a stream, one can also request for stream to stop before it has finished transmitting by sending STOP_SENDING frame.

Stream States

The lifetime of streams can be represented by their respective state machines, one for receiving and one for sending. Bidirectional streams necessitate the utilization of both state machines at both endpoints. Generally, the state machine usage remains consistent regardless of unidirectional or bidirectional nature. The complexity arises when opening bidirectional streams, as initiating either the sending or receiving side automatically opens the stream in both directions. While informative, these state machines primarily serve as an illustrative tool for defining rules regarding frame transmission and reception based on specific states. Implementations are not bound by these specific states and can tailor their own as long as they adhere to the intended behavior.
The receiving part is initiated upon receiving the first stream frame type for a specific stream ID. For bidirectional streams initiated by the peer, receiving a MAX_STREAM_DATA or STOP_SENDING frame also triggers its creation. In the initial "Recv" state, an arbitrary amount of STREAM or STREAM_DATA_BLOCKED frames are handled. Incoming data gets buffered and reassembled for your application. As buffer space frees up with data consumption, MAX_STREAM_DATA frames signal the sender to continue transmitting. Receiving a STREAM frame with the FIN bit

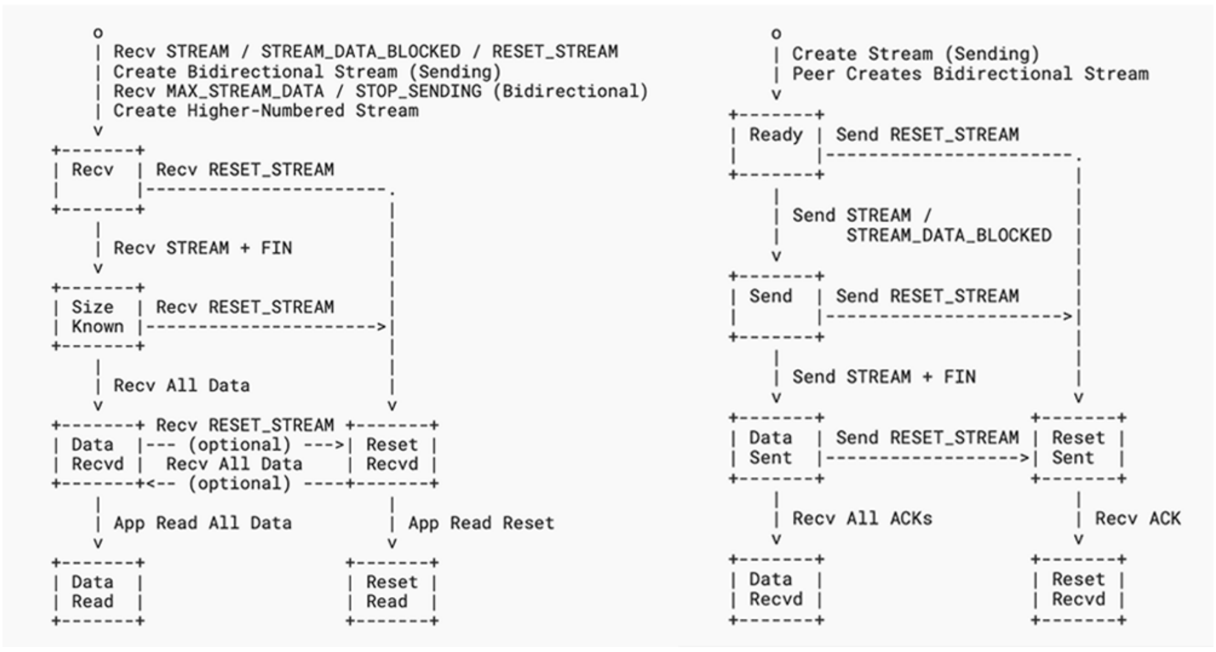

Figure 3.1: Stream States for Receiving (left) and Sending (right) [1, pp. 16–18]

marks the end of data flow, prompting the transition to the "Size Known" state. Here, MAX_STREAM_DATA frames are no longer sent, only potential retransmissions received. Once all data arrives, the stream enters the "Data Recvd" state. Any further STREAM or
STREAM_DATA_BLOCKED frames can be discarded at this point. From "Data Recvd", the application is able to read the stream data, promting a transition into the terminal "Data Read" state. Receiving a RESET_STREAM frame in the "Recv" or "Size Known" state transitions the stream to "Reset Recvd," potentially interrupting data delivery. Interestingly, implementations can handle this even if all data has arrived ("Data Recvd") or is still coming ("Reset Recvd"). A RESET_STREAM doesn't guarantee the sender won't deliver more data. The implementation can choose to:

1. Stop delivery, discard unconsumed data, and inform the application about the reset.
2. Suppress the reset signal if all data is received and buffered, keeping the stream in "Data Recvd."

Finally, receiving the reset notification by the application moves the stream to the terminal "Reset Read" state.

When initiating a QUIC stream, the sending part begins in the "Ready" state for both, streams initiated locally and bidirectional streams created by the peer, able to accept application data. This data might be buffered here before transmission. The first sent STREAM or STREAM_DATA_BLOCKED frame triggers a transition to the "Send" state. In the "Send" state, the endpoint transmits and retransmits data (STREAM frames) as needed, respecting flow control limits set by the peer and processing
MAX_STREAM_DATA frames. If blocked by flow control limits, the endpoint sends

STREAM_DATA_BLOCKED frames. After the application indicates all data is sent, and a STREAM frame with the FIN bit is sent, the stream enters the "Data Sent" state. Here, the endpoint only retransmits data as necessary and ignores flow control and blocked notifications.
MAX_STREAM_DATA frames might still be received until the final offset arrives. Once all data is acknowledged, the stream enters the terminal "Data Recvd" state. From "Ready," "Send," or "Data Sent," the application can signal abandonment or the endpoint can receive a STOP_SENDING frame, prompting a RESET_STREAM frame and transition to the "Reset Sent" state. Alternatively, a RESET_STREAM frame can be sent as the first stream mention, immediately opening and resetting the stream. Upon acknowledgment of the reset frame, the stream reaches the terminal "Reset Recvd" state.

3.2.2 Packets

QUIC endpoints communicate by exchanging packets. Packets are categorized by their use during connection establishment (Initial, Handshake, Retry) and data transfer (0-RTT, 1-RTT). Encryption levels vary based on packet type, ensuring confidentiality and integrity for sensitive data(3.2.4). Version Negotiation and Retry packets are the exception and lack encryption for initial communication as they do not carry sensitive data. Packets are distinguished by an unique identifier. This identifier can range from 0 to $2^{62} - 1$ and is encoded as variable sized integer(2.1) in headers for reduced overhead. During different stages of the connection process, packet numbers are counted separately, resulting in three packet number spaces: Initial, Handshake, and Application. Reusing numbers within a space is strictly prohibited to prevent ambiguity and maintain cryptographic separation. Certain packet types can be coalesced within a single UDP datagram for efficiency, especially while handshaking, but packets within a single datagram must be in increasing encryption level order for proper processing.
QUIC operates under the assumption that the minimum size of an IP packet is at least 1280 bytes, which aligns with the IPv6 standard and is commonly supported by modern IPv4 networks as well. Factoring in the minimum IP header size of 40 bytes for IPv6 and 20 bytes for IPv4, along with a UDP header size of 8 bytes, this yields a maximum datagram size of 1232 bytes for IPv6 and 1252 bytes for IPv4. Consequently, QUIC anticipates that both modern IPv4 and all IPv6 network paths can effectively accommodate its requirements. The minimum allowed packet size is 1200 bytes, therefore packets with not enough available payload must use padding frames to increase their packet size. The maximum packet size is defined as the largest size of UDP payload that can be sent across a network path using a single UDP datagram, but is ultimately governed by flow and congestion control.
A QUIC packet always consists of a header and a payload. Depending on the connection stage, the header structure changes accordingly to ensure optimal efficiency (3.2.3). The payload of a QUIC packet consists of individual frames, each with their own header, carrying various data types and instructions.

1. **PADDING** frames (type 0x00) carry no information and may be used to artificially inflate packet size, e.g. initial packets have to be padded to conform to the minimum packet size of 1200 bytes. They do not have to be acknowledged and packets that only contain padding may be used to probe new network paths. [1, p. 104]

2. **PING** frames (type 0x01) may be used to check reachability of the peer. [1, p. 105]
3. **ACK** frames (type 0x02 - 0x03) are used to acknowledge packets that have been received and processed. Type 0x02 carries one or more ACK ranges(2.1). If the type is 0x03, ACK frames also carry ECN(2.1) counts. They do not have to be acknowledged and packets only containing ACK frames do not count toward bytes in flight for congestion control. [1, p. 105]
4. **RESET_STREAM** frames (type 0x04) may be used to abruptly terminate the sending part of a stream. It also contains the application protocol error code. [1, p. 108]
5. **STOP_SENDING** frames (type 0x05) may be used to request that a peer ceases transmission on a stream. It also contains the application protocol error code. [1, p. 109]
6. **CRYPTO** frames (type 0x06) are used to transmit cryptographic handshake messages. They are functionally identical to stream frames except they do not have a stream identifier and they are not flow controlled. [1, p. 110]
7. **NEW_TOKEN** frames (type 0x07) contain a token, that is used in the header of an Initial packet for a future connection from the client. [1, p. 110]
8. **STREAM** frames (type 0x08 - 0x0f) carry stream data. The type is used to encode three different values: the OFF bit (0x04) is set to indicate that there is an Offset field present, the LEN bit (0x02) is set to indicate that there is a Length field present and the FIN bit (0x01) indicates that the frame marks the end of the stream.[1, p. 111]
9. **MAX_DATA** & **MAX_STREAM_DATA** frames (type 0x10 & 0x11) are used to inform the peer of the maximum amount of data that can be sent on the whole connection or a specific stream. [1, pp. 112, 113]
10. **MAX_STREAMS** frames (type 0x12 - 0x13) limits the number of concurrent streams the peer is allowed to open (type of 0x12 applies to bidirectional streams and a type of 0x13 applies to unidirectional streams). [1, p. 114]
11. **DATA_BLOCKED** & **STREAM_DATA_BLOCKED** frames (type 0x14 & 0x15) are used to inform the peer that due to connection-or stream-level flow control limits, no more data can be sent at the time. [1, p. 114]
12. **STREAMS_BLOCKED** frames (type 0x16 - 0x17) are used when a sender wishes to open a stream but is unable to do so due to the maximum stream limit set by its peer(0x16 for reaching the bidirectional stream limit, 0x17 for the unidirectional stream limit). [1, p. 115]
13. **NEW_CONNECTION_ID** & **RETIRE_CONNECTION_ID** frames (type 0x18 & 0x19) are used to inform the peer about new connection identifiers that may be used or to stop the peer from using specific identifiers in the future. [1, pp. 116, 117]

14. **PATH_CHALLENGE** & **PATH_RESPONSE** frames (type 0x1a & 0x1b) are used to check reachability of the peer and for path validation during connection migration. A response can only be sent after receiving a challenge. [1, pp. 118, 119]

15. **CONNECTION_CLOSE** frames (type 0x1c - 0x1d) are used to notify the peer that the connection is being closed. It also contains an error code and an optional reason phrase. [1, p. 119]

16. **HANDSHAKE_DONE** frames (type 0x1e) are used to signal confirmation of the handshake to the client. They do not carry any data.[1, p. 120]

Type Value	**Frame Type Name**	**Spec**
0x00	PADDING	NP
0x01	PING	
0x02 - 0x03	ACK	NC
0x04	RESET_STREAM	
0x05	STOP_SENDING	
0x06	CRYPTO	
0x07	NEW_TOKEN	
0x08 - 0x0f	STREAM	F
0x10	MAX_DATA	
0x11	MAX_STREAM_DATA	
0x12 - 0x13	MAX_STREAMS	
0x14	DATA_BLOCKED	
0x15	STREAM_DATA_BLOCKED	
0x16 - 0x17	STREAMS_BLOCKED	
0x18	NEW_CONNECTION_ID	P
0x19	RETIRE_CONNECTION_ID	
0x1a	PATH_CHALLENGE	P
0x1b	PATH_RESPONSE	P
0x1c - 0x1d	CONNECTION_CLOSE	N
0x1e	HANDSHAKE_DONE	

Table 3.2: Frame Types [1, pp. 70 – 71]

The "Spec" column in Table 3.2 provides a summary of any previously mentioned characteristics of the frame type, as indicated by the following symbols.

N: Packets containing only frames with this marking are not ack-eliciting

C: Packets containing only frames with this marking do not count toward bytes in flight for congestion control purposes

P: Packets containing only frames with this marking can be used to probe new network paths during connection migration

F: The contents of frames with this marking are flow controlled

3.2.3 Headers

In QUIC, packet headers play a crucial role in facilitating efficient communication between endpoints. They serve as the initial segment of each packet and contain key metadata necessary for proper packet processing and routing. QUIC packets can have either a short or a long header, each serving distinct purposes in the communication process. Packets with long headers are primarily used during the connection establishment or reestablishment phase and cryptographic handshake, while short header packets are utilized for ongoing communication sessions to reduce processing overhead.

Long Header

Long Header packets contain various fields essential for connection establishment, including the Source Connection ID, Destination Connection ID, Packet Number, and Packet Type. These fields provide information about the endpoints involved, the sequence number of the packet, and its type, facilitating proper routing and processing.

```
Long Header Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2),
  Type-Specific Bits (4),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Type-Specific Payload (..),
}
```

Figure 3.2: Base Long Header Format[1, p. 88]

Depending on the packet type, the type-specific payload changes. There are four different packet types that make use of the long header (Fig. 3.3) and which are encoded into the long packet type within the first byte of the long header. Additionally the version negotiation packet makes use of the long header, but is not explicitly encoded due to it being inherently not version specific.

Type	Name
0x00	Initial
0x01	0-RTT
0x02	Handshake
0x03	Retry

Table 3.3: Long Header Types [1, p. 90]

Version Negotiation Packets are identified by the version field having a value of 0. They are not acknowledged and only sent in response to a packet that indicates an unsupported version. Version Negotiation Packets also do not include the Packet Number and Length fields present in other packets that use the long header form. The payload of the Version Negotiation packet is a list of 32-bit versions that the server supports.

```
Version Negotiation Packet {
  Header Form (1) = 1,
  Unused (7),
  Version (32) = 0,
  Destination Connection ID Length (8),
  Destination Connection ID (0..2040),
  Source Connection ID Length (8),
  Source Connection ID (0..2040),
  Supported Version (32) ...,
}
```

Figure 3.3: Version Negotiation Packet[1, p. 91]

Initial Header Packets are sent by clients to a server and have their long packet type set to 0x00 (Fig. 3.3). The header contains additional fields for a token, the packet length and the packet number. A token may be used for address validation prior to completing the handshake and is derived from the source connection id which is randomly generated. The reserved bits, the packet number length, the length and the packet number are protected using header protection (3.2.4). The packet payload is encrypted separately. Initial packets carry crypto frames and ACKs in either direction. The crypto frames contain the initial client or server hello messages from TLS which are used for the key exchange[3].

```
Initial Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 0,
  Reserved Bits (2),
  Packet Number Length (2),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Token Length (i),
  Token (..),
  Length (i),
  Packet Number (8..32),
  Packet Payload (8..),
}
```

Figure 3.4: Initial Header Format[1, p. 92]

0-RTT Header Packets are used for instant connection resumption and have their long packet type set to 0x01 (Fig. 3.3). Keys for 0-RTT packets are derived during the handshake. 0-RTT packets carry early application data and can be processed if the peer recognizes the sender as a recently turned inactive connection. A handshake can then be carried out while application data is already being exchanged (Sec. 3.2.7).

```
0-RTT Packet {
    Header Form (1) = 1,
    Fixed Bit (1) = 1,
    Long Packet Type (2) = 1,
    Reserved Bits (2),
    Packet Number Length (2),
    Version (32),
    Destination Connection ID Length (8),
    Destination Connection ID (0..160),
    Source Connection ID Length (8),
    Source Connection ID (0..160),
    Length (i),
    Packet Number (8..32),
    Packet Payload (8..),
}
```

Figure 3.5: 0-RTT Header Format[1, p. 94]

Handshake Header Packets are used after the initial packets are exchanged and have their long packet type set to 0x02 (Fig. 3.3). Handshake packets use their own packet number space, are encrypted using the keys derived from the prior, initial key exchange and carry additional cryptographic handshake messages, including the server certificate and the encrypted TLS extensions[3].

```
Handshake Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 2,
  Reserved Bits (2),
  Packet Number Length (2),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Length (i),
  Packet Number (8..32),
  Packet Payload (8..),
}
```

Figure 3.6: Handshake Header Format[1, p. 95]

Retry Header Packets carry an address validation token created by the server and are used if a server wants to perform a retry. Retry packets do not contain any protected fields.

```
Retry Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2) = 3,
  Unused (4),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Retry Token (..),
  Retry Integrity Tag (128),
}
```

Figure 3.7: Retry Header Format[1, p. 96]

Short Header

Short Header packets, on the other hand, contain a reduced set of fields compared to Long Header packets, including only the Connection ID and Packet Number. This streamlined header format is employed for more efficient transmission during established communication sessions, reducing overhead and improving performance. Encrypted fields include the reserved bits, the key phase, the packet number length, the packet number and the packet payload. They are encrypted and decrypted using 1-RTT keys derived during the prior handshake.

```
1-RTT Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),
  Key Phase (1),
  Packet Number Length (2),
  Destination Connection ID (0..160),
  Packet Number (8..32),
  Packet Payload (8..),
}
```

Figure 3.8: Base Short Header Format[1, p. 98]

The latency spin bit facilitates passive monitoring of latency from various observation points along the network path throughout a connection's lifespan. Upon receiving the spin value, the server mirrors it, while the client toggles it after one round-trip time (RTT). By measuring the time elapsed between successive spin bit toggles, on-path observers can estimate the end-to-end RTT of the connection. The presence of the spin bit is confined to 1-RTT packets, as the initial RTT of a connection can be calculated by observing the handshake.

3.2.4 Header and Packet Protection

QUIC, similarly to TLS over TCP, uses keys derived from the TLS handshake to encrypt and decrypt packets, utilizing the AEAD algorithm negotiated by TLS. Depending on their type, QUIC packets employ varying levels of protection:

- **Version Negotiation Packets** lack cryptographic protection entirely.
- **Retry Packets** employ AEAD_AES_128_GCM (2.1) to prevent inadvertent tampering and restrict the entities capable of producing a valid Retry.
- **Initial Packets** utilize AEAD_AES_128_GCM with keys derived from the Destination Connection ID field of the client's initial Initial packet and a publicly available salt in RFC 9001 [3, p. 20].
- **Other Packets** boast robust cryptographic protections ensuring confidentiality and integrity, utilizing keys and algorithms negotiated by TLS.

QUIC employs separately derived keys for header and payload protection, ensuring confidentiality for specific header fields not exposed to on-path elements. Header protection extends to the least significant bits of the first byte and the Packet Number field. The header protection key remains constant throughout the connection, allowing header protection to be utilized continuously. Packet protection keys in QUIC are derived in a manner akin to TLS's derivation of record protection keys. Theses keys are computed from the TLS secrets using the Key Derivation Function (KDF) provided by TLS. For instance, in TLS 1.3, the Hash-based KDF-Expand-Label function is employed with a hash algorithm agreed on during the handshake and labels provided by the RFC [3, p. 20]. During 1-RTT communication, either endpoint may initiate a key update at any time (see sec. 3.5).

3.2.5 Flow Control

QUIC uses a limit-based flow control scheme to prevent buffer overflows and ensure smooth data delivery. This system operates on two levels:

1. **Stream Flow Control** manages individual streams, preventing any single stream from monopolizing the receiver's buffer. In turn, receivers advertise the maximum data they can accept for each stream and senders must adhere to these limits and stop sending when they reach the advertised amount.
2. **Connection Flow Control** provides an overall limit for the entire connection, ensuring the receiver's buffer doesn't become overloaded. Receivers advertise the total data they can accept for all streams combined and senders must adhere to this limit as well, distributing data across streams within the allowed quota.

A receiver must set initial limits for all streams through transport parameters during the handshake. Subsequently, a receiver sends MAX_STREAM_DATA frames or MAX_DATA frames to the sender to advertise larger limits. Flow control limits are dynamic, adapting to network conditions and buffer availability. Receivers can decrease the window size upon congestion or increase it when buffer space permits, ensuring efficient resource utilization.

3.2.6 Congestion Control

While QUIC's loss detection and congestion control algorithms share similarities with TCP, underlying protocol differences lead to some variations[4]:

- Clearer Loss Epochs: QUIC defines a loss epoch starting when a packet is lost and ending with the acknowledgement of any subsequent packet. This contrasts with TCP, where epochs might linger for multiple round-trips due to gaps in the sequence number space.
- No Reneging on Acknowledgements: QUIC acknowledgements, similar to TCP's Selective Acknowledgements (SACKs), are permanent, simplifying implementation and reducing memory usage on the sender side.
- Explicit Correction for Delayed Acknowledgements: QUIC accounts for delays between packet reception and acknowledgement, allowing for more accurate round-trip time (RTT) estimations.
- Probe Timeout (PTO) Replaces RTO and TLP: QUIC utilizes a PTO timer based on TCP's RTO concept but incorporates the peer's expected acknowledgement delay for a more precise timeout. Unlike TCP, QUIC doesn't collapse the congestion window upon PTO expiration for a single lost packet. This avoids unnecessary congestion control reductions.
- Minimum Congestion Window of Two Packets: QUIC recommends a minimum congestion window of two packets to avoid potential delays caused by waiting for a single packet acknowledgement after loss.
- Handshake Packets Treated as Regular Packets: QUIC handles lost handshake packets just like any other data loss, unlike TCP's approach of treating them as a sign of persistent congestion.

QUIC has two ways to detect packet loss, either by a missing acknowledgement or by timeout of the Probe Timeout (PTO) which is initialized whenever an ack-eliciting packet is sent (see Sec. 3.4.1).

Estimating Round Trip Times

An RTT sample is generated upon receiving an acknowledgement (ACK) frame that meets two criteria: First, it acknowledges the largest packet number not previously acknowledged. Second, at least one of the newly acknowledged packets triggered an acknowledgement (ack-eliciting frame). The RTT sample (`latest_rtt`) is calculated by measuring the time elapsed since the largest acknowledged packet was sent.

Additionally, the minimum round trip time (`min_rtt`) is tracked. The `min_rtt` represents the sender's observation of the absolute minimum RTT for the connection. It is initialized to the first received RTT sample (`latest_rtt`). Subsequent samples are compared to `min_rtt`, and the lower value is retained.
The Smoothed RTT (`smoothed_rtt`) represents an exponentially weighted moving average of RTT samples, providing a more stable estimate than individual samples. The RTT Variation `rttvar` reflects the degree of fluctuation in observed RTT samples.
While calculating the rtt times, a sample may be adjusted based on acknowledgement delays reported by the peer. Furthermore, during the handshake, QUIC might ignore peer-reported delays or limit them to a maximum value to avoid biasing the RTT estimate. Lastly, QUIC may account for local processing delays (e.g., decryption) when calculating RTT samples.

Algorithms

The sender-side Congestion Control Algorithm specified in RFC 9002[4, p. 17] shares similarities with TCP's NewReno[10] but presents only a suggestion. Implementations have the flexibility of choosing their own algorithm such as Cubic[11] or add improvements like HyStart++[12].

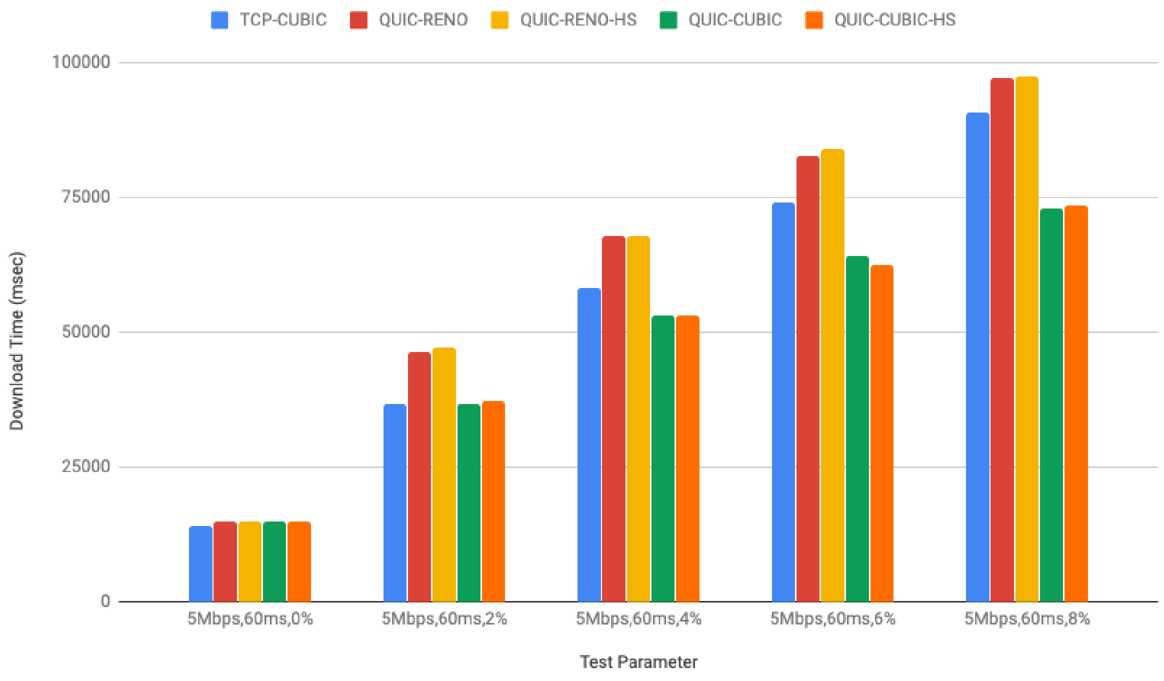

Figure 3.9: Comparison of different CC Algorithms (HS = HyStart++) [13]

QUIC uses a congestion window to limit the number of bytes in flight (unacknowledged packets) at any given time. A sender is not allowed transmit a packet that would exceed the congestion window which is adjusted dynamically based on network conditions. QUIC defines three congestion control states: Slow Start, Recovery, and Congestion Avoidance. Slow Start is used initially, the window grows exponentially to quickly probe the network capacity. The recovery is entered upon packet loss or ECN increase. The window is then reduced and slow start is re-initiated. During Congestion Avoidance, the window

is gradually increased using Additive Increase Multiplicative Decrease (AIMD) to avoid further congestion.
QUIC can detect persistent congestion when packet loss persists for a certain duration. Upon detection, the congestion window is drastically reduced to a minimum value similar to TCP's response on a timeout.
QUIC recommends pacing packets to avoid overwhelming the network with bursts. Pacing algorithms spread packets evenly over time based on the congestion window and RTT (Round Trip Time). ACK packets (acknowledgments) are exempt from pacing to ensure timely delivery. If the sender has data to send but the number of bytes in flight is less than the window (due to application limitations or flow control), the window won't be increased. A pacing mechanism might also delay sending packets, leading to underutilization. However, the sender shouldn't be considered application limited in this case.
Overall, QUIC's congestion control aims to achieve efficient data transfer by dynamically adjusting the sending rate based on network conditions while avoiding congestion collapse.

3.2.7 0-RTT Connection

The 0-RTT (Zero Round-Trip Time) connection handshake in QUIC is a feature designed to reduce latency by allowing clients to send data in the very first packet of a connection, without waiting for any round trips to complete. A server may choose to enable it and notifies the client in the initial cryptographic handshake. It is typically used when a client has previously established a connection with a server and has cached the session state, including cryptographic keys and other parameters, and wants to reestablish the connection, for example when loading a website the client has visited shortly before. This however implicates that clients and servers need to manage cached session data securely.
To initiate a 0-RTT connection, the client sends a packet with a long header of type 0x01 (3.3) and includes all cached session data from the previous connection. Upon receiving the client's initial packet, the server attempts to decrypt and validate the cached session data. If successful, it can immediately process and respond to the client's request, without requiring any additional round trips. To ensure security, the cached session data is encrypted using keys derived from the previous session's cryptographic handshake. If the server cannot successfully validate the cached session data (e.g., due to expiration, tampering, or other reasons), it falls back to a regular full handshake process, which involves additional round trips for establishing a new session.
By allowing the client to send data in the first packet, the 0-RTT handshake significantly reduces the connection establishment latency, leading to faster transmission of data. Additionally, minimizing round trips and accelerating the connection setup process improves network resource utilization and reduces congestion, benefiting both clients and servers. Allowing application data before properly securing the connection however introduces a potential security risk. Therefore applications may limit the data a 0-rtt can access, for example many HTTP/3 implementations only allow GET request prior to the completed 0-rtt handshake.

3.2.8 Connection Migration

As Quic was designed with todays technology in mind, a key feature was its ability to remove the reliance on a never changing network interface during communication. For

example, if a device from which a QUIC session has been established would leave the home WIFI network with his or her mobile phone and go outside, the mobile phone would log into the nearest Radio Access Node. A traditional TCP connection would collapse under these circumstances because the host address would change. Therefore the connection would have to be re-initiated from scratch, resulting in significant time and processing overhead. QUIC solves this problem by introducing a connection identifier which is not bound to any interface and only refers to the connection itself. Therefore a reconnecting host can still identify itself and resume the session on a new network path.
During connection migration, two processes play a crucial role:

(1) **Path Validation** assesses the quality, reliability, and suitability of a network path for communication. It verifies whether a new network path, including its latency, packet loss characteristics, and other properties, is suitable for transmitting data. Path validation often involves sending probing packets along the new network path and analyzing the responses to determine its characteristics and suitability for data transmission.[1, p. 46]

(2) **Address Validation** verifies the legitimacy and ownership of a peer's address. Address validation typically involves checking cryptographic properties or exchanging validation tokens to ensure that the IP address is genuinely associated with the intended peer. [1, p. 42]

If a QUIC connection is migrated to a new network interface, the new network path may get validated first through the use of probing packets. The migration is initiated with the first packet containing non-probing frames. If an endpoint receives a packet with a known connection identifier from an address which it does not recognize, it must perform address validation first before sending any data over the new path. If the address validation is successful, connection identifiers are renewed and congestion control and the round-trip time estimation are reset for the new path. QUICs Ack-based loss detection will detect and resend any packets that may have gotten lost during connection migration or due to packet loss in the network to achieve a graceful resumption of the previous connection.

Preferred Address

Servers can communicate a preferred address to clients by including it in the quic_transport_parameters extension inside the initial handshake messages allowing for migration to a more preferred address after the handshake. Clients select the preferred address, validate it, and initiate migration accordingly. This implies a setup in which a QUIC server functions as a load balancer, handling only handshakes and distributing incoming connections to other servers [6].

3.3 Connection

Having explored the complex architectural foundations of QUIC in the previous chapter, we now shift our focus to the heart of the protocol: the connection process. This critical process dictates how clients and servers initiate communication, negotiate security parameters, and ultimately exchange data securely. Unlike the traditional three-way handshake

[6] `https://github.com/envoyproxy/envoy/issues/12651`

of TCP, QUIC adopts a more streamlined and robust approach, incorporating TLS directly and allowing for connection migration or zero round-trip time (0-RTT) handshakes, where application data may be sent before the handshake is completed.
In traditional TCP with TLS (Transport Layer Security), the handshake typically spans three round-trip times (3-RTT) (Fig. 3.10). During this process, the client initiates the connection by sending a SYN packet to the server, which responds with a SYN-ACK packet. Upon receiving the SYN-ACK, the client sends an ACK packet, completing the handshake. Following this, TLS negotiation occurs, involving multiple exchanges of handshake messages to establish cryptographic parameters and keys for secure communication. While effective in ensuring security, the 3-RTT handshake introduces latency due to the multiple round trips required, impacting the connection's responsiveness, especially over high-latency networks.
In contrast, QUIC introduces a more streamlined approach to handshaking, aiming to minimize latency and improve performance. QUIC's 1-RTT handshake (Fig. 3.10) achieves this by combining the connection establishment and cryptographic negotiation into a single round trip. The client's initial packet contains cryptographic parameters and keys. Upon receiving this packet, the server decrypts the information, verifies the client's authenticity, and responds with its own cryptographic parameters and keys, allowing secure communication to commence. By consolidating these steps into a single round trip, QUIC significantly reduces handshake latency compared to TCP + TLS.
Moreover, QUIC introduces the even more efficient 0-RTT handshake, aimed at further minimizing latency for subsequent connections between known peers. In the 0-RTT handshake, a client caches cryptographic parameters and keys from a previous connection with the server. Upon reconnecting, the client is able to send encrypted application data within the first packet. If the server can validate the client's authenticity, it is able to resume the previous session, while redoing the handshake simultaneously, saving one round trip. This approach eliminates the need for the initial handshake rountrip, enabling instantaneous connection reestablishment. However, it comes with security considerations, as replay attacks and potential cryptographic vulnerabilities need to be addressed to ensure the integrity and confidentiality of the communication. In the following subchapters the different types of QUIC handshakes are presented in more detail.

3.3.1 1-RTT Handshake

If a client wishes to initiate a connection to a QUIC server, it starts by sending the initial packet. An initial packet has a long header of type 0x00 (3.3) and carries a source connection id chosen by the client and a randomly generated destination connection id. The payload contains a crypto frame which holds the TLS Client Hello and additional padding bytes to conform to the minimum packet size of 1200 bytes. The Client Hello follows the TLS 1.3 specification [14] but has been specially modified for QUIC in rfc 9001 [3]. The client offers a set of cipher suites, shares the public key, provides the used application layer protocol and specifies the clients transport parameters for QUIC. The application layer protocol chosen by the client is encoded as the UTF-8 representation of its name string. IANA keeps a list of registered and standardised protocol codes. Proprietary protocols may be used if the client and the server use the same encoded string. The transport parameters include various stream limits, the initial source connection id also used in the header, a stateless reset token and time limits for later acknowledgements.

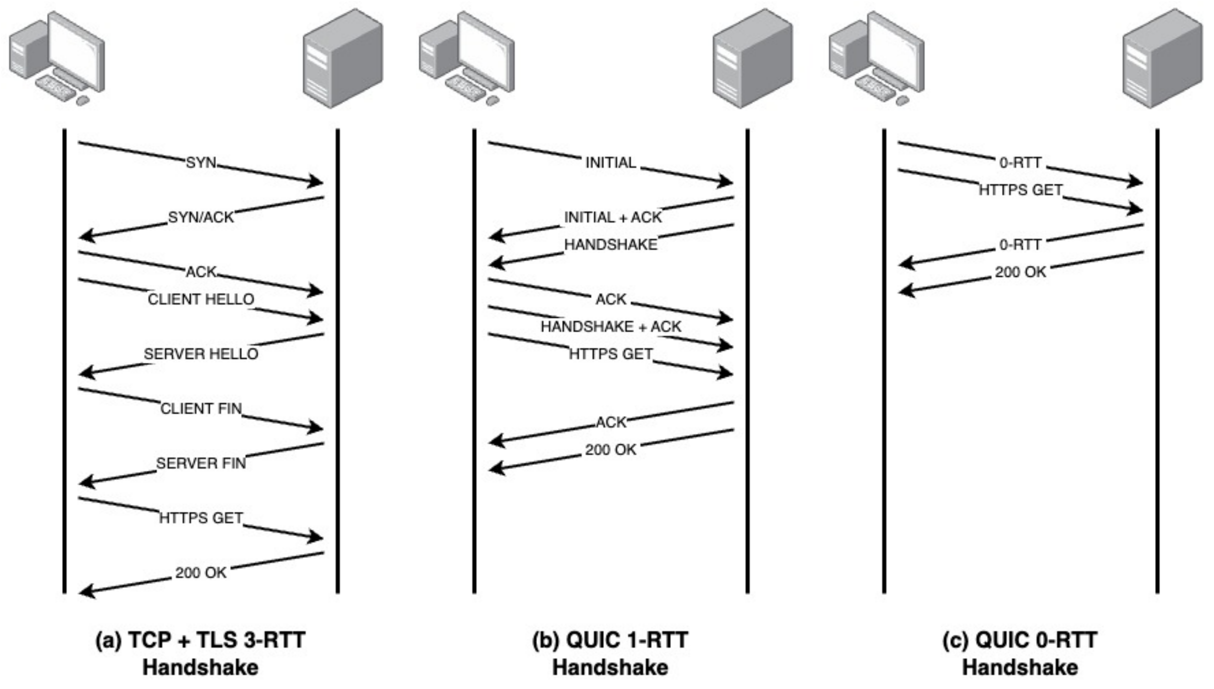

Figure 3.10: Comparison between handshakes

QUIC transport parameters are fixed and apply for possible retry and 0-rtt packets. The provided source connection ids from the transport parameters and the packet header must be identical to be deemed authenticated. Packet payloads from the initial packet number space are encrypted using a key derived from the randomly generated connection id. Header protection is always applied after packet protection because the first 16 bytes from the already encrypted payload are sampled and used as additional input for the encryption algorithm.

Upon receiving an initial packet from a client, the server first performs a partial header decode in which all unencrypted header data is read. Using the provided destination connection id and the public salt value from the rfc, the server can now generate the key pair to decrypt the packet. After decrypting the packet, the packet number and the payload can now be processed. The server can now authenticate the source connection id by comparing it to the source connection id provided in the QUIC transport parameters. It chooses one of the provided cipher suites, verifies that the specified application layer protocol is supported and prepares its own transport parameters. If the server does not support the specified protocol, the connection is immediately terminated. The servers transport parameters include all parameters it received from the client with the servers' values alongside the original destination connection id it received and a newly generated source connection id which replaces the original destination connection id (Fig.3.11). The server is now able to construct its answer which includes a crypto frame with the returning TLS Server Hello and an acknowledgement frame to confirm the successful receival of the initial packet. The Server Hello contains all the negotiated upon parameters and the servers public key.

When building the outgoing packet header, the received source connection ID becomes the destination connection ID, while the self-generated source connection ID replaces the provided destination connection ID. The packet and its header is then encrypted using the

```
Client                                                      Server

Initial: DCID=S1, SCID=C1 ->
                                 <- Initial: DCID=C1, SCID=S3
                             ...
1-RTT: DCID=S3 ->
                                            <- 1-RTT: DCID=C1
```

Figure 3.11: Use of connection ids during Handshake[1, p. 38]

previously derived initial keys associated with the original destination ID. The Server now has the option to consolidate multiple packets into a single UDP datagram, giving it the opportunity to directly provide the first handshake packet, drastically increasing efficiency. In order to do so, the server upgrades the packet number space to the handshake space. Subsequently, the new keys can be computed utilizing the data extracted from the client hello. A handshake packet features a long header of type 0x03 (3.3), and also carries crypto frames and acknowledgments, albeit with enhanced security measures. The server can now fill the first handshake packet with additional TLS data like the encryption extensions, the server certificate and the server certificate verify. The packet is then also encrypted and directly appended to the initial packet. It's imperative that coalesced packets maintain an ascending order in the packet number space to ensure that the necessary information is processed in the correct sequence.

```
Client                                                      Server

Initial[0]: CRYPTO[CH] ->

                                 Initial[0]: CRYPTO[SH] ACK[0]
                       Handshake[0]: CRYPTO[EE, CERT, CV, FIN]
                                 <- 1-RTT[0]: STREAM[1, "..."]

Initial[1]: ACK[0]
Handshake[0]: CRYPTO[FIN], ACK[0]
1-RTT[0]: STREAM[0, "..."], ACK[0] ->

                                         Handshake[1]: ACK[0]
           <- 1-RTT[1]: HANDSHAKE_DONE, STREAM[3, "..."], ACK[0]
```

Figure 3.12: Example 1-RTT Handshake[1, p. 35]

Figure 3.12 offers an overview of the previously described 1-RTT handshake process. Each line illustrates a QUIC packet, presenting the packet type and packet number followed by the contained frames. For example, the initial packet is categorized as "Initial," featuring packet number 0 and comprising a CRYPTO frame that carries the Client Hello. The complete handshake requires a minimum of four UDP datagrams (within the protocol's limitations, such as congestion control and flow control), using coalescing packets. For instance, the server's initial UDP datagram includes an initial packet, a handshake packet,

and possible "0.5-RTT data" enclosed within a fully protected 1-RTT packet.
The handshake completes when all initial and all handshake packets have been acknowledged by the opposing endpoint.

3.3.2 Transmitting Data

In an established QUIC connection, data is continuously exchanged between the client and server over the established connection. Each endpoint sends and receives application data using streams (3.2.1). Each packet is individually encrypted and authenticated using the negotiated keys, ensuring data integrity and security. During the life-span of the connection, both parties adjust their congestion control (3.2.6) parameters to optimize data transmission based on network conditions. This ensures efficient utilization of available bandwidth while helping to avoid congestion and packet loss.
In the event of packet loss, QUIC employs mechanisms such as retransmission and adjusts its congestion control algorithms to recover from the loss and maintain data integrity (3.4). Furthermore, the connection may adapt to changes in network conditions (3.2.5) or migrate to a new network interface (3.2.8).

3.3.3 Connection Termination

A connection can terminate in three different ways:

(1) **Idle Timeout:** If specified in the transport parameters, a connection is closed and its state discarded if it remains idle for longer than the minimum of the max_idle_timeout value negotiated upon by both endpoints. Endpoints restart their idle timers upon receiving and processing packets from their peers or when sending specific acknowledgments. [1, p. 57]

(2) **Immediate Close:** An endpoint can terminate the connection abruptly by sending a CONNECTION_CLOSE frame, causing all streams to immediately close. After sending this frame, the endpoint enters a closing state, while the receiving endpoint enters a draining state to ensure proper closure. [1, p. 58]

(3) **Stateless Reset:** In scenarios where an endpoint loses or is unable to confirm a connection state, it may send a stateless reset as a last resort. This involves sending a packet containing a reset token that, when received by the peer, prompts immediate termination of the connection. Stateless resets are used when an endpoint cannot properly continue the connection due to crashes or outages. [1, p. 61]

In addition to the primary methods of connection termination, there are several important procedures related to handling connection termination scenarios effectively.

(1) **Liveness Testing:** When a connection is at risk of timing out, endpoints can send PING frames or other acknowledgment-eliciting frames to test the connection for liveness. This helps ensure that connections are not erroneously terminated due to perceived inactivity.

(2) **Deferring Idle Timeout:** In situations where an endpoint expects response data but cannot send application data, there may be a need to defer the idle timeout.

This option allows the endpoint to periodically send PING frames to restart the idle timeout period, preventing premature closure of the connection while waiting for further activity.

(3) **Detecting and Preventing Looping:** Stateless Resets are designed to be indistinguishable from valid packets, but precautions must be taken to prevent looping scenarios. Endpoints should ensure that every Stateless Reset they send is smaller than the packet that triggered it, and limits on the number of Stateless Resets sent can prevent infinite exchanges that could disrupt network operations.

(4) **Stateless Reset Token Management:** Stateless reset tokens are used as a last resort for terminating connections when state is lost. Endpoints use pseudorandom functions with static keys and connection IDs to generate tokens, ensuring that they are unique and secure. It's crucial to manage these tokens carefully to prevent misuse or unintended consequences, such as denial-of-service attacks.

By following the specified procedures, QUIC implementations can manage connection termination scenarios effectively, ensuring the stability and reliability of the protocol in various network conditions.

3.4 Error Handling and Recovery

During an active connection, errors may occur at any given time. Error handling and recovery are critical aspects of any network protocol, ensuring reliability of communication in events such as packet loss, network congestion, and endpoint implementation failures. QUIC employs many error handling and recovery mechanisms to address potential issues and mitigate the impact of errors during data transmission, for example QUIC's loss detection mechanisms, which enable the protocol to identify and respond to packet loss. Additionally, QUIC can detect and handle numerous other errors, propagate error information between endpoints and handle exceptional conditions such as protocol violations and connection failures.

3.4.1 Loss Detection and Recovery

Like its predecessor TCP, QUIC uses an acknowledgement-based loss detection approach to ensure reliability while transferring data. Packets are identified by their sequentially increasing packet numbers. A packet must not be acknowledged until it has been successfully decrypted and all frames contained in the packet have been processed. A packet is ack-eliciting if it contains at least one other frame except acknowledgements itself, PADDING or CONNECTION_CLOSE. Each acknowledgment is sent inside an ACK frame, encoded either as type 0x02 or type 0x03, carry acknowledgment ranges to identify acknowledged packets. Additionally, type 0x03 ACK frames include information about Explicit Congestion Notification (ECN) feedback. The content of ACK frames, as depicted in Figure 3.13, contain the Type, Largest Acknowledged, ACK Delay, ACK Range Count, First ACK Range, ACK Ranges, and optionally ECN Counts.
Largest Acknowledged represents the highest packet number the peer is acknowledging. The ACK Delay field encodes the acknowledgment delay in microseconds. Moreover, ACK frames may contain an arbitrary number of additional ACK Ranges, which consist

```
ACK Frame {                          ACK Range {
  Type (i) = 0x02..0x03,                 Gap (i),
  Largest Acknowledged (i),              ACK Range Length (i),
  ACK Delay (i),                     }
  ACK Range Count (i),
  First ACK Range (i),
  ACK Range (..) ...,
  [ECN Counts (..)],
}
```

Figure 3.13: ACK Frame (left) [1, p. 106], ACK Range (right) [1, p. 107]

of alternating Gap and ACK Range Length values in descending packet number order. Each ACK range acknowledges a contiguous range of packets. The Gap field signifies the number of contiguous unacknowledged packets preceding the packet number one lower than the smallest in the preceding ACK Range. The ACK Range Length field indicates the number of contiguous acknowledged packets preceding the largest packet number. Furthermore, if the ACK frame type is 0x03, ECN feedback is included, providing insights into the number of packets received with specific ECN codepoints (ECT0, ECT1, ECN-CE) in the packet number space of the ACK frame. ECN counts are maintained separately for each packet number space. Figure 3.14 shows an example ACK frame for a provided number of successfully received packets.

```
Received packet numbers: [5, 6, 7, 8, 10, 11, 12, 13, 14, 17, 18, 19, 20]

ACK Frame {
  Type = 0x02,
  Largest Acknowledged = 20,
  ACK Delay (i),  //ommitted for this example
  ACK Range Count = 2,
  First ACK Range = 3,
  Gap = 2,
  Ack Range Length = 5,
  Gap = 1,
  Ack Range Length = 4,
}
```

Figure 3.14: Example ACK frame

An endpoint should attempt to include an ACK frame with every packet sent to aid in timely loss detection at the peer.

QUIC employs two types of thresholds for determining packet loss. First, packet number-based thresholds compare the sequence number of in-flight packets with the acknowledged packet. If the in-flight packet's sequence number is smaller than the acknowledged packet by a agreed upon threshold ($x - t$), it is marked as lost. Second, time-based thresholds assess the time elapsed since the acknowledged packet was sent. If an in-flight packet was sent before a certain time threshold ($t - t_0$), it is declared lost. These thresholds allow for some tolerance for packet reordering and aim to prevent unnecessary retransmissions without compromising congestion control performance.

To detect packet losses, QUIC initializes a Probe Timeout (PTO) timer whenever a packet requiring acknowledgment is sent. This timer incorporates factors such as smoothed network RTT, RTT variation, and maximum acknowledgment delay. When the PTO timer expires, the sender sends a new packet as a probe to elicit acknowledgments, potentially retransmitting some in-flight data to minimize redundant retransmissions.

QUIC packets that are determined to be lost are not retransmitted whole; instead, the information carried in lost packets is sent again in new frames which are packed into new outgoing packets, each assigned new packet numbers unrelated to the lost ones. Retransmitted data can still be contextualized by examining the offset parameter in the frame headers. This process ensures reliable ordered byte-stream delivery, akin to TCP's functionality, despite packet loss occurrences.

3.4.2 Protocol or Application Errors

One key aspect of QUIC's error handling is its reliance on error codes and connection states to communicate and manage errors effectively. When errors occur, QUIC endpoints may close the connection using the defined transport error codes, allowing the peer to understand the nature of the problem. These error codes cover a wide range of scenarios, including connection establishment failures, packet decryption errors, and protocol violations, providing comprehensive diagnostic information to facilitate error resolution. QUIC transport error codes can be used in a CONNECTION_CLOSE frame with a type of 0x1c.

Error Code	Name	Description
0x00	NO_ERROR	An endpoint uses this with CONNECTION_CLOSE to signal that the connection is being closed abruptly in the absence of any error.
0x01	INTERNAL_ERROR	The endpoint encountered an internal error and cannot continue with the connection.
0x02	CONNECTION_REFUSED	The server refused to accept a new connection.
0x03	FLOW_CONTROL_ERROR	An endpoint received more data than it permitted in its advertised data limits.
0x04	STREAM_LIMIT_ERROR	An endpoint received a frame for a stream identifier that exceeded its advertised stream limit for the corresponding stream type.
0x05	STREAM_STATE_ERROR	An endpoint received a frame for a stream that was not in a state that permitted that frame.
0x0a	PROTOCOL_VIOLATION	An endpoint detected an error with protocol compliance that was not covered by more specific error codes.
0x0c	APPLICATION_ERROR	The application or application protocol caused the connection to be closed.
0x0f	AEAD_LIMIT_REACHED	An endpoint has reached the confidentiality or integrity limit for the AEAD (2.1) algorithm used by the given connection.
0x10	NO_VIABLE_PATH	An endpoint has determined that the network path is incapable of supporting QUIC.

Table 3.4: A Selection of QUIC Transport Error Codes [1, p. 121]

In addition to handling protocol-level errors, QUIC also provides a way for applications to define and use their own error codes. This enables applications to gracefully handle errors such as resource exhaustion, application-layer protocol violations, or user-defined conditions, enhancing the overall robustness and resilience of QUIC-based applications. Application-defined error codes can be utilized for various purposes, including the RESET_STREAM frame, the STOP_SENDING frame, and the CONNECTION_CLOSE frame with a type of 0x1d (3.2).

3.5 Security

QUIC security is based on a threat model described in existing security considerations, addressing both passive and active attacks. Passive attackers can intercept packets, while active attackers can manipulate or inject packets into the network, potentially intercepting packet routes. Attackers are further categorized as on-path or off-path, depending on their ability to intercept and manipulate packets. On-path attackers have direct access to packet routes, enabling various forms of interference, including modification and packet dropping. Off-path attackers, though unable to directly manipulate packet routes, can still inject packets into the network. The handshake process in QUIC leverages TLS 1.3, inheriting its cryptographic properties and protections like continuously renewing keysets or different protection levels for different packet types. Any compromise in the TLS handshake could impact the security guarantees provided by QUIC. Moreover, QUIC incorporates defenses against denial-of-service (DoS) attacks during the handshake, such as address validation and server-side DoS prevention mechanisms. Packet protection in QUIC employs authenticated encryption, ensuring the integrity and confidentiality of transmitted data. While only on-path attackers can passively observe packets, active attacks are mitigated by cryptographic protection, preventing unauthorized modifications to packet payloads. Connection migration capabilities in QUIC allow endpoints to transition between network paths while maintaining security. Path validation mechanisms help mitigate address spoofing attacks, limiting the impact of packet interception and manipulation.

3.5.1 Security Considerations by QUIC

In its specification, QUIC lists several considerations which had an impact on its design, including known attacks and countermeasures[1]. These encompass but are not limited to the following:

- **Amplification Attacks** exploit an address validation token to spoof the same address for a 0-RTT connection. An attacker is then able to direct a server to send data towards a victim endpoint. Mitigation involves limiting token usage and lifetime.

- **Optimistic ACK Attacks** involve an endpoint acknowledging packets which it did not receive and can lead to the congestion controller permitting excessive sending rates. Endpoints may close the connection upon detecting this behavior.

- **Explicit Congestion Notification Attacks** are performed by manipulating ECN fields in the IP Header to influence the sender's rate of transmission. Therefore endpoints may ignore ECN fields until at least one QUIC packet from the IP packet is processed without errors.

- **Targeted Attacks by Routing** can be prevented by limiting the ability of an attacker to target a new connection to a particular server instance.

3.5.2 Tested Attacks

Attack Name	Type	Impact	Reference
Source-Address Token Replay Attack	Replay	Server DoS	[15]
Packet Manipulation Attack	Manipulation	Connection establishment failure / server load	[16] [15]
Crypto Stream Offset Attack	Manipulation	Connection establishment failure	[16]
Randomized Packet Fuzzing	Fuzzing	Possible server crash	[17]

Table 3.5: A selection of tested attacks against QUIC

Source-Address Token Replay Attack

The Source-Address Token Replay Attack exploits the vulnerability of the initial token, which is designed to prevent packet spoofing by verifying that a connection request originates from the claimed IP address. The token is generated by the server as part of the server reject message and contains encrypted information about the client's IP address and current time. In order to establish a 0-RTT connection, a client must provide a valid token in its ClientHello message before encryption is established. However, since the token must be presented prior to encryption, any attacker capable of intercepting network traffic can gather tokens, enabling them to spoof connection requests from a specific host for a limited time, typically 24 hours. This attack works by monitoring the network for ServerReject messages from the targeted server, each containing a new token sent to a client. Upon detecting a new token, the attacker captures it along with the public key and the client's IP address, then initiates repeated spoofed 0-RTT connection attempts using random connection IDs from the same client. To the target server, these spoofed requests appear legitimate, as the token is replayed from a genuine connection with an actual client at the forged IP address. Consequently, the server proceeds to establish a new connection, generating initial and forward secure encryption keys and sending a Server Hello message, under the impression that it has completed the connection establishment with the falsified client.

Packet Manipulation Attack

Unencrypted QUIC packets lack protection against adversarial tampering. If an attacker assumes the role of a Man in the Middle (MitM) by gaining access to the communication channel, they can manipulate unprotected parameters such as the connection identifier or source address token through fabrication. This manipulation can result in the client and server deriving different initial keys, causing the connection establishment to fail. Subsequently, the server prompts the client to renegotiate the connection establishment, but due to the protocol's low round-trip time, the adversary can once again manipulate packets, leading to an endless loop of connection establishment attempts for the client. Consequently, the client's quality of experience is significantly diminished, potentially resulting in connection abandonment. A potential solution would require a signature of all the unencrypted fields. Though that is not only possible but also a proven technique in preventing tampering, it incurs a significant computing overhead which could then again be taken advantage of in a Denial of Service attack.

Crypto Stream Offset Attack

In QUIC, all handshake messages constitute a continuous byte-stream. If a MiTM adversary gains access to the communication channel and inserts random data to disrupt this byte-stream, they can effectively disrupt the entire stream and halt the connection establishment process. Consequently the legitimate user is denied access to the intended web service and may have to revert to using TLS/TCP, resulting in increased RTT. In both scenarios, while the server's quality of service may remain unaffected, the client's quality of experience suffers a decline.

Randomized Packet Fuzzing

Fuzzing involves supplying programs with randomized inputs and observing how they behave in response. G.S. Reen[17] uses the self developed *DPIFuzz*, a stateless fuzzing framework, to test different open-source QUIC implementations by duplicating packet numbers or randomizing stream offsets and identifiers. Fuzzing attacks in general do not aim at gaining access to confidential data, but to produce undefined behaviour and exploit unchecked edge cases. Therefore this attack could only be used in an attack against a server as it requires full access to the decrypted packets. Nevertheless, it presents a considerable security issue because G.S. Reen was able to crash multiple QUIC implementations.

Chapter 4

Implementation

This chapter covers a implementation of the QUIC protocol in Rust as standalone program including its design and source code. Provided code examples may be simplified from the original implementation.

4.1 Design Decisions

The implementation of QUIC is written in Rust (v1.72.0). The decision had two main reasons.

(1) **Performance and Safety:** Rust offers exceptional performance and its built-in ownership system prevents memory leaks and data race conditions. This is highly beneficial for building efficient and less error prone programs. Furthermore, Rust's static typing guarantees memory safety at compile time, significantly reducing the risk of security vulnerabilities and crashes.

(2) **Rich Ecosystem and Ease of Use:** Rust is backed by a mature ecosystem of well-maintained libraries. These libraries integrate seamlessly into the project through Rust's included packet manager, Cargo. Additionally, compared to C/C++, Rust offers significant advantages in platform independence. Building and testing source-code is simpler and less error-prone thanks to Cargo and Rust's built-in testing frameworks.

The implementation works as standalone program, exposing just a server endpoint, and relies on the Rust standard library (std). The latter is essential, because the standard library provides a complete UDP socket and the possibility to utilise numerous external libraries (4.1).

4.1.1 Scope

A complete QUIC implementation presents a major challenge, considering the extent of QUIC and the time limitation of this thesis. Therefore only a subset of the QUIC specification is going to be implemented. This includes:

- Sending and receiving of QUIC packets through a server endpoint
- Decrypting and encrypting the header and payload of QUIC packets

- Parsing and processing of all header types and packet payload, mainly comprised of CRYPTO, ACK and STREAM frames
- Constructing packets, including acknowledgements for received packets and crypto data used in the handshake
- Accepting incoming connections, including a full TLS 1.3 1-RTT handshake
- Packet handling and management through all three packet number spaces

Due to the mentioned constraints of time and complexity, the following features could not be implemented:

- 0-RTT handshake
- Ack-based loss detection, congestion and flow control
- Version negotiation and retry packets

4.1.2 Program Layout

The program layout follows the modularization of logic into binaries encouraged by Rust into binaries, libraries and modules. The entry point of the program lies in the `main.rs` file which contains the main function. This file is compiled into a standalone binary executable.

```
project
   |- Cargo.toml
   |- Cargo.lock
   |- README.md
   |- src/
       |- main.rs
       |- quic/
           |- lib.rs
           |- packet.rs
           |- stream.rs
           |- [...]
```

Figure 4.1: Source code directory tree

The project leverages an internal library named `quic` located in the dedicated `src/quic` folder. Internal libraries encapsulate several modules, each contained in its own file, and can be used within the context of the overlying binary. A rust library can only be accessed through the `lib.rs` file, which exposes all public functionality through the dedicated namespace, in this case `quic::`.

This modular structure separates the main program logic (implemented in `main.rs`) from the reusable functionalities provided by the `quic` library. This increases code organization, reusability, and maintainability of the project.

4.1.3 Development Environment

The development environment for this project utilizes the Rust compiler version 1.72.0 on macOS 14.2 (Sonoma). However, due to Rust's inherent portability, the compiled binary can run on any platform that supports the mentioned Rust compiler version.

Testing

Testing the QUIC library and the overlying server binary has to be done in two different ways. Firstly, rusts built-in unit testing framework is used to test single functions of the library for their intended functionality. However, simply unit testing parts of the QUIC library does not offer the required level of testing coverage. To test for full protocol functionality, a complete, external QUIC implementation is needed which acts as client and simulates authentic packets with requests and answers. To achieve this, the open-source QUIC implementation quinn[1] is used which provides a simple client example out of the box which performs a HTTP/3 request using `quinn`. This setup has the additional benefit of locality, which means that client and server will connect only over the local host network and therefore packet loss and connection timeouts can be disregarded initially to simplify the implementation process.

External Libraries

Name	Version	Purpose	Reference
rustls	0.21.7	Full TLS 1.3 Implementation with API specifically for QUIC	[18]
octets	0.3.0	Zero-copy mutable byte buffer wrapper	[19]
rcgen	0.12.0	Generate self signed X.509 certificates	[20]
ring	0.17.7	Using HMAC to generate reset tokens from connection ids	[21]

Table 4.1: External Libraries

4.2 QUIC Library

The QUIC library is designed to work on two levels: the endpoint and the connection. An endpoint can manage multiple connections and handles incoming packets. Arriving packets are either matched to an existing connection and handed of, or in case of initial packets, decrypted and then handed of into a newly created connection. This saves the endpoint from having to retrieve the state of each connection and therefore allows for stateless packet handling in the endpoint. This is also made possible because the initial keying material is derived from the destination connection id encoded in the header of every initial packet. In case of an existing connection, packets are decrypted inside the connection as vastly more complex keying material is needed which is only kept inside the connection context. This allows for a clear distinction between contexts. After handling a packet, a connection emits an event. A server or client implementation may act upon

[1] `https://github.com/quinn-rs/quinn`

this event by either passing it into the endpoint again or by choosing another action, i.e. a server may choose to either ignore initial packets from clients or immediately close the connection if the endpoint is overloaded. In both ways, the endpoint retrieves the connection from the handle contained in the event and passes the event further down into the connection itself which then builds one or multiple, coalesced answer packets.

4.2.1 Endpoint

The endpoint structure holds the UDP socket object, an optional server config in case the endpoint assumes the role of a server, a randomly generated hmac reset key to generate reset tokens, a vector which stores the connections itself and a hash map which uses the connection id as key to store connection handles. A connection handle is a plain type alias for a `usize` and simply refers to the index of a connection inside the vector. This approach allows for more efficient and faster code because handles can easily be copied and passed around without having to copy the whole `Connection` object every time. Also through the inherent design of Rust it is impossible to just create an arbitrary number of pointers every time a connection is needed.

lib.rs (Endpoint) Rust

```rust
pub struct Endpoint {
    socket: UdpSocket,

    //server config for rustls
    server_config: Option<rustls::ServerConfig>,
    //RFC 2104, used to generate reset tokens from connection ids
    hmac_reset_key: ring::hmac::Key,

    //stores connection handles
    connections: Vec<Connection>,
    conn_db: HashMap<ConnectionId, Handle>,
}
```

4.2.2 Connection

The `Connection` struct encapsulates all required state information for a complete connection. Key Fields include the `side`, which is either `Server` or `Client`, the `remote` which stores the current IP address and port of the peer and the `state` which represents the current connection state. To keep track of packet statistics, the fields `recved`, `sent` and `lost` are incremented accordingly. The `tls_session` field encapsulates all functionality of TLS 1.3 and is imported from rustls (4.1). It is of type `RustlsConnection` used to process and generate handshake data, manage cryptographic material and check for any tls specific alerts or errors. In addition, the fields `next_secrets`, `prev_1rtt_keys`, `next_1rtt_keys`, `initial_keyset` and `zero_rtt_keyset`, are used to keep track of specific crypto material which is kept outside of the packet number spaces. The fields `initial_dcid`, `initial_remote_scid`, `initial_local_scid` and `retry_scid` hold all connection ids which need to be cached in order to allow for retry and 0-rtt packets. For example, `initial_dcid` is used in zero-rtt packets. All packet number spaces are kept inside `packet_spaces` (4.2.3) with the current space being held inside `current_space`.

Lastly, `zero_rtt_enabled` dictates if a connection allows a zero rtt handshake, potentially allowing faster connection establishment for subsequent connection reestablishment.

lib.rs (Connection) Rust

```rust
struct Connection {
    //side
    side: Side,

    //tls13 session via rustls and keying material
    tls_session: RustlsConnection,
    next_secrets: Option<rustls::quic::Secrets>,
    prev_1rtt_keys: Option<PacketKeySet>,
    next_1rtt_keys: Option<PacketKeySet>,
    initial_keyset: Keys,
    zero_rtt_keyset: Option<Keys>,

    state: ConnectionState,

    // First received dcid
    initial_dcid: ConnectionId,
    // First received scid
    initial_remote_scid: ConnectionId,
    // First generated scid after handshake receive
    initial_local_scid: ConnectionId,
    // Retry Source Connection Id
    retry_scid: Option<ConnectionId>,

    // Packet number spaces, inital, handshake, 1-RTT
    packet_spaces: [PacketSpace; 3],
    current_space: usize,

    // Physical address of connection peer
    remote: SocketAddr,

    // Packet stats
    recved: u64,
    sent: u64,
    lost: u64,

    //0-Rtt enabled
    zero_rtt_enabled: bool,
}
```

4.2.3 Packet Number Spaces

Each connection has three packet number spaces (initial, handshake, 1-rtt)(4.2.3). Each individual `PacketNumberSpace` keeps track of three aspects: The keyset specific to that packet number space, acknowledgements for in and outgoing packets and the next packet number. Additionally, two of the three packet number spaces, that is initial and handshake, have an outgoing crypto buffer, in which the Server Hello and subsequent TLS data is buffered before being sent to the peer.

lib.rs (PacketNumberSpace) Rust

```rust
pub struct PacketNumberSpace {
    keys: Option<Keys>,

    outgoing_acks: Vec<u64>,

    outgoing_crypto_data: Option<Vec<u8>>,

    max_acked_pkt: Option<u64>,

    next_pkt_num: u64,
}
```

4.2.4 Connection ID

Each connection id is represented by the respective `ConnectionId` struct. It merely contains a byte vector and derives from Eq, Hash, PartialEq and Clone to provide basic functionality like comparison to other connection ids without having to explicitly implement each method seperately. Additionally `ConnectionId` implements a method to retrieve its length (`len(&self) -> usize`) and another to get an immutable reference `as_slice(&self) -> &[u8]` to the underlying data.

lib.rs (ConnectionId) Rust

```rust
#[derive(Eq, Hash, PartialEq, Clone)]
pub struct ConnectionId {
    id: Vec<u8>,
}
```

A `ConnectionId` can be created by providing raw byte data or as randomized id with a specified length.

lib.rs (impl ConnectionId) Rust

```rust
pub fn generate_with_length(length: usize) -> Self {
    assert!(length <= MAX_CID_SIZE);
    let mut b = [0u8; MAX_CID_SIZE];
    rand::thread_rng().fill_bytes(&mut b[..length]);
    ConnectionId::from_vec(b[..length].into())
}

#[inline]
pub const fn from_vec(cid: Vec<u8>) -> Self {
    Self { id: cid }
}
```

4.3 Packets

A packet on its own does not have a dedicated struct. Instead a buffer is declared once as a fixed sized array for every UDP datagram received and only the header and individual frames are constructed as seperate structs. Through the use of references and `OctetsMut` objects, which is a small zero-copy wrapper, we avoid expensive copying while still maintaining 100% memory safety. The stack buffer has a fixed size of 65536 bytes, which is the maximum UDP datagram size. The advantage of this approach is that no dynamic memory allocation on the heap is needed. However, always using the maximum buffer size may implicate a major overhead in memory use which could become a concern should the server go into the thousands of open connections. Considering a fairly modern system with at least 16GB of RAM, one can simply calculate that the CPU would still be the first bottleneck by a long shot, for example if only a fourth of the systems memory would be used to store packets, the systems memory could still handle over 60,000 packets at any given time. Therefore the "lazy" approach has a significant advantage over the dynamic approach in that it offers superior performance and increased simplicity over minor memory efficiency gains.

4.3.1 Header

The `Header` struct contains all fields both long and short headers contain and is therfore used throughout the whole library. The first header byte (`hf` = header form), which also includes the header type and packet specific bits, is saved without modification. Individual values may be extracted through bitshifting using the provided helper functions inside the `Header` implementation. Other fields include `version`, `dcid`, `scid`, `packet_num`, `packet_num_length`, `token`, and `packet_length`, exactly as specified in the RFC.

packet.rs (Header) Rust

```rust
pub struct Header {
    hf: u8, //header form and version specific bits
    version: u32,
    dcid: ConnectionId,
    scid: Option<ConnectionId>,
    token: Option<Vec<u8>>,

    //The following fields are under header protection
    packet_num: u32,
    packet_num_length: u8,

    //packet length including packet_num
    packet_length: usize,
}
```

The implementation of `Header` provides important functionality to decode and decrypt a header and vice versa. The function `parse_from_bytes()` (4.3.1) takes a mutable reference to a `OctetsMut` object, which wraps the raw, received buffer, and a destination connection id length in case the received packet has a short header as they do not declare the length of the connection id in their header. Connection identifier length is set for a connection with the inital packet received from the peer. The function returns either the decoded header inside a `Header` object or an `BufferTooShortError` which is provided by the `octets` crate. It is used to perform a partial decode of the header, that is parsing all fields that are not protected by QUICs partial header protection. To fill in the remaining fields, one has to call `decrypt()` (4.3.1) which takes a mutable reference to itself and a `OctetsMut` object and a reference to a `HeaderProtectionKey`. It returns just an error, should one occur, and utilizes `decrypt_in_place()` by rustls to decrypt the header and then fills in the remaining fields `packet_num_length` and `packet_num`. The header form bit is also overwritten as it contains encrypted fields in its original state.

Parsing and Decrypting

packet.rs (Header) Rust

```rust
pub fn parse_from_bytes(
    b: &mut octets::OctetsMut,
    dcid_len: usize,
) -> Result<Header, BufferTooShortError> {
    //get header form as raw byte
    let hf = b.get_u8()?;

    if ((hf & LS_TYPE_BIT) >> 7) == 0 {
        [...] //short packet
    }

    let version = b.get_u32()?;

    let dcid_length = b.get_u8()?;
    let dcid = b.get_bytes(dcid_length as usize)?.to_vec();

    let scid_length = b.get_u8()?;
    let scid = b.get_bytes(scid_length as usize)?.to_vec();

    let mut tok: Option<Vec<u8>> = None;

    match (hf & TYPE_MASK) >> 4 {
        0x00 => {
            // Initial
            tok = Some(b.get_bytes_with_varint_length()?.to_vec());
        }
        [...] //Zero-RTT, Handshake, Retry
        _ => panic!("Fatal Error with packet type"),
    }

    let pkt_length = b.get_varint()?;

    [...]
}
```

The function `parse_from_bytes()` (4.3.1), shown slightly simplified, performs a partial decode of an encrypted header. The presented version shows all necessary code to decode an initial packet. The `octets` library enables an easy and readable way to step through a byte buffer, i.e. the function `get_bytes_with_varint_length()` decodes a variably encoded integer from the current offset and then gets that amount of bytes as a slice `&[u8]` and `to_vec()` then transforms the data into a vector.

The `decrypt()` (4.3.1) function presents a greater challenge, because the protected fields are not ordered and occur at the beginning and the end of the header. When calling `decrypt()` the current pointer inside the buffer points at the start of the packet number field located at the end of the header which is encrypted. However the length of the packet number is unknown as it is also encrypted inside the first byte of the header. Additionally, when decrypting, we need the first 16 bytes of the payload because they are used as sample input. Therefore the buffer is first split after the maximum packet number length (four bytes) and the sample length (16 bytes). Then `decrypt_in_place()` is called on the `HeaderProtectionKey` and the previously separated sample, the header form byte and the encrypted packet number are passed as paramters. As soon as the header form byte is decrypted, the packet number length can be extracted and subsequently the packet number can be read.

packet.rs (Header) Rust

```rust
pub fn decrypt(
    &mut self,
    b: &mut octets::OctetsMut,
    header_key: &HeaderProtectionKey,
) -> Result<(), BufferTooShortError> {
    // get packet number and sampled payload cipher
    let mut pn_and_sample = b.peek_bytes_mut(MAX_PKT_NUM_LEN + SAMPLE_LEN)?;
    let (mut pn_cipher, sample) = pn_and_sample.split_at(MAX_PKT_NUM_LEN)?;

    match header_key.decrypt_in_place(sample.as_ref(), &mut self.hf,
      pn_cipher.as_mut()) {
        Ok(_) => (),
        Err(error) => panic!("Error decrypting header: {}", error),
    }

    self.packet_num_length = usize::from((self.hf & PKT_NUM_LENGTH_MASK) + 1);

    self.length += self.packet_num_length;

    self.packet_num = match self.packet_num_length {
        1 => u32::from(b.get_u8()?),
        2 => u32::from(b.get_u16()?),
        3 => b.get_u24()?,
        4 => b.get_u32()?,
        _ => return Err(BufferTooShortError),
    };

    Ok(())
}
```

4.3.2 Payload

A packet's payload exclusively contains an arbitrary number of frames (3.2). To provide as much flexibility as possible while also considering code and library structure, I chose to utilise rusts `trait` feature. A `trait` can be applied to any struct and forces the implementation of the specified methods. It is similar to an `Interface` in Java. In this case, the `Frame` trait enforces the implementation of three functions: `from_bytes()`, `to_bytes()` and `len()`. These functions provide the basic functionality to read and write frames from a buffer. Most of the frames specified in Table 3.2 have a dedicated struct, for example the `CRYPTO` and `ACK` frames. The exception are frames which only consist of one to two data fields and therefore can be trivially read and written from and into a buffer, for example `PATH_CHALLENGE`, `MAX_STREAMS` and `MAX_STREAM_DATA` frames.

packet.rs — Rust

```rust
pub trait Frame {
    fn from_bytes(frame_code: &u8, bytes: &mut octets::OctetsMut<'_>) -> Self;

    fn to_bytes(&self, bytes: &mut octets::OctetsMut<'_>);

    fn len(&self)
}
```

A frame which has the `Frame` trait enforced, is the `CRYPTO` frame. As per specification [1, p. 110], it has the type field which is set to 0x06, an offset, a length and the actual data. The type field can be ommitted, as well as the length field because the data is copied into a vector of which we can directly get the length using the `len()` function. Writing the same frame to a buffer therefore presents only a trivial problem. All other frames with dedicated structs follow the same pattern.

packet.rs (CRYPTO frame) — Rust

```rust
pub struct CryptoFrame {
    offset: u64,
    crypto_data: Vec<u8>,
}
```

packet.rs (CRYPTO frame) — Rust

```rust
impl Frame for CryptoFrame {
    fn from_bytes(frame_code: &u8, bytes: &mut octets::OctetsMut<'_>) -> Self {
        let offset = bytes.get_varint().unwrap();

        CryptoFrame {
            offset,
            crypto_data: bytes.get_bytes_with_varint_length().unwrap().to_vec(),
        }
    }

    [...] //to_bytes(), len()
}
```

4.3.3 Events

Events are omitted by a connection or the endpoint when a new packet arrives and may be acted upon in the application implementation. Each variation contains the according connection handle so that any event can be associated with the correct connection.

lib.rs (Event) Rust

```rust
pub enum Event {
    NewConnection(Handle),
    Handshaking(Handle),
    ConnectionEstablished(Handle),
    DataExchange(Handle),
    ConnectionClosed(Handle),
}
```

4.4 Connection

In this chapter the focus lies on an examplary packet and how it is processed within the library to gain a better understanding of the libraries inner design. Incoming packets are always received in the `recv()` function inside the endpoint object, as the UDP socket is not accessible from outside the endpoint. Immediately after that a partial header decode is performed, yielding an incomplete `Header` struct. The destination connection id is then used to query for an existing connection. If the latter yields a valid result, the packet and its partially decoded header are being passed down to the corresponding connection and the `recv()` function inside the connection is called. In this case the endpoint only returns the resulting event and any further processing is being performed within the context of the connection (4.4.2).

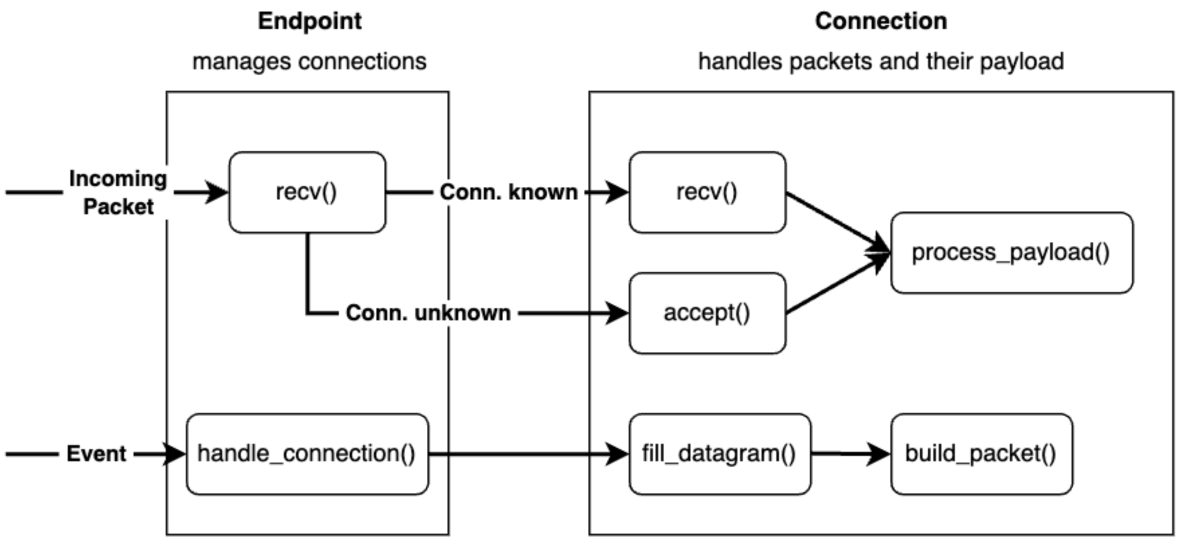

Figure 4.2: Rough Library Outline of Data Flow and Event Handling

4.4.1 Initial packets

If the query yields no valid result however, the packet is considered to be of the initial type. Any other type results in an error and the whole packet will be discarded. The endpoint now has to perform a number of steps to initialize the new connection and create all necessary configurations. First, the initial keys are derived using rustls:

```rust
let ikp = Keys::initial(Version::V1, &head.dcid.id, Side::Server);
```

With the resulting keyset, the header is decrypted (4.3.1) and the packet number, as well as the packet length fields are populated with the decrypted values. Through using the `split()` fucntion of the `OctetsMut` object we are now able to isolate the `payload_cipher` and pass it to rustls with the correct key to gain access to the decrypted frames contained within the packet.

lib.rs Rust

```rust
let dec_len = {
    let decrypted_payload_raw = match ikp.remote.packet.decrypt_in_place(
        head.packet_num.into(),
        header_raw.as_ref(),
        payload_cipher.as_mut(),
    ) {
        Ok(p) => p,
        Err(error) => panic!("Error decrypting packet body {}", error),
    };
    decrypted_payload_raw.len()
};
```

When decrypting the payload, the decrypted data is always smaller than the cipher text. That is why `decrypt_in_place()` returns only a part of the `payload_cipher` as a slice.

After that, we have to initialize the QUIC transport parameters of our endpoint. Most of these parameters are set with default values except for the original destination connection id(1), the initial source connection id(2) and the stateless reset token(3). (1) can be obtained trivially from the decoded header, (2) has to be generated (3.11) randomly using 4.2.4 and (3) is generated using the hmac reset key and the previsouly generated initial source connection id.

lib.rs Rust

```rust
let mut transport_config = transport_parameters::TransportConfig::default();
transport_config
    .original_destination_connection_id(orig_dcid.id()) // (1)
    .initial_source_connection_id(initial_local_scid.id()) // (2)
    .stateless_reset_token(
        token::StatelessResetToken::new(&self.hmac_reset_key, &initial_local_scid)
            .token
            .to_vec(), // (3)
    );
```

With the transport configuration parameters being set, the tls session of type `RustlsConnection::Server` can be initialized. In order to so it needs an `Arc` (a thread-safe reference-counting pointer) to the server config, the QUIC version being used and the encoded transport parameters.

lib.rs Rust

```rust
let conn = RustlsConnection::Server(
    rustls::quic::ServerConnection::new(
        std::sync::Arc::new(self.server_config.as_ref().unwrap().clone()),
        rustls::quic::Version::V1,
        transport_config_data.to_vec(),
    )
    .unwrap(),
);
```

After the tls session is initialized, a `Connection` object is created and inserted into the endpoints connection vector. Additionally the new connection handle is created and can now be used to retrieve that connection. The packet and its header are then being passed into that new connection to the `accept()` function.

lib.rs Rust

```rust
pub fn accept(
    &mut self,
    header: &Header,
    payload: &mut OctetsMut<'_>,
) -> Result<(), quic_error::Error> {
    self.process_payload(header, payload);

    self.generate_crypto_data();

    Ok(())
}
```

This function starts by processing the packet's payload (4.4.3), which contains a CRYPTO frame which in turn contains the Client Hello message and generates the returning crypto data, in this case the Server Hello message using the tls session and the function `write_hs()`. After returning from the connection, the endpoint returns a `Event` of type `NewConnection` which contains the new connection handle.

4.4.2 Known Connection

If the incoming packet can be matched to an existing connection it is immediately passed into the corresponding object to the `recv()` function. By using the `current_space` identifier, the correct `PacketNumberSpace` is retrieved. Within that object is the current keyset with which the header and payload are decrypted. The decrypted payload is then passed on to `process_payload()`.

4.4.3 Processing Payload

To process a QUIC packets payload, every frame has to be decoded and processed in order. Firstly, the function checks for every frame if it is ack eliciting.
`ack_eliciting = !matches!(frame_code, 0x00 | 0x02 | 0x03 | 0x1c | 0x1d);`
If the frame code is anything else than `PADDING`, `ACK`, and `CONNECTION_CLOSE`, the packet requires an acknowledgement to be sent which is checked after the whole payload has been processed. The function then loops over the buffer with each iteration extracting a single frame until all frames have been parsed.

```rust
lib.rs                                                              Rust
match frame_code {
    0x00 => continue, //PADDING
    0x02 | 0x03 => {
        let _ack = AckFrame::from_bytes(&frame_code, payload);
    } //ACK
    0x05 => {
        let _stream_id = payload.get_varint().unwrap();
        let _application_protocol_error_code = payload.get_varint().unwrap();
    } //STOP_SENDING
    0x06 => {
        let crypto_frame = CryptoFrame::from_bytes(&frame_code, payload);
        self.process_crypto_data(&crypto_frame);
    } //CRYPTO
    [...] //all other frames
    _ => eprintln!(
        "Error while processing frames: unrecognised frame {:#x} at {:#x}",
        frame_code,
        payload.off()
    ),
}
```

Depending on the frame, a dedicated struct may be created except for frames which contain so few fields, that a dedicated struct is not necessary, for example the `STOP_SENDING` frame which only contains two fields. The resulting data may then be processed accordingly. Crypto data is passed to `process_crypto_data()` which calls `read_hs()` provided by the `RustlsConnection` object. This function consumes the crypto data and prepares the according answer which is extracted using `write_hs()` after the payload has been processed in `accept()`.

4.4.4 Event Handling

The endpoint function `recv()` returns an `Event` which can then be acted upon in the applications interest. The application may also choose to emit another event, such as `ConnectionClose`.

```rust
main.rs                                                             Rust
let event = endpoint.recv();
match event {
    Ok(event) => match event {
        Event::NewConnection(ch) => {
            //maybe check if endpoint can handle more connections
            let _ = endpoint.handle_connection(ch);
        }
        //[...] other events
        _ => (),
    },
    Err(error) => eprint!("{}", error),
}
```

It is then passed into the endpoint again, which takes the connection handle contained within, extracts the connection and calls `fill_datagram()`. Said method returns the length up to which the buffer has been written to. The `packet_length` is then used to slice the buffer and send the packet to the current remote address.

lib.rs (handle_connection()) Rust

```rust
let connection = match self.connections.get_mut(connection_handle);

let mut buffer = [0u8; 65535];

let packet_length = connection.fill_datagram(&mut buffer).unwrap();

let size = match self
    .socket
    .send_to(&buffer[..packet_length], connection.remote);
```

4.4.5 Constructing Packets

One UDP datagram may be filled with multiple, coalesced QUIC packets. If so, packets have to be ordered by their packet number space with the lowest being the first. The function `fill_datagram()` loops through every packet number space and assembles a datagram with at least one packet.

lib.rs (fill_datagram()) Rust

```rust
let mut size: usize = 0;

for space_id in 0..(self.current_space + 1) {
    size += self.build_packet_in_space(&mut buffer[size..], space_id)?;
}
```

For every `PacketNumberSpace` the function `build_packet_in_space()` is called which creates and encodes the correct header, then populates the payload with a call to `populate_packet_payload()` and then encrypts the payload and the header. It also checks numerous edge cases, for example if the packet conforms to the minimum size of 1200 bytes. The function `populate_packet_payload()` only checks for available data for different frame types and encodes these.

lib.rs (populate_packet_payload()) Rust

```rust
let mut size = 0;

//CRYPTO
if let Some(crypto_buffer) =
  &self.packet_spaces[packet_number_space].outgoing_crypto_data {
    if !crypto_buffer.is_empty() {
        size += match CryptoFrame::new(0, crypto_buffer.to_vec()).to_bytes(buf);
    }
}

//[...] other frame types, i.e. ACK, STREAM, ...
```

4.5 Error Handling

Errors are handled on three different levels.

(1) **QUIC Errors** as defined in Sec. 20 [1], including possible TLS errors, result in the appropriate error code being sent via a CONNECTION_CLOSE frame for example. All Error types except TLS errors are represented by an enum 4.5. TLS erros range from 0x0100 to 0x01ff and are handled outside of the enum.

(2) **Application Errors** that result in connection termination may be thrown by any application via an event and result in a `CONNECTION_CLOSE` frame with type 0x1d which holds the application defined error code.

(3) **Implementation Errors** are handled within the library, most often through the `Result<(), Error>` return type and cover all possible scenarios. A self-defined `Error` contains a code and a reason. Error codes are consistent across the whole library and defined in `quic_error.rs`. For easier implementation a macro is used. 4.5

quic_error.rs Rust

```rust
enum QuicTransportErrors {
    NoError = 0x00,
    InternalError = 0x01,
    ConnectionRefused = 0x02,
    FlowControlError = 0x03,
    StreamLimitError = 0x04,
    StreamStateError = 0x05,
    // [...] 0x06 to 0x10
}
```

quic_error.rs Rust

```rust
#[derive(Debug)]
pub struct Error {
    code: u64,
    msg: String,
}

macro_rules! quic_error {
    ($name:ident, $code:expr) => {
        pub fn $name<T>(reason: T) -> Self
        where
            T: Into<String>,
        {
            Self {
                code: $code,
                msg: reason.into(),
            }
        }
    };
}

impl Error {
    quic_error!(fatal, 0x00);
    quic_error!(unknown_connection, 0x01);
    quic_error!(socket_error, 0x02);
    [...] //Other Errors
}
```

Chapter 5

Evaluation

Throughout the process of writing this thesis, the QUIC library accumulated 1855 lines of code (as of the submit date). All major goals that have been set in advance to this thesis have been achieved, except for one. The QUIC library is able to receive QUIC packets, parse, decrypt, process, and encrypt header and payload data through all three packet number spaces, and construct answer packets. The only exception is the full TLS 1.3 1-RTT handshake. Due to numerous challenges (5.2) and the short time frame the full handshake has narrowly not been achieved. Additionally the original objective to test the implementation with `aioquic` had to be adjusted to instead use `quinn`.

Using Rust as only programming language proved to be a valuable lesson. Rusts concepts of ownership, borrowing and lifetimes dictate a special style of programming that prevents handling of raw pointers and hence massively complicates the manipulation of data through references outside of the scope of the data the pointers are referring to. Keeping references in structs for example is notoriously difficult due to Rusts lifetime annotations. While all these measures have resulted in not a single segmentation fault being thrown for the whole development phase, it gravely impacted the library design. One of those being that the `recv()` and `accept()` functions from the `Connection` object always return into the endpoint to minimise the amount of references having to be kept. Additionally, Rusts effort to improve the readability of compiler error messages greatly accelerated debugging.

While one might think that the limitations of Rust may limit performance, the `quiche`[1] implementation by cloudflare, also developed in Rust, can even outperform `picoquic`[2], an implementation in pure C[3].

5.1 Execution

Rust and Cargo should enable fully functionaly cross-platform compilation as long as `rustc` version `1.72.0` or higher is used. To test with an external client, clone the external library `quinn` first.

[1] https://github.com/cloudflare/quiche

[2] https://github.com/private-octopus/picoquic

[3] https://www.diva-portal.org/smash/get/diva2:1691838/FULLTEXT03

```
[user@computer thesis]$ git clone https://github.com/quinn-rs/quinn
...
[user@computer thesis]$ cd quinn
```

Open a second terminal and navigate to the `project` folder inside the thesis directory. Start the server by executing `cargo run`.

```
[user@computer thesis] ~/thesis/project $ cargo run
```

Switch back to the first terminal and start the client connection by executing the following command.

```
[user@computer thesis] ~/quinn $ cargo run --example client
  https://127.0.0.1:34254/Cargo.toml
```

The second terminal window now shows the librarys debug statements.

```
[user@computer thesis] ~/thesis/project $ cargo run
Received 1200 bytes from 127.0.0.1:63844
0x00 version: 0x0001 pn: 0x00000000 dcid: 0xe5d0924ff82d5b9a scid:
  0x365428d847571dab token:[] length:1162
...
```

5.2 Challenges

The vast scope of QUIC inherently leads to high complexity. This is evident from the very beginning - the implementation overhead needed to just be able to build an answer to an initial packet is immense. Throughout the whole implementation process a carefully designed library is crucial to avoid major refactoring and rewrites later on.
Unfortunately, the newness of QUIC means there's a scarcity of widely available implementations, tutorials, and explanations. This lack of resources makes it more challenging to understand the protocol itself and derive a design for the QUIC library.
Most of the major setbacks encountered during the implementation process stemmed from the Rustls library. Despite its full support for TLS 1.3 and an API specifically designed for QUIC, a significant portion of its functionality is undocumented. This lack of documentation extends over most of the API, in that the explanations of function parameters is often incomplete or missing and examples with broader contexts are missing entirely. For instance, when initializing the RustlsConnection, it was unclear that the third parameter (`params`) referred to RFC section 18. Similarly, when calling `read_hs()`, which takes a reference to data, it wasn't clear where the pointer should start - at the beginning of the CRYPTO frame or the TLS message within it. Further testing revealed that the function returned an error, but not one of the standardized TLS 1.3 errors. To identify the issue, one has to explicitly poll for TLS alerts using the `alerts()` function which wasn't mentioned anywhere (Fig. 5.2).

lib.rs Rust

```rust
match self.tls_session.read_hs(crypto_frame.data()) {
    Ok(()) => println!("Successfully read handshake data"),
    Err(err) => {
        eprintln!("Error reading crypto data: {}", err);
        eprintln!("{:?}", self.tls_session.alert().unwrap());
    }
}
```

The biggest challenge, however, was related to the application layer protocol negotiation. This is an obligatory field, carried in any TLS client hello, and its contents are arbitrary but a standardized list of protocol codes is maintained by IANA, because when negotiating a protocol, for example http/3, the protocol code has to match on both endpoints. The initial test client, built with `aioquic` in Python, did not include this field, leading to an error and a TLS alert. After switching the test client to the example HTTP/3 client provided by the quinn library, the ALPN field was included, but the connection still failed. The Rustls documentation made no mention of its inability to handle the standardized IANA list. After digging deep within the library code, it became apparent that ALPN protocols have to be specified manually outside of the usual configuration. This involved directly setting the field within a specific struct of the RustlsConnection class.

5.3 Future Improvements

While the current QUIC library provides a solid foundation, it lacks several key features that need to be implemented first, in order to facilitate complete QUIC connections as per specification. The library is designed with the whole protocol in mind and should be easily expandable without requiring a major restructuring effort. That being said, there are still a number of areas within the existing codebase that could benefit from optimization and improvement, even if the pure QUIC implementation is feature complete at some point in the future.

5.3.1 Asychronous Design

The current design of the QUIC library operates within a single thread. However, a production grade implementation requires concurrency. Rust offers two primary models for achieving concurrency: OS threads and asynchronous runtime programming.

Asynchronous programming excels in scenarios dominated by I/O bound tasks, which is precisely the case for network applications like servers and databases. It significantly reduces CPU and memory overhead compared to traditional thread-based approaches. An async runtime manages a limited pool of expensive OS threads more efficiently. These threads are optimised to handle a much larger number of lightweight tasks, enabling significantly more concurrent operations compared to using raw OS threads directly.

Unfortunately, transitioning the current library to a concurrent model would necessitate a major rework, particularly for the `Connection` and `Endpoint` components. Facilitating concurrent reading and writing of serveral resources would require significant changes. For instance, the connection "database" design might need to be guarded by a mutex or even removed entirely in favor of a different approach (see Sec. 5.3.2). Additionally, the

`recv()` function for example would need to return a `Future` object, requiring the use of await statements to retrieve data in an asynchronous environment, along with multiple other functions.
The transition to an asychronous design would have major performance benefits but would require a major effort as Rusts asynchronous features are widely known to require signifiant effort to work with and are still maturing[4].

5.3.2 Connection Database Rework in Endpoint

The current design of the library keeps all connections within a vector inside the Endpoint struct. To manage these connections, a hashmap is used in which the current connection ID acts as the key and the corresponding index (`Handle`) in the vector serves as the value. This approach necessitates lookups for every received packet and every executed event, potentially becoming a performance bottleneck as the server aims to scale to handle hundreds of concurrent connections.
To address this concern and leverage the benefits of concurrent programming, a possible solution would be to shift the responsibility of managing individual connections to the user. This could be achieved by wrapping the `Connection` and `Endpoint` structs in new structures that expose a direct API of both underlying objects to the user.
For example, an `Endpoint` wrapper could be implemented as a `Server` struct. This `Server` would offer an asynchronous `accept()` method that would yield a `Future` containing the wrapped `Connection` object directly. Similarly, the Connection wrapper could expose functions like `accept_bi_stream()`, which again returns a wrapped `Stream` object directly.

main.rs (Concept) Rust

```rust
// copied from s2n-quic: https://github
   .com/aws/s2n-quic/blob/main/examples/rustls-mtls/src/bin/quic_echo_server.rs
while let Some(mut connection) = server.accept().await {
    // spawn a new task for the connection
    tokio::spawn(async move {

        while let Ok(Some(mut stream)) =
           connection.accept_bidirectional_stream().await {
            // spawn a new task for the stream
            tokio::spawn(async move {

                // echo any data back to the stream
                while let Ok(Some(data)) = stream.receive().await {
                    stream.send(data).await.expect("stream should be open");
                }
            });
        }
    });
}
```

This approach offers a two-fold advantage. Users would benefit from a more intuitive and user-friendly API design, while the library itself wouldn't need to deal with the

[4]https://bitbashing.io/async-rust.html

internal management of these objects, resulting in improved performance and scalability. Additionally this design would remove the need for individual events, as every action can now be performed on the object itself.

Chapter 6

Summary

QUIC, officially released in 2021 as a successor to TCP, aims to address longstanding issues affecting its predecessor. TCP over TLS suffers from problems such as Head-of-Line Blocking, lengthy handshakes, reliance on fixed IP addresses, and an inherent coupling between congestion control and data reliability.

QUIC features a robust architecture designed to overcome these shortcomings. It allows for multiple concurrent uni-directional and bi-directional streams within a single connection. Packets consist of headers and payloads, with payloads further divided into frames. Each frame carries specific data based on the frames type. Headers come in two forms: a longer version for connection establishment and a shorter version used during an established connection to minimize processing overhead. Notably, both the header and the complete payload are individually protected for enhanced security.

Flow control within QUIC operates at both the stream and connection level, with adjustments communicated via dedicated frames. The congestion control mechanism resembles TCP's approach, utilizing slow start and a cubic algorithm. Additionally, QUIC introduces other features such as a 0-RTT (Zero Round Trip Time) handshake which allows for immediate application data transmission upon re-establishing a recently turned inactive connection. Additionally, connection migration enables connections to seamlessly switch between network interfaces without disruption. During a typical 1-RTT (One Round Trip Time) QUIC handshake, TLS integration enables application data transfer after a single round trip.

Connections can be terminated due to timeouts, immediate closure, or stateless resets. QUIC employs acknowledgment (ACK) frames for loss detection, with a single ACK frame capable of acknowledging multiple QUIC packets. Frames within a lost packet can be retransmitted across multiple succeeding packets for improved reliability. The security of QUIC has been rigorously tested against various attack vectors, including replay attacks, packet manipulation, and fuzzing. Interestingly, fuzzing proved to be the most effective method, causing a server crash by manipulating offsets within stream frames.

A key element of this thesis is a dedicated QUIC implementation in Rust. Even though the implementation can not yet be considered feature complete, it contains the most important core functionality. It features an event based design which proved highly useful in conveying important information between the application and the connection.

The `Endpoint` object, handles a UDP socket and manages connections. Incoming packets are either matched to an existing connection or trigger the creation of a new one. Each `Connection` object encapsulates the TLS session provided by Rustls, manages received

packets across all packet number spaces, handles keying material, and populates outgoing packets with data. Encryption and decryption of packet headers and payloads are handled by functions provided by Rustls.
The software design is highly influenced by Rusts characteristics and feature set, specifically the borrow checker, which prevented memory related bugs but altered the design process. While some parts of the language are highly complex, such as the lifetime annotations when working with references, the compiler always provided clear and useful error messages resulting in only a small amount of the overall time used for bug fixing. Rust promotes to minimise usage of mutable variables and references and encourages to use the stack in favor of the heap as much as possible.
Time constraints resulted in remaining potential for further improvements. These include the adoption of an asynchronous design and shifting the responsibility of connection management to the user to reduce workload in the endpoint.

Bibliography

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

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

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

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

[5] D. Lee, B. E. Carpenter, and N. Brownlee, "Observations of udp to tcp ratio and port numbers", in *2010 Fifth International Conference on Internet Monitoring and Protection*, 2010, pp. 99–104. DOI: 10.1109/ICIMP.2010.20.

[6] D. Borman, *TCP Options and Maximum Segment Size (MSS)*, RFC 6691, Jul. 2012. DOI: 10.17487/RFC6691. [Online]. Available: https://www.rfc-editor.org/info/rfc6691.

[7] *Quic working group history*, https://datatracker.ietf.org/wg/quic/history/, Accessed: 2024-02-14.

[8] M. Bishop, *HTTP/3*, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.

[9] M. Duke, *QUIC Version 2*, RFC 9369, May 2023. DOI: 10.17487/RFC9369. [Online]. Available: https://www.rfc-editor.org/info/rfc9369.

[10] A. Gurtov, T. Henderson, S. Floyd, and Y. Nishida, *The NewReno Modification to TCP's Fast Recovery Algorithm*, RFC 6582, Apr. 2012. DOI: 10.17487/RFC6582. [Online]. Available: https://www.rfc-editor.org/info/rfc6582.

[11] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger, *CUBIC for Fast Long-Distance Networks*, RFC 8312, Feb. 2018. DOI: 10.17487/RFC8312. [Online]. Available: https://www.rfc-editor.org/info/rfc8312.

[12] P. Balasubramanian, Y. Huang, and M. Olson, *HyStart++: Modified Slow Start for TCP*, RFC 9406, May 2023. DOI: 10.17487/RFC9406. [Online]. Available: https://www.rfc-editor.org/info/rfc9406.

[13] *Cubic and hystart++ support in quiche*, `https://blog.cloudflare.com/cubic-and-hystart-support-in-quiche`, Accessed: 2024-03-11.

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

[15] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is quic? provable security and performance analyses", in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 214–231. DOI: `10.1109/SP.2015.21`.

[16] A. Saverimoutou, B. Mathieu, and S. Vaton, "Which secure transport protocol for a reliable http/2-based web service: Tls or quic?", in *2017 IEEE Symposium on Computers and Communications (ISCC)*, 2017, pp. 879–884. DOI: `10.1109/ISCC.2017.8024637`.

[17] G. S. Reen and C. Rossow, "Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic", in *Proceedings of the 36th Annual Computer Security Applications Conference*, ser. ACSAC '20, , Austin, USA, Association for Computing Machinery, 2020, pp. 332–344, ISBN: 9781450388580. DOI: `10.1145/3427228.3427662`. [Online]. Available: `https://doi.org/10.1145/3427228.3427662`.

[18] *Rustls library*, `https://github.com/rustls/rustls/`, Accessed: 2024-02-27.

[19] *Octets library*, `https://github.com/cloudflare/quiche/tree/master/octets/`, Accessed: 2024-02-27.

[20] *Rcgen library*, `https://github.com/rustls/rcgen/`, Accessed: 2024-02-27.

[21] *Ring library*, `https://github.com/briansmith/ring/`, Accessed: 2024-02-27.

List of Figures

List of Tables

Statutory Declaration

I hereby state that I have written this Bachelor's Thesis independently and that I have not used any sources or aids other than those declared. All passages taken from the literature have been marked as such. This thesis has not yet been submitted to any examination authority in the same or a similar form.

Düsseldorf, 18. März 2024

Christoph Matthias Britsch