NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

TIAGO JORGE DA SILVA DUARTE

BSc in Computer Science

QUICSAND: A DEEP STUDY ON THE PERFORMANCE OF DIFFERENT QUIC IMPLEMENTATIONS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

Draft: *February 14, 2024*

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

QUICSAND: A DEEP STUDY ON THE PERFORMANCE OF DIFFERENT QUIC IMPLEMENTATIONS

TIAGO JORGE DA SILVA DUARTE

BSc in Computer Science

Adviser: Professor João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
Draft: *February 14, 2024*

Abstract

QUIC is a transport protocol developed by Google, that is rapidly becoming deployed on server infrastructure, making it the transport protocol used by a relevant share of Internet traffic. Designed to offer security, multiplexing, low-latency communication, and connection migration, among multiple other features, QUIC stands out for its versatility and rich feature set. Differently than most transport protocols, QUIC operates at the application layer, the key motivation being to simplify the evaluation of the protocol without the cost of deploying a new version across a large number of operating systems kernels. This encourages the emergence of diverse implementations compared to the classical transport protocols.

Nowadays, it is becoming increasingly challenging for applications to identify the optimal combination that best suits their specific requirements. This complexity underscores the need for new mechanisms that ease the selection process ensuring that applications can efficiently leverage the most effective implementation for their intended purposes.

To this end we propose to build QuicSand, a novel dedicated tool to evaluate and compare various QUIC protocol implementations. By simplifying the testing process, we aim to help users be able to quickly and systematically evaluate performance, allowing informed decision-making in selecting the most suitable QUIC implementation for their needs. This work also aims to produce a comprehensive report with results and experiments that can be performed with QuicSand to evaluate the performance of the different QUIC implementations.

Keywords: QuicSand, QUIC protocol, performance evaluation, tool, QUIC implementations, comparison, analysis

Resumo

QUIC é um protocolo de transporte desenvolvido pela Google que está a ser rápidamente implantado na infrastrutura do servidor, tornando-se o protocolo de transporte utilizado por uma parcela relevante do tráfego da Internet. Projetado para oferecer segurança, multiplexagem, comunicação de baixa latência, e migração de conexões, entre muitas outras características, o QUIC destaca-se pela sua versatilidade e conjunto de funcionalidades. Diferentemente da maioria dos protocolos de transporte, QUIC opera na camada da aplicação, tendo como principal motivação simplificar a avaliação do protocolo sem o custo de implantar uma nova versão num grande número de kernels de sistemas operativos. Isto incentiva ao surgimento de diversas implementações em comparação com os protocolos de transporte clássicos.

Atualmente, está-se a tornar cada vez mais desafiador para as aplicações identificar a combinação ideal que melhor atenda aos seus requisitos específicos. Esta complexidade sublinha a necessidade de haver novos mecanismos que facilitem o processo de seleção, garantindo que as aplicações possam aproveitar eficientemente a implementação mais eficaz para os seus propósitos pretendidos.

Para este fim, propomos desenvolver o QuicSand, uma ferramenta inovadora e dedicada para avaliar e comparar várias implementações do protocolo QUIC. Ao simplificar o processo de teste, pretendemos ajudar os utilizadores a serem capazes de avaliar rapidamente e de forma sistemática o desempenho, permitindo a tomada de decisões informadas na seleção da implementação QUIC mais adequada às suas necessidades. Este trabalho também tem como objetivo produzir um relatório abrangente com resultados e ensaios que podem ser realizados com o QuicSand para avaliar o desempenho das diferentes implementações do QUIC.

Palavras-chave: QUIC, implementações QUIC, avaliação de performance, testes, análise

Contents

List of Figures

List of Tables

1 Introduction

The Internet, and computer networks in general, are essential in modern societies, having a direct impact on the everyday life of citizens across many domains, from interaction with government services, health, work, and even cultural purposes. However, the Internet stack has remained without significant changes for decades with few exceptions, such as IPv6 at the network layer and the evolution of the TLS protocol, among others.

In the last decade, Google has developed a novel Transport protocol, named QUIC, whose main goal is to improve the support offered for web applications and that, contrary to classical transport protocols such as TCP and UDP, operates at the application level (on top of UDP) combining features of TCP with security mechanisms usually implemented on upper layers (i.e., TLS). The protocol operates at the application level to foster faster evolution and innovation. This has resulted in the proliferation of different implementations and variants of QUIC including different alternatives to implement mechanisms such as congestion control, and flow control among others.

Hence, there is a critical necessity for tools that simplify the validation and evaluation of performance across various implementations. Such tools would simplify the process and enable a more effective comparison, aiding in the selection of the most suitable implementation for supporting a particular application in a particular setting.

Motivation

The main issue of the current network stack is the difficulty of innovating and deploying in the lower layers, in this work we focus on the transport layer. The increase in short-lived connections, a slow TCP/TLS handshake process and the search for the most effective flow and congestion control algorithms have motivated the development of QUIC, a new transport protocol that operates at the application level (on top of UDP) which turns to help an ease deploy on the network stack.

While a few works [15, 19, 25, 28, 31] have already compared the different performance gains of different implementations and variants in some operational conditions, an additional effort has to be conducted to enable more complete and comprehensive

comparisons between new implementations and variants of QUIC that might appear, particularly considering different network conditions.

In this work, we plan to push the state of the art by developing a methodology and tools to simplify the performance evaluation of different implementations and variants of QUIC across different operational conditions. To achieve this we plan to exploit both network emulators, which can create realistic network conditions to test such implementations, and different workloads including the ones using QUIC and packet flows using other transport protocols to condition the environment. We aim to develop a methodology and tools that will allow users to gain a deeper understanding of the performance. Evaluation-wise, we also plan to take advantage of the network emulator to emulate network anomalies that can affect the operation of the protocol. Finally, based on these results we plan to propose and if time allows implement and test modifications to the operation of QUIC.

Expected Contributions

We aim to produce the following contributions:

1. A tool to simplify and automatize the evaluation process of different QUIC implementations and variants in different network conditions;
2. A set of experiments and tests to evaluate the performance of different QUIC implementations.
3. A analysis of results and a comparison of evaluated performance for different existing QUIC implementations.

Document Structure

The rest of this document is organized as follows:

- Chapter 2 presents the concepts and technologies that are relevant to this work starting with a brief overview of the Internet stack model, then it covers the most relevant protocols of each layer to better contextualize this work, and finally the chapter ends with a group of previously developed tools and performance studies of QUIC, and performance metrics that might help to evaluate different implementations, finishing with a quick summary of all the covered topics;
- Chapter 3 is the chapter that introduces some of our initial ideas to develop our evaluation methodology, a tool architecture preliminary design and measurement plans that will be developed to achieve the goals of this work;

2

Background and Related Work

This work is focused on the development of a tool for simplifying the execution of comprehensive performance evaluations to address the faster deployment of QUIC implementations. This chapter introduces the concepts and technologies that are relevant to this work starting with a brief overview of the Internet stack model, then it covers the most relevant protocols. The chapter proceeds with a group of previously developed tools, performance studies of QUIC, and performance metrics that might help to evaluate different implementations, concluding with a presentation of network emulation tools.

2.1 Network Protocols

For this work, it is important to understand the network protocols that are used on the Internet. To further understand the QUIC protocol and the main focus of this work, it's necessary to understand how the network stack is defined. The Open System Interconnection (OSI) model [21] is a reference to how messages navigate from two points in the network. The model refers to a combination of 7 layers, each having a different role, as can be seen in Figure 2.1. In this section, we will present the most relevant protocols, their purpose, and their main features.

The OSI model is characterized by the following layers presented from the lowest to the highest layer, following some concepts of the textbook by Kurose and Ross (2016) [22]:

Physical Layer - Layer 1 The physical layer is responsible for the transmission and reception of the unstructured raw data between a device and a physical transmission medium. The physical layer is the lowest layer of the OSI model.

Data-Link Layer - Layer 2 The data-link layer is responsible for the node-to-node delivery of the network layer datagrams. This layer is responsible, for example, for providing reliable delivery, depending on the protocol that manages the data delivery. Some examples of link-layer protocols are Ethernet, Wi-Fi, DOCSIS and PPP [6].

Network Layer - Layer 3 The network layer is responsible for addressing the source and the destination of segmented data received from the transport layer, encapsulating

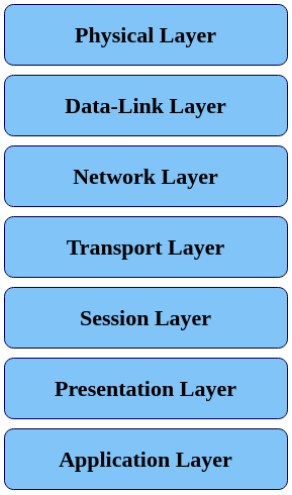

Figure 2.1: OSI Model

the data into packets. The layer also counts on a protocol named ARP [1] for the address resolution to know the media access control (MAC) [27] of the source and destination, transferring the data to the data-link layer. The network layer includes the Internet Protocol (IP) [17] and routing protocols that determine the routes in a best-effort way to the destination endpoint.

Transport Layer - Layer 4 The transport layer is responsible for the delivery of the messages from one process to another. This layer can provide segmentation and reassembly of the data, flow control, error control, and connection control. The most relevant protocols are TCP [36] and UDP [37] both offering different features, explored in depth in further sections.

Session Layer - Layer 5 The session layer is in charge of the establishment, management and termination of the connections between applications. This layer is also responsible for the synchronization of the dialogue between the two applications.

Presentation Layer - Layer 6 The presentation layer manages the translation, compression and encryption of the data. This layer is also responsible for the data syntax so that the application layer can understand the data, carrying well-known protocols such as SSL [13] and TLS [8].

Application Layer - Layer 7 The application layer resides at the top of the OSI model and includes protocols such as HTTP [26], FTP [12], and DNS [9, 10]. This layer is responsible for the interaction between the application and the network.

2.1.1 Transport Layer Protocols

In this section, we will present the most relevant transport layer protocols but from two different perspectives: the Transport Control Protocol (TCP) in a more detailed fashion to understand some decisions in QUIC Protocol design due to principles that both protocols

incorporate, such as reliability, flow and congestion control, among others. In addition, we will introduce the User Datagram Protocol (UDP) as the protocol that QUIC is built on top of.

2.1.1.1 Transport Control Protocol (TCP)

According to the textbook by Kurose and Ross (2016) [22], the transport control protocol (TCP) is a protocol that integrates the transport layer of the OSI model. It is a connection-oriented and reliable protocol, that relies on many principles such as error detection, cumulative acknowledgements, retransmission of lost packets, flow and congestion control, and timers, among others, which will be explored in this section.

TCP Connection. This network transport protocol is known as connection-oriented due to its requirements for a "handshake" between processes before any data transmission occurs, which implies exchanging connection parameters to initialize TCP state variables. Is important to understand that TCP only runs in the end systems, which means intermediate network elements do not have any knowledge of the connection state.

The connection affords a full-duplex service, allowing two processes to transmit data simultaneously. Furthermore, it adheres to a point-to-point model, signifying the presence of a single sender and a single receiver.

Considering the following example, in a TCP connection two processes are running, let's call the client and server process, where the client application tells the transport layer of its intention to establish a connection with another process. This process called a three-way handshake, starts with the client sending an SYN (synchronize) segment to the server, consequently, the server sends an SYN-ACK (synchronize-acknowledgement) segment to the client, and finally, the client sends an ACK (acknowledgement) segment to the server that may or may not contain data payload. After this process the connection is established and the client and server can exchange data (in both directions).

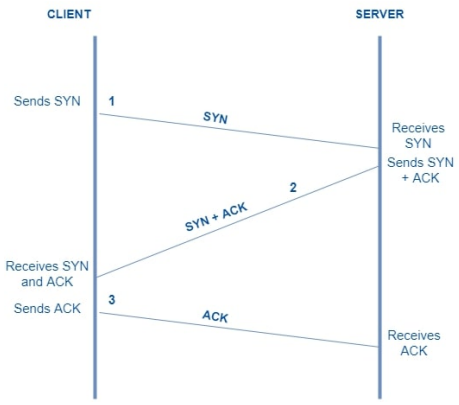

Figure 2.2: TCP Connection Establishment (Taken from [32])

Once the connection is established, the TCP process within the client retrieves periodically chunks of data from the send buffer transferring them to the network layer. The TCP

process on the server side acquires the data from the network layer and passes them to the receive buffer. The formation of TCP segments involves pairing the data payload with the TCP header.

TCP Segment Structure. The TCP segment structure, as shown in Figure 2.3 involves a TCP header of 20 bytes long and a data payload that can vary from 0 to 65,535 bytes, limited by a variable called maximum segment size (MSS), which is set during the handshake. The TCP header contains the following fields:

- The 16-bit each source and destination port field, used to multiplex or demultiplex the data from or to the applications.
- The 32-bit sequence number field and acknowledgement number field, are used to provide reliable data transfer.
- The 16-bit window size field, is used for flow control.
- The 4-bit header length field, is used to indicate the length of the TCP header, due to the possible options field.
- The options field (variable length), used for instance for MSS negotiation, window scaling factor or timestamping.
- Flag fields (6 bits), used to indicate the purpose of the segment, such as SYN, ACK, and FIN, among others.

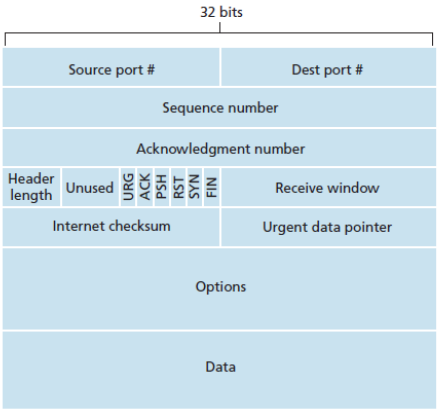

Figure 2.3: TCP Segment Structure (Taken from [22])

The TCP sequence number for a segment depends on the MSS, incremented by the byte count within the segment's data field. The acknowledgement number is the sequence number of the next byte the sender of the segment is expecting to receive. The TCP RFC [36] does not imply anything about receiving segments out-of-order, which gives the freedom to the TCP implementation to handle this situation. There are two alternatives:

1. The TCP implementation can buffer the out-of-order segments until all the data is received, and then deliver the data to the application in order;

2. The TCP implementation immediately discards out-of-order segments.

During the TCP connection, both sides choose initial sequence numbers randomly. This is done to avoid the possibility of old duplicate segments being mistaken for new segments.

Timeout and Estimation of the Round-Trip Time. To recover from packet loss, TCP uses a timeout and retransmission mechanism. However, a question arises on which is the optimal value timeout. Perhaps the value needs to be higher than the round-trip time (RTT). To address this issue TCP needs to first estimate the RTT.

The sample RTT, represented by *SampleRTT* is the time it takes for a segment to go from the sender to the receiver and the acknowledgement to go from the receiver to the sender. This value is recalculated for each segment that is sent, for the first time, which means that retransmitted segments are not considered. Then TCP maintains a value called *EstimatedRTT*, which is obtained by the following equation:

$$EstimatedRTT = (1 - \alpha).EstimatedRTT + \alpha.SampleRTT \tag{2.1}$$

Where α is a constant value between 0 and 1, which is used to weight the current sample. The value of α is recommended to be set to 0.125.

Beyond having an estimate of the RTT, TCP also needs to measure the variability of the RTT, denoted as *DevRTT*, which is calculated by the following equation:

$$DevRTT = (1 - \beta).DevRTT + \beta.|SampleRTT - EstimatedRTT| \tag{2.2}$$

Where β is a constant value between 0 and 1, which is used to weight the current sample and recommended to be set as 0.25.

The timeout interval is now set to be calculated requiring to be greater or equal to the *EstimatedRTT*, however, it can not be too much higher. The TCP RFC recommends setting the timeout interval equal to the *EstimatedRTT* plus four times the *DevRTT*.

$$TimeoutInterval = EstimatedRTT + 4.DevRTT \tag{2.3}$$

The TCP RFC advises initiating the *TimeoutInterval* at 1 second although, in case of a premature timeout, the value should be doubled. Upon receiving a segment acknowledgement a new value calculated by the equation above should be promptly set.

TCP Reliable Connection. TCP provides a reliable data transfer on top of an unreliable network layer. Where the network layer can lose, duplicate or reorder packets. TCP provides a reliable data transfer triggering three events:

1. Upon receiving application data TCP encapsulates the data into segments and sends them to the network layer. If the timeout timer is not running yet, it is started when the first segment is passed to the IP layer.

2. Upon timeout expiration, the segment is retransmitted and the timeout timer is restarted.

3. Upon receiving an ACK, TCP compares the value received with the next sequence number expected. If the value is equal to or greater than the next sequence number expected, the value is updated and the timer is restarted if there are any not-yet-acknowledged segments. If the value is less than the next sequence number expected, the ACK is ignored.

The TCP also doubles the timeout interval after each timeout event, to avoid premature timeouts. The value *TimeoutInterval* is derived from the latest values. TCP also uses a mechanism called fast retransmit, which is triggered when the sender receives three duplicate ACKs indicating that the segment with the sequence number after the ACK received is lost. This mechanism is used to prevent the timeout interval from expiring, which might be a slow process.

Flow Control. In a TCP connection the protocol passes the data from the IP layer to the receiver buffer. This data might or might not be immediately read by the application. So to cover possible congestion in the network or the receiver side, TCP provides mechanisms to control the flow and congestion of the data. It is important to understand that these two mechanisms are different, the flow control matches the sender's rate to the receiver's rate, while the congestion control is a possible limitation on the sender to avoid possible network congestion problems.

As mentioned before, the TCP header has a field called receive window. This field is meant to inform the sender of the amount of free space in the receiver buffer. The receiver sends the value of the receive window in the ACK segment and limits the amount of data sent to the receiver using the value of the receive window field. The receiver can dynamically change the value of the receive window, to inform the sender that the buffer is full or not. The equation to calculate the receive window is:

$$rwnd = RcvBuffer - (LastByteRcvd - LastByteRead) \tag{2.4}$$

In Equation 2.4, *RcvBuffer* is the size of the receiver buffer, *LastByteRcvd* is the last byte of data received correctly from the network and placed in the receiver buffer, and *LastByteRead* is the last byte of data read from the receiver buffer. For the validation of the equation, it is necessary to know that on the receiver side, the buffer is limited by the value of the receive window, so the receiver can only receive data if the following inequation is true:

$$LastByteRcvd - LastByteRead \leq RcvBuffer \tag{2.5}$$

On the sender side, the buffer is limited by the value of the receive window, so the sender can only send data if the following inequation is true:

$$LastByteSent - LastByteAcked \leq rwnd \tag{2.6}$$

TCP Connection Management. The TCP connection management refers to two topics: (i) connection establishment, which is known as the three-way handshake and (ii) connection termination mentioned as the four-way handshake [33]. QUIC, as the main focus of the study, has a different approach to connection establishment and termination, so it is important to go more deeply into this topic, also due to most of the attacks performed are in these two phases.

In terms of connection establishment, the TCP protocol has the famous three-way handshake mentioned earlier. In the first step, the SYN segment is sent from the client without any payload, the SYN bit is set to one, and the client sends a random initial sequence number (*client_isn*) to avoid possible attacks, like replay attacks. In the second step, the server receives the SYN segment and sends a SYN-ACK segment to the client, the SYN bit is set to one, the ACK bit is set to one, the acknowledgement number is set to *client_isn + 1*, and the server sends a random initial sequence number (*server_isn*) for the same reason. In the third step, the client receives the SYN-ACK segment and sends an ACK segment to the server, the SYN bit is set to zero, the ACK bit is set to one, and the acknowledgement number is set to *server_isn + 1*. After these three steps, the connection is established and the client and server can exchange data.

On the other hand there is connection termination, composed by the four-way handshake. In the first step, the client sends a FIN segment to the server, the FIN bit is set to one, and the server responds with the ACK to that segment, the same process is repeated on the server side. After this process, the connection is terminated and all resources on both sides are deallocated.

TCP Congestion Control. Another mechanism in TCP is the congestion control. This mechanism is used to avoid possible congestion issues in the network, so the protocol is responsible for treating the issues in the endpoints.

Although TCP already has a window to control the flow of the data, the congestion control also uses a window *cwnd* (congestion window) to control the rate of the data sent, based on the congestion of the network. The congestion window imposes a constraint on the sender, which is especially based on the number of bytes that are unacknowledged and the sender not exceeding the minimum bytes available between *cwnd* and *rwnd*.

The congestion window and the receiver window control the rate of the data transfer we now explain how the congestion window is calculated and managed in conformity

with the congestion of the network. The congestion can be detected with a timeout or the sender receiving three duplicated acknowledgements. The TCP needs to readjust the value of the *cwnd* based on the rate of the acknowledgements received.

The TCP protocol has three different congestion control algorithm components: the slow start, the congestion avoidance, and the fast recovery. The slow start is the first phase of the algorithm. Is typically used when the connection begins setting the *cwnd* to a small value (1 MSS). This value is increased by doubling the value of the *cwnd* for each ACK received. The slow start ends when the value of the *cwnd* reaches the threshold (*ssthresh*) or whenever some datagram is dropped. Upon a dropped datagram, TCP sets the value of the *cwnd* to 1 again and slowly starts again until it reaches the *ssthresh* set before. Congestion avoidance is the second phase of the algorithm, typically used after the slow start phase, setting the *cwnd* to the *ssthresh* value increased by one MSS for each RTT. The fast recovery is the third phase of the algorithm used when the sender receives three duplicated ACKs. The value of the *cwnd* is set to half of the current value of the *cwnd* plus three MSS. The congestion avoidance is then used to increase the value of the *cwnd*.

Fairness Among TCP Connections. TCP due to its congestion control mechanism can distribute the bandwidth fairly among the connections, enabling equal bandwidth values for each connection independently of the size of the objects transferred in the connection. The problem is that a client that wants to transfer data from a server can open multiple connections to the server and try to get more bandwidth compared to other clients, that only use one connection to the server.

2.1.1.2 User Datagram Protocol (UDP).

UDP protocol is a connectionless and unreliable transport protocol where QUIC is built on top of it. The main motivation for the design of this transport protocol is to provide low-overhead data transport service for applications that do not require the reliability of TCP.

UDP Segment Structure. The UDP segment structure as it is shown in Figure 2.4, has only four fields composing the header of the datagram 16 bits each. The source port and destination port fields as mentioned before are used to multiplex and demultiplex the data from or to the applications, proceeding it become the length field that is used to indicate the length of the UDP header and data, and finally, the checksum field is used to detect errors in the UDP segment.

UDP Checksum. The checksum is calculated by the sender and verified by the receiver. The checksum is the sum of the 16-bit words in the segment, with the checksum field considered to be 0. If the result is all 1s, the checksum is considered to be 0. The checksum is used to detect errors in the UDP segment, but it does not provide any error recovery mechanism.

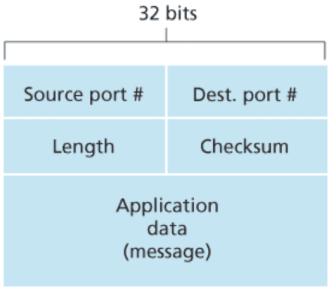

Figure 2.4: UDP Segment Structure (Taken from [22])

2.1.2 Session Layer Protocols

In this section, we discuss a particularly relevant session layer protocol: the Transport Layer Security (TLS) protocol [8]. This protocol is used to provide security to the connection between two endpoints. There are three versions of the protocol already. In the implementation of QUIC, TLS 1.3 is the one that is used so the focus will be primarily on this version.

Exploring the TLS protocol we first need to start with the SSL [13], the Secure Socket Layer. The SSL is composed of two different layers: the higher layer contains the SSL Handshake Protocol, the SSL Change Cypher Spec Protocol and the SSL Alert Protocol, and the lower layer holds the SSL Record Protocol. The SSL Handshake is the protocol that is responsible for establishing a secure connection between two endpoints, exchanging cipher suites and keys, and proving peer and message authentication. The SSL Change Cipher Spec Protocol is responsible for changing the cipher suite and keys used to encrypt the data. The SSL Alert Protocol is responsible for sending alerts to the peer. Finally, the SSL Record Protocol provides the fragmentation, compression and encryption of the data.

Various attacks were conducted to this protocol, and a study [11] was made to analyze the attacks and the countermeasures that were made to the protocol and we will present some of the attacks:

Cipher Suite Rollback - This attack is based on the capacity of the attacker to intercept the cipher suit message and change the cipher to a weaker one.

Heartbleed - This attack consisted of a buffer overflow where the client sends a heartbeat message to the server, and the server responds with the same message, however, the server does not check the size of the message, so the client can send a message with a bigger size than the server can handle, leaking possible sensitive information from the server.

POODLE - This attack consisted of the attacker forcing the client and server to use the SSL 3.0 protocol, which is vulnerable to a padding oracle attack.

Key exchange algorithm confusion - In this attack, the attacker forces a server to use a RSA key exchange algorithm, while exchanging a Diffie-Hellman key exchange algorithm with the client.

There are other attacks and due to these vulnerabilities, the community decided to create a new protocol to reinforce the security challenges that the SSL protocol was facing, creating a new protocol called TLS.

2.1.2.1 Transport Layer Security (TLS)

TLS 1.3 is the version of the protocol that is currently used to provide security to the connection between two endpoints. There are some differences between TLS 1.2 and 1.3, but they have similar security goals. The main difference between the two versions is the fact that the handshake in TLS 1.3 is one round-trip time before the client can send the application data and the new 0-RTT mechanism allows clients to send data in the first RTT. The TLS [7] handshake is divided into several steps, as can be seen in Figure 2.5.

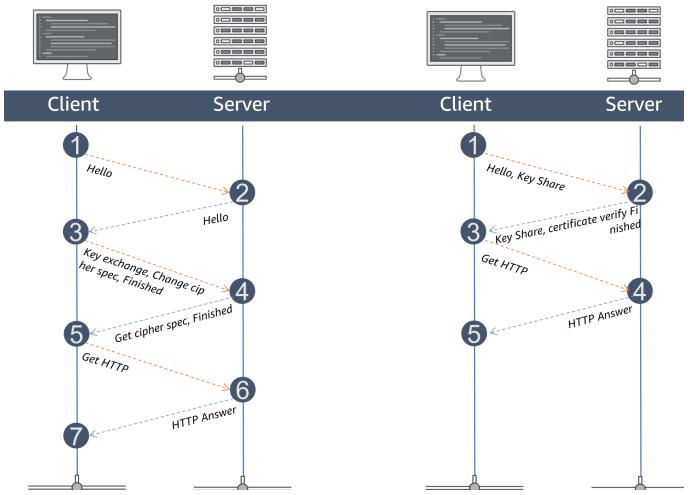

Figure 2.5: TLS 1.2 2-RTT and TLS 1.3 1-RTT Handshakes (Taken from [39])

Client Hello: TLS initiates the handshake with the client hello message, which contains: the highest TLS version supported by the client, a random number generated, and the list of supported cipher suites.

Server Hello: The server responds to the client with a server hello containing the TLS version chosen, the selected cipher suite, a random number generated by the server and the server's digital certificate.

Authentication and Key Exchange: The client verifies the server's digital certificate, consulting the certificate chain and getting to a certificate authority (CA). It can also check the revocation list of the CA to see if the certificate is still valid. The key exchange and authentication are done using symmetric and asymmetric encryption.

Symmetric encryption is typically used to encrypt the data, and the most used protocol is the Diffie-Hellman for key exchange. The client and the server exchange a public number and combine it with their private number to generate a shared secret. Asymmetric encryption is mostly used for server authentication. The server sends a digital signature of the shared secret that can be verified by the client using the server's public key.

Change Cipher Spec: This is a simple message that indicates that the client and server will start to use the negotiated cipher suite and keys to encrypt the data.

TLS Components and Security Features. TLS is composed of several components and security features, such as Cipher Suites which are cryptographic algorithms used for key exchange, authentication and encryption; the TLS Record Protocol, responsible for the fragmentation and encryption of the data; the Public Key Infrastructure (PKI) and CA's providing secure communication with digital certificates and the certificate authorities, providing peer authentication; Perfect Forward Secrecy (PFS) that ensures security even if the private key is compromised, achieved with ephemeral key exchange.

2.1.3 Application Layer Protocols

The application layer is the top layer of the OSI model stack, where reside protocols, such as HTTP which provides transfers of documents on the web, FTP [12] which supplies the transfer of files between two hosts, SMTP [20] for the transfer of e-mail messages, and SPDY, among others. This subsection has presented two protocols, the HTTP protocol [26] that are one of the most used protocols on the Internet and uses the QUIC protocol in its latest version, and the SPDY protocol [2] which was one of the motivations for the development of the QUIC protocol.

2.1.3.1 Hyper Text Transfer Protocol (HTTP)

HTTP protocol is a data transfer protocol, initially designed for HTML (Hyper Text Markup Language) [3] documents, but now it is used for a wide range of data formats, such as images, videos, or programming code to be interpreted by the web browser.

The HTML language is a notation for hypertext documents, that uses tags to define the structure and how the document must be formatted and visualized.

Now that the HTML language was introduced we can finally go deep in the HTTP. The protocol in its base version is built on top of TCP connections and has just two different messages: HTTP Request and HTTP Reply. Each message can only have a single object, for instance, if a client wants to send multiple n objects, it needs to exchange n messages with the server. In HTTP it is possible to make different types of requests: the most known and referred ones are the GET, POST and PUT. The first one is used to ask for an object, the second one is used to send forms, and the last one is used to upload objects to the server (Legatheaux *et al.*, 2018) [23].

The messages exchanged in the HTTP protocol have a specific structure, and in comparison with other protocols already mentioned like TCP and UDP, the HTTP messages are human readable, which means that are not ordered sequences of bytes like the mentioned ones. These messages have structured headers and a body, where a body part is optional. The lines are written in the US-ASCII character set, and the end of the line is represented by the sequence of two characters: carriage return and linefeed, also known as CRLF. The headers are separated from the body by a blank line.

The protocol is widely used on the Internet and there was a need to improve different performance mechanisms, for example, the extensive usage of caching on the client side, as well as the reutilization of TCP connections. These improvements were made in the HTTP/1.1 version. The HTTP/2 version was released in 2015, and the main goal was to improve the performance of the protocol, by reducing the latency and improving the network and server resources usage, by introducing a new multiplexing mechanism, where multiple requests can be sent in parallel over a single TCP connection. The HTTP/3 version is the one that is built on top of QUIC connections to outperform the earlier versions of the protocol.

2.1.3.2 SPDY

SPDY is a protocol developed to address the issues of: opening multiple concurrent HTTP connections, the impossibility of pushing content to the client if the server already knows what resource a client will request, and the fact that an HTTP client can only fetch one resource at a time over a single TCP connection. SPDY introduced the possibility of multiplexing multiple requests over a single TCP connection, compressing HTTP headers, prioritizing requests in parallel, and a pushing mechanism whenever a server knows which resource a client will request.

SPDY [5] showed improvements compared to the HTTP/1.1 protocol in wired connections. The problem still emerges when the connection is wireless, where the latency is higher and the packet loss is more frequent. The packet loss is a big deal and brings a disadvantage to SPDY, because the protocol is built on top of TCP connections, and the TCP protocol is not able to recover from packet loss in a fast way. If a packet is lost in SPDY will block the entire TCP connection, and the single loss will affect all streams because they share the same congestion window. The same problem occurs when there are out-of-order packets, which is a common problem in wireless connections. The SPDY also relies on the TCP handshake, which requires one RTT plus three RTTs in case of an SSL/TLS connection that will be costly in high latency connections [5].

Therefore, the SPDY protocol was one of the motivations to still improve the performance of the HTTP protocol, especially in wireless connections.

2.2 QUIC: Quick UDP Internet Connections

QUIC (Quick UDP Internet Connections) is a transport protocol developed by Google that aims to provide a faster and more secure alternative to TCP (Transmission Control Protocol). It is designed to operate at the application layer, which means that it does not require kernel support and can be implemented entirely in user space. This allows for faster development and deployment of new transport-layer optimizations. To further understand the comparison between the different variants of QUIC, it is important to understand the main features of the protocol. In this section, we will start to present the invariants properties of QUIC presented in the RFC8999 [34]. It will be complemented with the first version of the QUIC version presented in RFC9000 [18].

2.2.1 QUIC Version-Independent Properties

QUIC as defined in the RFC8999 [34] "is a connection-oriented protocol between two endpoints. Those endpoints exchange UDP datagrams. These UDP datagrams contain QUIC packets. QUIC endpoints use QUIC packets to establish a QUIC connection, which is a shared protocol state between those endpoints". To understand better the QUIC protocol, we will present the invariant properties of the protocol.

QUIC Packets. Endpoints that use QUIC, exchange UDP datagrams, and these UDP segments might or might not have multiple QUIC packets. We will only consider the first packet to explain how the packets are structured. There are two types of packets: long header packets and short header packets, illustrated in Figure 2.6. The packet type is determined by the most significant bit of the first byte of the packet, if the bit is set to one, then the packet is long header, if not the packet is short header. It is important to understand that QUIC packets in version negotiation are not integrity protected and the packet payload is version dependent with an arbitrary length.

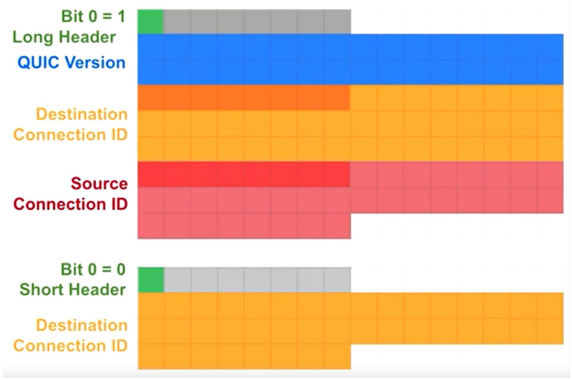

Figure 2.6: Long and Short Packets Header (Taken from [29])

Long Header Packets. The long header packets are composed of the following fields:

- **Packet Type** (1 bit): The most significant bit of the first byte of the packet is set to one.
- **Version-Specific Bits** (7 bits): These bits are version-dependent.
- **Version** (4 bytes): The version of the QUIC protocol.
- **Destination Connection ID Length** (1 byte): The length of the destination connection ID.
- **Destination Connection ID** (0-255 bytes): The destination connection ID.
- **Source Connection ID Length** (1 byte): The length of the source connection ID.
- **Source Connection ID** (0-255 bytes): The source connection ID.
- **Version-Specific Data**: This field is version-specific content with no specific length specification.

Short Header Packets. The short header packets are composed of the following fields:

- **Packet Type** (1 bit): The most significant bit of the first byte of the packet is set to zero.
- **Version-Specific Bits** (7 bit): These bits are version dependent.
- **Destination Connection ID**: The destination connection ID with no particular length specification.
- **Version-Specific Data**: This field is version-specific content with no specific length specification.

Connection ID and Version. The connection ID is used as a way to identify the connection between two endpoints, disallowing QUIC packets to be delivered to the wrong QUIC endpoint. This field is also chosen by the endpoint of each endpoint. The version field is used to identify the version of the protocol being used. The important property is that the 0x00000000 version is reserved for version negotiation packets.

Version Negotiation Packet. The version negotiation packet is a longer header packet that is used to negotiate the version of the protocol. This packet contains some particular fields, such as the version field, which is set to 0x00000000, and the version-specified data field is unused, and ignored by the receiver. Right after the source connection ID field, there is a list of supported versions by the sender of the packet. The receiver of the packet will choose one of the versions and send a new packet with the chosen version. If there is no supported version on the packet the receiver must ignore that packet. When the receiver sends the packet to the sender, must change the source connection ID to the

destination connection ID of the packet received, to inform the sender that the packet was not generated by a possible attacker.

2.2.2 QUIC: A UDP-Based Multiplexed and Secure Transport

In this subsection we will analyze the first version of QUIC that is presented in the RFC9000 [18]. This version of the protocol follows the properties presented above, but it is important to analyze the particular characteristics of this version of the protocol.

Overview. QUIC integrates the TLS 1.3 handshake, although with some slight changes to permit data to be exchanged with fewer RTT possible. This is done using for example the 0-RTT mechanism, which allows the client to send data in the first RTT if it has already established a connection with the server. Applications exchange data using QUIC connection streams with limited size, these streams can be unidirectional or bidirectional. The protocol also provides a recovery mechanism for packet loss and also has a flow and congestion control mechanism. It is important to emphasize that this version just enables the client to migrate to a new network address.

QUIC Streams. Streams in QUIC are lightweight, ordered and bidirectional or unidirectional sequences of bytes and are used to exchange data from the two endpoints. The idea of these streams is to avoid head-of-line blocking and provide a way to multiplex data. In QUIC, a loss of a packet does not imply the block of the entire connection as it happens in TCP, in fact, the loss of a UDP packet only affects the streams that are carried by that packet. Streams are identified by the stream ID, which is carried in the stream frames. These streams are implicitly created upon sending bytes in a stream that does not exist yet. The streams are closed in two ways, the first one receives a FIN on the last frame, and the second one is defined by the client or the server when the stream is no longer needed. Although QUIC is reliable, data can't be transferred or retransmitted by cancelled streams. Each stream is flowed control and the available bandwidth must be divided between the multiple streams.

QUIC Flow Control. QUIC flow control is mainly controlled by frames. To control the amount of data sent, the protocol has two frame types that control the maximum of data that can be sent in the connection. The MAX_DATA frame is responsible for limiting the amount of data that can be sent in the connection, the other type of frame is the MAX_STREAM_DATA which limits the data that can be sent in a particular stream. Other types of frames are used to control the flow of the data, such as the STREAM_DATA_BLOCKED frame which is used to inform the peer that the stream is blocked, and the DATA_BLOCKED frame which is used to inform the peer that the connection is blocked. With these frames, the protocol is capable of controlling the sender and receiver flow of data.

QUIC Connections. To establish a connection QUIC, combines the cryptographic protocol with the transport handshake to set up a secure connection between two endpoints. The protocol offers different mechanisms, QUIC's initial handshake (1-RTT), 0-RTT handshake, and the 0-RTT resumption handshake.

Initial Handshake: A client with no connections before to the server, which means zero information about the servers, starts the handshake with a message called *inchoate client hello (CHLO)*. This message makes the server respond with a *reject message (REJ)* containing the following information:

1. a server config that includes the se4rver's long-term Diffie-Hellman public value;
2. a certificate chain authenticating the server;
3. a signature using the server's private key that can be verified using the server's certificate;
4. a source-address token that contains the client's IP address and a server timestamp;

Finally, the client validates the server's certificate chain and signature, and sends a *complete client hello (CHLO)* message to the server, with the client's ephemeral Diffie-Hellman public value.

Final Handshake: After *complete CHLO* the client is capable to start sending encrypted data as it can calculate the initial keys for the connection with the public value received in the *REJ* reply. Upon receiving the *server hello (SHLO)* message the client knows that the handshake is complete and changes the keys to the final keys. These final keys are calculated using the client's ephemeral Diffie-Hellman private value and the server's ephemeral Diffie-Hellman public value, sent in the *SHLO*. This message is encrypted with the initial keys. On the server side upon receiving the message of *complete CHLO* the server can calculate the final keys. The server can also send the *REJ* data encrypted immediately after using the final keys to encrypt that data. The client must cache the server's config information and the source-address token, to use in the next connection, starting instantly with the complete CHLO.

Version Negotiation: The version negotiation is done by the client proposing a version to use in the connection, there are two possible scenarios: a server does not support the version proposed by the client and sends a version negotiation packet back to the client increasing one RTT to the normal handshake, or the server supports the version proposed by the client and the handshake precedes normally.

Authentication and Encryption. QUIC packets are fully authenticated and encrypted, except for a few early handshake messages, but some fields in the packet header are

important to not encrypt due to some proposes. One of the fields is the packet number, used for packet loss detection and reordering, some flags are not encrypted to avoid some overhead, and the connection ID goes in plaintext for connection identification, for instance, load balancers can use this field to locate the connection state and route the packets to the correct server, and finally, the version field and diversification nonce are only presented in early packets. Some of these fields can tamper and the state of the connection can be different on the client and server side, the problem is detected by some failure in the decryption of the packet, and the connection is closed.

Foward Error Correction. The Forward Error Correction, also known as FEC, is a mechanism that adds redundant information in the data payload of the packets to recover from a packet loss without the need for retransmission. This mechanism, for instance, works in a group of 10 packets sent and on 9 packets received, the receiver can recover the lost packet using the redundant information. This mechanism can be enabled or disabled by the client and server, and the client can also choose the amount of redundant information to be added to the packets.

Congestion Control. QUIC does not have a specific congestion control mechanism, each implementation can use the one that is more suitable for the application. Three algorithms are mainly deployed as referred to in the study [16]: New Reno, CUBIC and BBR.

New Reno: The New Reno congestion control is very close to the TCP Reno with some slight differences. It consists of the slow start phase, like TCP Reno. When a packet is lost or packets are repeatedly acknowledged, it enters the fast recovery phase, which consists of retransmitting the lost packet or packets. If all the packets are retransmitted with success the congestion window maintains the same and increases by one MSS for each new ACK received, if the packets are not retransmitted with success the congestion window is set to half of the current value of the congestion window, then the window increases linearly until the next packet loss and triggering the same process again.

CUBIC: The CUBIC algorithm is similar to Reno, and has the same slow start phase, but the fast recovery phase is slightly different. The fast recovery is triggered the same way as explained above and sets the congestion window to half of the size, and the increase of the congestion window is accelerated with a cubic function.

BBR: This algorithm is called Bottleneck Bandwitdh and Round-trip propagation time and is defined by the Bandwitdh Delay Product (BDP). The BDP is the maximum amount of data that can be in the network at any given time. The BDP is calculated with the function $BDP = Bandwidth \times RTT$. Is not possible to measure the bandwidth and RTT at the same time, but it is calculated estimated values.

Migration. A migration is a mechanism implemented by QUIC that solves the problem of TCP connections not supporting IP address changes. For instance, if an endpoint changes its IP address, the connection in TCP can no longer be used, and a new handshake can be performed. So QUIC utilizes connection identifiers, exchanged during the handshake, for the endpoint not to rely on IP address, and giving the possibility of IP address changes restores the connection through the identifier without any previous handshake, making the process more efficient.

A migration is only possible if the connection has been properly initialized through a handshake. The ability to migrate can be disabled, and in such cases, packets arriving from different IP addresses must either be dropped or validated for migration. Servers never initiate migrations, and probing new network paths involves specific packet types exempted from congestion control.

To initiate connection migration, non-probing packets are sent, and both endpoints validate each other's addresses. Connection rates to unvalidated addresses are limited to prevent amplifying attacks and failed validations result in reverting to the last validated address. After migration, the migrating endpoint resets its congestion controller, RTT estimate, and ECN (Explicit Congestion Notification, which is a code point in the IP header that informs the sender about congestion building at routers) capability to avoid old network path characteristics affecting the new path.

For privacy reasons, connection IDs cannot be reused, even during path probing or with migrated endpoints. Servers may advertise preferred IP addresses, and clients can connect to these addresses through normal procedures. Server migration is not supported, and clients should validate both server addresses if migrating to a preferred address. The server must validate the new address as well.

2.2.3 QUIC Implementations

The QUIC IETF working group oversees various implementations of the QUIC protocol, with over twenty different implementations in different programming languages. These implementations aim to support QUIC specifications up to draft 29 or later and participate in interoperability testing. Some notable implementations include:

aioquic: Python-based QUIC implementation, that has several features: QUIC API that follows the "bring your own I/O" pattern (leaves I/O operations to the API user), support for HTTP/3, and a minimal TLS 1.3 implementation with encryption operations implemented in C linked to OpenSSL.[1]

AppleQUIC: Limited information is available; programmed in C and Objective-C, likely related to OS X and/or iOS.

[1] https://github.com/aiortc/aioquic

ats (Apache Traffic Server): Integrated into Apache project, supports multiple SSL libraries, provides limited testing tools in C++.[2]

Chromium's QUIC implementation: Integrated into Chrome, requires Chromium source code for testing in C and C++. Has few public resources and documentation.

F5: Part of F5 TMOS, a real-time operating system, supports draft 32 in C, no more information is available in the QUIC working group.

Haskell QUIC: Developed in Haskell, has some articles and blogs published, has a published article about testing QUIC servers with the h3spec, interesting tool for future testing.[3]

kwik: Java-based QUIC client implementation, developed by Peter Doornbosch. Started as a client library, but since May 2021 it also supports server functionality.[4] It also provides a link for the HTTP/3 implementation in Java, that runs on top of kwik implementation, which name is Fulpke, that can be used for future protocol testing.[5]

lsquic: C-based QUIC and HTTP/3 implementation that works on Linux, macOS, FreeBSD, and Windows supports the latest drafts. It also refers to an event-driven architecture, understands Apache rewrite rules, and has friendly admin interfaces, among many other traits.[6]

msquic: Microsoft's cross-platform C-based QUIC implementation with superior optimization, and support for the latest drafts. They present several features that differentiate this implementation from the other ones, such as optimization for client and server, optimization for maximal throughput and minimal latency, asynchronous IO, receiving side scaling support, and UDP sending and receiving merging support.[7]

mvfst: C++ implementation of Facebook tested at scale on mobile platforms and servers, support for large-scale deployment. It was built to perform and adapt to both Internet and data centre environments.[8]

neqo: Rust-based implementation with limited documentation, updated to support up to draft-7 through version 1 and version 2 of QUIC.[9]

ngtcp2: C-based QUIC implementation with support for the latest drafts, and a minimal low-level API.[10]

[2] https://github.com/apache/trafficserver/wiki/HTTP-3-Documentation
[3] https://github.com/kazu-yamamoto/quic
[4] https://bitbucket.org/pjtr/kwik/src/master/
[5] https://bitbucket.org/pjtr/fulpke/src/master/
[6] https://github.com/litespeedtech/lsquic
[7] https://github.com/microsoft/msquic
[8] https://github.com/facebook/mvfst
[9] https://github.com/mozilla/neqo
[10] https://github.com/ngtcp2/ngtcp2

nginx: C-based QUIC implementation integrated into Nginx, supports the latest drafts.[11]

nginx-cloudfare: C-based QUIC implementation integrated into Nginx, based on quiche implementation, and is optimized for Cloudflare's edge networks.

picoquic: Minimalist C-based QUIC implementation with test tools and ongoing documentation progress.[12]

quant: C-based QUIC implementation for research purposes, supports traditional POSIX platforms (Linux, MacOS, Windows, etc.), lacks HTTP/3 functionality. It is a research project and is not intended for production use.[13]

quiche: Rust-based implementation for Cloudflare's edge networks, features a low-level API for handling packets and connection state, leaving the IO work to the application.[14]

quic-go: Go language implementation with HTTP/3 support, Datagram Packetization Layer Path MTU Discovery (DPLPMTUD), and a standardized logging format schema.[15]

2.3 Previous Performance Studies

In this section, we delve into a comprehensive overview of the preceding performance analysis conducted on the QUIC protocol, drawing comparisons with both TCP and various implementations of the protocol. The aim is to provide a nuanced understanding of the dynamics influencing the efficacy of these transport protocols.

To initiate this exploration, an examination of diverse metrics utilized in prior studies will be presented. These metrics serve as the yardstick by which the performance of transport protocols is assessed. Each metric, chosen through previous research endeavours, will be expounded upon, unravelling the layers of evaluation and shedding light on the nuanced aspects that contribute to the overall performance dynamics.

The discourse will illuminate the real-world implications and contextual relevance of these metrics. By doing so, we aim not only to comprehend the quantitative disparities but also to discern the qualitative dimensions that differentiate the protocols under scrutiny.

2.3.1 QUIC vs TCP

According to some studies [19], QUIC outperforms TCP in scenarios with 0-RTT connection establishment, recovering from packet loss, and also with scenarios where there are

[11] https://hg.nginx.org/nginx-quic/
[12] https://github.com/private-octopus/picoquic
[13] https://github.com/NTAP/quant
[14] https://github.com/cloudflare/quiche
[15] https://github.com/quic-go/quic-go/tree/master

bandwidth varies. However, there are no perfect scenarios and this article emphasizes that TCP is still better in situations with out-of-order packet delivery and in older phones with worse computing resources. This study also concludes that the usage of a TCP proxy can approximate the performance of TCP to QUIC, under scenarios with low latency and packet loss.

Fairness. Other studies were conducted to analyze the fairness among the TCP and QUIC connections. According to the studies QUIC is not fair with TCP, despite using the same congestion control algorithm, QUIC consumes twice the bandwidth compared to one TCP connection. This can only be mitigated if compared with one QUIC connection with two TCP connections. QUIC competing with n TCP connections should consume approximately $2/(n+1)$ of the bandwidth, this was not observed in the study [19] where QUIC still consumed over 50% of the bandwidth. They conclude that despite TCP and QUIC both using CUBIC as the congestion control algorithm, the second one increases its window size faster in terms of slope or frequency of increasements, than the first one, getting the reason why QUIC consumes more bandwidth than TCP.

Page Load Time and Throughput. Another study from 2022 analyzed the performance of QUIC and TCP in terms of page load time. The study used quiche from the client side and Cloudflare's implementation from the server side. The study concluded that QUIC outperforms TCP in terms of page load time, in a range of 8% to 23% faster than TCP, which can be deeply explored in [28]. The study also analyzed the throughput of the protocol implementation and concluded that QUIC outperforms TCP in terms of throughput, from approximately 12% to 27%. It summarizes that QUIC outperformed TCP by reducing the acknowledged frequency and using larger packets.

2.3.2 LSQUIC vs AIOQUIC vs NGTCP2 vs TCP case study.

These implementations were compared in a case study performed in 2021 [15]. This study observed that LSQUIC outperformed TCP in terms of connection establishment time, except in ideal scenarios (scenarios with low latency and packet loss), despite having the longest handshake compared with other ones. NGTCP2 was the implementation that demonstrated improvements in some scenarios, and other scenarios were not so but showed a consistent improvement in the handshake duration. AIOQUIC was the implementation that showed the worst performance in the longer connections with higher delays, but the best performance in the shorter connections with lower delays. The study also emphasizes that QUIC implementations send more packets than TCP due to the smaller packet payload sizes. Finally, concluded that the server overhead for QUIC implementations is higher than TCP.

2.3.3 Performance Metrics

The performance of a transport protocol can be evaluated by measuring different metrics, which can involve the network or the endpoint delays. In this subsection, we highlight some metrics that can be relevant to this study.

2.3.3.1 Delays

A packet when travelling through the network is exposed to different resources, such as routers, switches, and links, and each of these resources can cause a delay in the packet. In this subsection, it is exhibited some of the nodal delays that a packet can be exposed to [22].

Processing Delay. The processing delay is the time that a node has to process the packet. This delay can be important to understand how much overhead the protocol is exposed and some metrics can be concluded.

Queuing Delay. The queuing delay is the time that a packet experiences when it waits to be transmitted to the link. In the case of the QUIC, this can verified by the time the packet enters the sender buffer until the time it is transmitted. This can be a good measurement to understand the performance of the protocol.

Transmission Delay. Transmission delay is the time that a packet takes to be transmitted in the link. This delay is not so important to be considered in the study.

Propagation Delay. The propagation delay is the time that a packet takes to travel from the sender to the receiver. This delay in separate is not so important to be considered because it is more related to a network layer measurement and does not affect how the transport protocols perform due to their possible design choices.

2.3.3.2 Latency

Network latency is the delay presented in communication between two endpoints. It represents the time that a packet takes to travel across the network [41]. The latency is an important measure of the performance of the transport protocol to understand how much time the protocol takes to send and receive a packet. The latency can be divided into two different types: the round-trip time (RTT) and the one-way latency, but the most interesting to evaluate the performance of a transport protocol is the RTT.

2.3.3.3 Average Throughput

Network throughput is the average volume of data that can travel in the network for a given file transfer for instance [42]. This network metric would be interesting to analyze

to understand how much data the protocol can send and how the protocol manages the available computational resources. This can be related to the congestion control algorithm and the fairness among multiple connections in a single bottleneck link.

2.3.3.4 Packet Loss Rate

The packet loss rate is the percentage of packets lost in the network in comparison with the packets sent. This will lead to a congestion control reaction, triggering some mechanisms of packet retransmission. The majority of the QUIC implementations, implement mechanisms of packet loss detection. The packet loss detection is important to trigger the fast recovery phase of the congestion control algorithm and to retransmit the lost packets [43].

2.3.3.5 Jitter

A jitter is the variation in which a packet is sent and received in the network. Normally jitter is associated with terms like latency. The congestion of the network or poor computational resources can be one of the causes of this phenomenon. This metric is not so important to be included in the analysis of the performance of the protocol but can be interesting to understand how the network is behaving in the performance tests.

2.3.3.6 Time To First Byte

The Time To First Byte (TTFB) is the time between the request of the resource until the first byte of the received response arrives, according to the definition of the metric [35]. This metric involves many phases, such as: redirect time, the start of the service if it is not already started, the DNS lookup, the connection establishment involving TLS negotiation for the secure connection, the request and the response until the first byte arrives. This metric can substitute the page load time in the performance analysis because it dictates the same performance aspects. A good score in this metric can be 0.8 seconds or less according to the study [35].

2.3.3.7 Congestion Detection

Congestion Detection is the metric that detects if there is any congestion at any location point [30]. This can be divided in two different ways: Buffer occupancy which refers to how much the memory of the sender buffer is occupied in comparison with the maximum size and the Packet Rate which consists of the number of sent packets in a specific time interval.

2.3.3.8 Channel Utilization

The study [5] considered channel utilization, a metric that evaluates the average rate of received packets in that connection vs the maximum link capacity. This metric can be an

interesting measurement of how the protocol manages the computational resources that as available.

2.3.4 QUIC Performance Evaluation Tools

There are multiple performance studies of QUIC, we now present some of the tools and their key features, that were created to evaluate some type of performance aspect(s) of the QUIC implementations.

Mahimahi. Mahimahi [25] is a framework designed for accurately recording and replaying HTTP traffic under emulated network conditions. It offers several features to facilitate the web service evaluation process, including isolation using multiple instances of its shells to run concurrently without interference; emulation of the network introducing packet delays over the DelayShell, emulation over network links using the LinkShell and packet loss over LossShell; it also offers a performance metrics visualization over the different shells.

qperf. Qperf is a network tool with little documentation that measures bandwidth and latency between two nodes. It is very similar to the iperf tool, but instead of using TCP, it uses the quicly protocol implementation, for more information visit [4].

quic_perf_eval. This is a repository that contains a set of scripts to perform *Performance Evaluation of Various QUIC Implementations: Performance and Sustainability of QUIC Implementations on the Cloud* [31] experimental of study. The tool has incorporated the ability to measure the throughput, latency, and packet loss rate of the QUIC implementations such as lsquic, mvfst, picoquic, Quiche, and quic-go, in different network conditions.

Table 2.1 compiles the main features of the aforementioned tools, namely Fundamental Performance, which refers to throughput and latency metrics. Several QUIC Implementations evaluate the tool's versatility in testing various QUIC implementations, including novel variants. Mahimahi and quic_perf_eval are marked with question marks in this category as they can assess multiple implementations but not all. Then Manipulate Network Properties refers to the capacity of the tool to emulate the network conditions. quic_perf_eval earns a question mark here, as it offers pre-implemented network scenarios but lacks user freedom in setting custom conditions. Content and Parallel Fluxes highlight the tool's capability to create competing fluxes between different connections passing at a single point in the network. Finally, Automation indicates if the tool is capable of deploying the experiment by itself. Each tool has its features and limitations, and to create a comprehensive and accurate performance evaluation, we propose to cover all features using QuicSand.

	Fundamental Performance Metrics	Several QUIC Implementations	Manipulate Network Properties	Contention and Parallel Fluxes	Automation
qperf	✓	X	X	X	X
Mahimahi	✓	?	✓	X	X
quic_perf_eval	✓	?	?	X	X
QuicSand	✓	✓	✓	✓	✓

Table 2.1: QUIC Performance Evaluation Tools

2.3.5 Security Analysis

Developing a new protocol is a hard task, and that is made even harder when there is the need to develop a secure protocol. Google developers have been working on QUIC for a long time, and they have been improving the protocol to make it more secure. However, there are still some security issues that need to be addressed. In this section, we will present some of the most relevant security issues of QUIC.

2.3.5.1 QUIC security model

In the article *How Secure and Quick is QUIC? Provable Security and Performance Analyses* (Lichev, 2015) [24] the authors define a security model for Google developer's QUIC protocol. The model consisted of many security properties presented below.

Firstly they defined an Authentication and Confidential Channel Establish Model (ACCE) that is an extension of the security definitions used in proving TLS, modified to fit the new QUIC security challenges. The model maintains the goal of achieving the CIA triad (Confidentiality, Integrity and Availability) plus some protection against attacks such as IP spoofing, replay attacks, and packet-level attacks.

Secondly, the model considers a very powerful attacker, capable of knowing all server's public keys, traffic control inspection, misordering packets, compromising data packet integrity by bit-flipping attacks, dropping packets and also capable of performing denial of service attacks (DoS).

Another property that the model considers is Forward Secrecy, which means that if a server is corrupted by the attacker at any possible time, the amount of sensitive data that is compromised is limited to the time that the attacker has access to the server.

2.3.5.2 QUIC security issues

In the article [24] the authors highlight novel security issues that QUIC faces, in particular during the handshake process. Here are the security issues highlighted:

Denial of Service (DoS) attacks. QUIC is vulnerable to DoS attacks, due to some of the features that it provides, like unprotected fields and public information in the server

and client in the handshake process. The results obtained were that the attacker can prevent both sides from establishing a connection likewise.

Lack of Foward Secrecy. QUIC does not guarantee the forward secrecy property by the fact that the protocol does not force the usage of ephemeral keys, such as TLS-DHE for instance. If the server has computational resources to support the computational overhead of the ephemeral keys, the forward secrecy property can be achieved.

Replay Attacks. According to the study [24], QUIC is vulnerable to two different types of replay attacks: the first one, name as Server Config Replay Attack consists of the attacker sending the server's public config to the client, this will cause a new key establishment in the client side without server's knowledge, this will lead to an impossibility of the server to decrypt the client's messages, resulting in the server dropping client's packets. The second one, defined as the Source-Address Token Replay Attack by the authors, results in replaying the source-address token to the server on behalf of the client, this will cause the server to accept the various connections and establish new initial keys or even final keys, possibly resulting in a DoS due to the server's resources exhaustion.

Trade-offs Between Latency and Security. The protocol as a whole permits some trade-offs between latency and security compared to TCP with TLS. So the authors conclude that QUIC benefits from the trade-off when there are no attackers.

Amplification Attacks. The QUIC protocol is vulnerable to amplification attacks, where an attacker can send a small packet to the server and the server will respond with a large packet to the client. This can be used to amplify the attack and cause a DoS attack. A study from 2021 [15], informed that QUIC has a countermeasure to mitigate these attacks imposing sending limits to the client before a validation of the client's connection.

2.3.6 Discussion

Considering all the studies presented, we conclude that an additional effort needs to be made to help users evaluate their implementations of QUIC. On one hand, the previous tools developed can evaluate the fundamental performance metrics highlighting the Mahimahi for the ability to evaluate page load time and to emulate some network conditions that the other tools are not capable of. On the other hand, none of the tools are capable of giving the user an automatic and comprehensive evaluation of the performance of the QUIC implementations and evaluating contention and parallel fluxes. Due to a lack of tools that simplify this process, we propose to develop QuicSand which will gather existing and novel features and metrics to guarantee a comprehensive and robust performance evaluation of the various QUIC implementations.

2.4 Network Emulation Tools

Our goal is to study the performance of a network protocol and compare the difference between variants and implementations of QUIC protocol. However, these concepts can

be highly dependent on particular network topologies and conditions. To address this aspect it is necessary to evaluate a protocol implementation in a large-scale distributed system to produce the closest real-world evaluation conditions possible. The issue is that testing a real distributed system would be hard, slow and costly. Therefore tools that allow network emulation are a solution possible to mitigate this issue.

Network emulation involves running a real system against a model of the network that replicates real-world behaviour. This model includes network topology and its elements, such as switches and routers, along with their internal behaviour. By doing so, network emulation allows researchers and practitioners to conclude the behaviour of real systems in specific scenarios, rather than relying solely on theoretical models, abstracting an application from having a notion of the real network to observe the emulated network.

There are plenty of network emulation tools, some centralized others, decentralized, of particular significance with different approaches such as user-space or container-based, most notably with some linked-level emulation capacities. Consequently, the selection of the tool will be crucial to get the most approximate results for how a realistic distributed system could behave in a realistic environment.

2.4.1 *NetEm*

NetEm [38] is a tool included in the Linux kernel that offers network emulation over an IP network interface. It allows users to add delay, packet loss, duplicated packets, reordering packets, and corrupted packets to the network.

2.4.2 *ModelNet*

ModelNet [40] is a network emulation tool that provides link-level emulation capabilities, such as delay, bandwidth, and packet loss. Its main focus is on large-scale systems and it is designed to be scalable and flexible. It relies on a centralized approach, where a single machine emulates the network for all the nodes in the system.

2.4.3 *Kollaps*

Kollaps is a container-based tool that provides at kernel mode static and dynamic network emulation [14]. It's a tool that is focused on decentralized and dynamic topology emulation, addressing the limitations of existing network emulation tools and providing a solution for accessing the impact of network properties on large-scale distributed applications.

Kollaps addresses these challenges from various design principles:

- **Focus on End-to-End Properties**. The tool recognizes that the end-to-end properties (e.g., latency, bandwidth, packet loss, and jitter) are more important for the behaviour of the distributed system than the internal network topology. Therefore, the tool provides a simple way to specify the end-to-end properties of the network, while abstracting the internal network topology.

- **Decentralized Maintenance**. *Kollaps* maintains its emulation model in a fully decentralized manner, allowing the emulation to scale with the number of machines in the application. This decentralized approach ensures that the emulation remains accurate without sacrificing scalability.
- **Dynamic Adaptability**. The ease of doing quick changes in the network such as link additions, removals and background traffic turns out to be crucial for an accurate emulation of real-world network conditions.
- **Container-based Emulation**. The tool integrates container orchestration platforms such as Docker Swarm and Kubernetes, providing a simple evaluation through the use of unmodified containers in the emulated network environment.

These are the principles leveraged by *Kollaps* that provide the ease of usage of a network emulation tool to provide accurate and reliable results for QUIC implementations performance evaluation.

2.4.4 Discussion

The network emulation tool will have a crucial role in the development of our tool. Despite having a key factor but not being the main focus of the work we chose *Kollaps*, due to its flexibility and ability to produce all network scenarios and conditions that we want to implement in our tool, compared with *NetEm* a more link-level emulation tool that is not capable of providing bandwidth emulation, neither network topology dynamics, and *ModelNet* that does not provide the flexibility and scalability considering the decrease of accuracy when the number of applications nodes increases [14].

2.5 Summary

This chapter presented the main concepts of the QUIC protocol, starting with a background study of the transport layer protocols TCP and UDP, then a quick overview of the SPDY protocol, which was one of the motivations for developing the QUIC protocol. This chapter also presented the main features of the QUIC protocol, such as the packet structure, the connection establishment, the streams, the flow control, the congestion control, and the migration of connections. We then also presented the main implementations of the protocol. This aims at providing a better understanding of the problem we tackle in this work, and to further identify some relevant performance metrics and indicators. Then comes a security analysis and issues encountered in some QUIC implementations. Finally, the chapter ends with network emulation tools that will be helpful for the development of realistic network scenarios and conditions. In the following chapter, the methodology, metrics, and experimental environment will be presented.

3

Future Work

As presented previously, this work intends to study the performance of the different implementations with the support of a novel tool and methodology that can ease the usability and the readability of the metrics and indicators to the user.

The following sections will present the initial idea proposed for the work. Section 3.1 refers to the overall methodology, followed by Section 3.2 which presents the initial proposed tool design and features. Then, Section 3.3 presents the measurement study plan, which identifies the metrics that we pretend to use to measure to gather results, as well as the different scenarios that we want to test. Finally, we provide a calendarization (Section 3.4) that schedules the tasks that we pretend to accomplish until the final delivery of the work.

3.1 Methodology

This section will present the metrics and the network scenarios that will be considered in the development of the study about the performance of the different QUIC implementations and variants.

Metrics for performance evaluation. For the performance evaluation of the different QUIC implementations, we will consider the following metrics:

- **Throughput:** The amount of data that can be transmitted from the source to the destination in a given period. It will be measured in bits per second by sending a file from the server to the client;
- **Latency:** The time that a packet takes to travel from the source to the destination. It will be measured by sending a message from the server to the client and calculating the time that the first byte of the packet takes to arrive at the destination;
- **Packet Loss:** The percentage of packets that are lost during the transmission. It will be measured by sending a variable size file from the server to the client and calculating the number of packets sent per the number of packets acknowledged;

- **Fairness:** The ability of the protocol to share the network resources fairly among different connections. For instance, a fair implementation implies that if two clients have a connection for the same server, the bandwidth available for both clients needs to be as close as possible, independently of the transferred object size. This metric will be measured using a network scenario explained below;

- **CPU and Memory Usage:** The amount of CPU and memory that the protocol consumes. It will be measured by running the protocol and monitoring the CPU and memory usage.

- **Time To First Byte:** The time that the protocol takes from the first packet sent to the first byte received. Contrary to the page load time it will not be influenced by the object size.

Network Scenarios. For the performance evaluation of a transport protocol in general it is important to consider how the protocol behaves in different network conditions. Therefore, it will be helpful to have tools that can simulate these different network conditions, because it is expensive and time costly to test the protocol in a real network. The following network scenarios will be considered:

- **Optimal:** This scenario will consider a network with high bandwidth, low latency and no packet loss rate. In this scenario, it will be measured the maximum and average throughput the implementation can achieve, the minimum and average latency of the connection, will be also measured the CPU and memory usage of the protocol and the TTFB. The following scenarios will consider the same metrics and indicators as the ideal scenario but with different network conditions.

- **High latency:** This scenario will consider a network with high bandwidth and no packet loss rate, but with high latency. The idea is to understand the behavior of the protocol when the latency is high with the other conditions as favorable as possible to the performance of the protocol.

- **Limited Bandwidth:** The network in this scenario will have low bandwidth, low latency and no packet loss rate. The main focus is to understand how the protocol reacts to a bandwidth limit and how the protocol manages the throughput in such conditions.

- **Client mobility:** The client mobility will consist of changing the client's IP address during the connection, triggering the migration mechanism of the QUIC protocol. Creating this scenario will allow us not only to evaluate the time the protocol takes to migrate the connection to the new IP address but also the amount of CPU and memory usage during the operation, for example.

- **Network Congestion:** This scenario will consist of the congestion of the connection path saturating the network with some packet injection. The idea is to understand the behavior of the protocol when the network is congested and to test how the protocol congestion control algorithm reacts to the congestion of the network.
- **Varying packet size:** Some applications might want to send small packets, while others might want to send large packets. Therefore, it is important to test how the different implementations behave when the packet size varies. In this scenario, it will be measured the the same metrics and indicators as the other network conditions.
- **High packet loss:** This scenario will consider a high packet loss rate, which will affect the performance of the protocol, and it will evaluate how the implementation can manage the packet loss and retransmission of packets.
- **Network link failure and instability:** Last but not least, this scenario takes into account how the variants react to the network link failure and instability, involving connection reestablishment and some packet loss.
- **Competing flows:** This scenario will consist of having multiple clients sending data to multiple servers, passing through the same network point. It will test the fairness and how the congestion control algorithm manages the network resources.

3.2 Tool Architecture

The development of the performance evaluation tool is the main purpose of the work, where the goal is to help the user have an easy and mostly automatic way to test the existing, or new implementations knowing which one fits the best for the user's application. The tool will need to have client and server implementations for the languages Python, C, Go, C++, Haskell and Java to try to embrace all possible implementations. The tool is intended to follow these design principles:

- **Capture Mechanism:** This is the mechanism that will give a tool's capacity of capturing and inspecting the packets involved in the performance testing environment.
- **Metrics Collection:** A principle that represents the capability of the tool to collect the metrics for future analysis.
- **Network Emulation:** The tool will have the capacity to emulate different network conditions, as presented in the previous section, to test the performance of the different implementations in different network scenarios.
- **Logging and Exporting:** The tool will capture the logs of performance data and exporting for offline analysis. Data will be exported in a defined format to be able to analyze the results in a common tool.

- **Analysis and Visualization:** Implementation of the performance evaluation that will gather all logs and data to present the results to the user.
- **Compatibility and Extensibility:** The capacity of the tool to be compatible with different implementations and to be extensible to new implementations that might appear in the future.
- **Documentation and Support:** The tool will have documentation and support for the user's ease of navigation and usage.

The tool initial proposed architecture is illustrated in Figure 3.1. The architectural flow unfolds as follows: the orchestrator component starts receiving the user's input configuration alongside the desired QUIC implementation. The orchestrator then assumes several tasks including generating all corresponding input and implementation binaries, building the docker client(s) and server(s) docker containers pushing the generated binaries to compile on the respective containers, and starting the network emulator. After all components are ready, the orchestrator starts the performance evaluation tests, by sending requests to the containers that will generate logs. The communication will be controlled through the network emulator tool previously deployed by the orchestrator and prepared to emulate the different network scenarios. The logs will be collected in the docker containers and pushed to the orchestrator for further analysis and performance evaluation at the end of the measurement campaign.

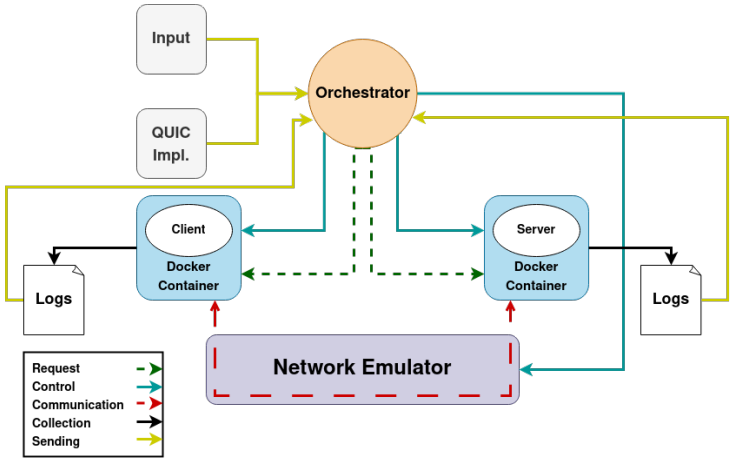

Figure 3.1: QuicSand Architecture Design

3.3 Measurement Study Plan

The plan for the measurement study will focus on testing all the different implementations of QUIC, in the different network scenarios, presented above using the tool developed. The idea is to use the set of test scenarios and conditions that the tool developed will offer to the user.

For the purpose of the work, it will be considered to test all the QUIC implementations available at the QUIC Working Group (discussed in Section 2.2.3), to demonstrate not only the performance of each implementation but the concretization of the design principles proposed.

Turning the attention to the network scenarios, the study intends to test the scenarios: optimal, high latency, limited bandwidth, client mobility, network congestion, varying packet size, high packet loss, network link failure and competing flows. These scenarios are considered to be integrated as the automatic performance evaluation tests.

The final plan will consist of having a group of results tested for each implementation and making a final analysis and comparison between each one presenting graphics and histograms to a better read of the results.

3.4 Calendarization

The following grant chart (Figure 3.2)presents the plan for the tasks that will be accomplished in future work. From the refinement of the design model to the implementation, crossing the testing and experiments and finishing with the writing of the dissertation and the article.

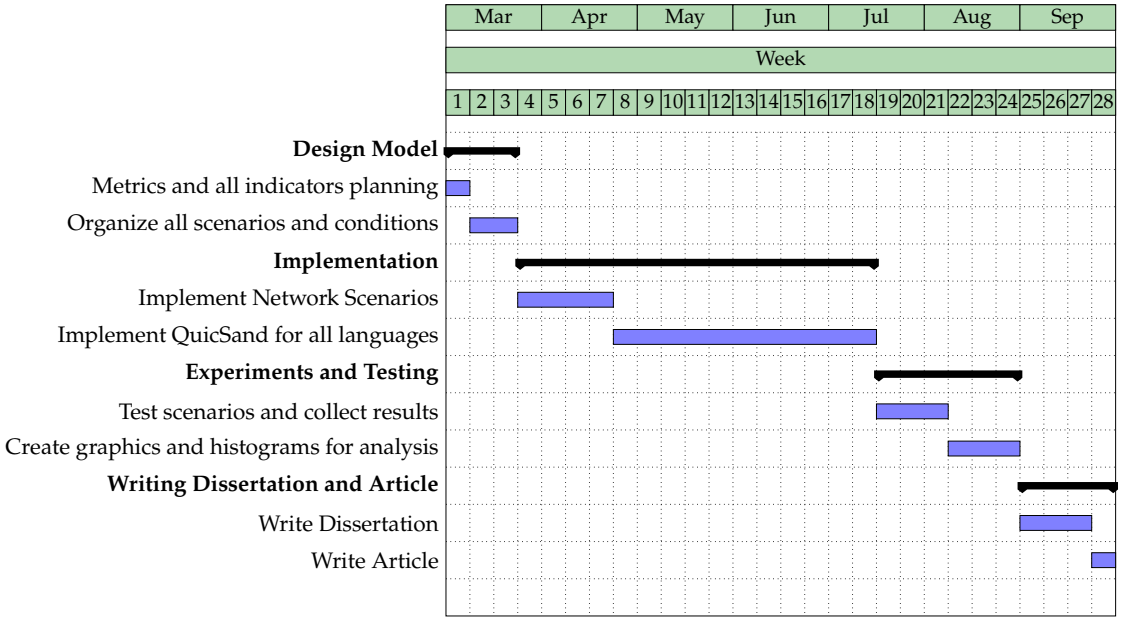

Figure 3.2: Calendarization of the work tasks

Bibliography

[1] *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826. 1982-11. DOI: 10.17487/RFC0826. URL: https://www.rfc-editor.org/info/rfc826 (cit. on p. 4).

[2] M. Belshe and R. Peon. *SPDY Protocol*. Internet-Draft draft-mbelshe-httpbis-spdy-00. Work in Progress. Internet Engineering Task Force, 2012-02. 51 pp. URL: https://datatracker.ietf.org/doc/draft-mbelshe-httpbis-spdy/00/ (cit. on p. 13).

[3] T. Berners-Lee and D. W. Connolly. *Hypertext Markup Language - 2.0*. RFC 1866. 1995-11. DOI: 10.17487/RFC1866. URL: https://www.rfc-editor.org/info/rfc1866 (cit. on p. 13).

[4] R. Bruenig. *A performance measurement tool for QUIC similar to iperf*. https://github.com/rbruenig/qperf. [Online; accessed 9-Fev-2024] (cit. on p. 26).

[5] G. Carlucci, L. De Cicco, and S. Mascolo. "HTTP over UDP: an experimental investigation of QUIC". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: Association for Computing Machinery, 2015, pp. 609–614. ISBN: 9781450331968. DOI: 10.1145/2695664.2695706. URL: https://doi.org/10.1145/2695664.2695706 (cit. on pp. 14, 25).

[6] Y.-C. Chen, S.-C. Tseng, and Y.-B. Hu. "Seamless Layer 2 Framework for Heterogeneous Networks". In: *2010 Fourth International Conference on Genetic and Evolutionary Computing*. 2010, pp. 598–601. DOI: 10.1109/ICGEC.2010.153 (cit. on p. 3).

[7] C. Cremers et al. "A Comprehensive Symbolic Analysis of TLS 1.3". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1773–1788. ISBN: 9781450349468. DOI: 10.1145/3133956.3134063. URL: https://doi.org/10.1145/3133956.3134063 (cit. on p. 12).

[8] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346. 2006-04. DOI: 10.17487/RFC4346. URL: https://www.rfc-editor.org/info/rfc4346 (cit. on pp. 4, 11).

[9] *Domain names - concepts and facilities*. RFC 1034. 1987-11. DOI: 10.17487/RFC1034. URL: https://www.rfc-editor.org/info/rfc1034 (cit. on p. 4).

[10] *Domain names - implementation and specification*. RFC 1035. 1987-11. DOI: 10.17487/RFC1035. URL: https://www.rfc-editor.org/info/rfc1035 (cit. on p. 4).

[11] A. E. W. Eldewahi et al. "SSL/TLS attacks: Analysis and evaluation". In: *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*. 2015, pp. 203–208. DOI: 10.1109/ICCNEEE.2015.7381362 (cit. on p. 11).

[12] *File Transfer Protocol*. RFC 959. 1985-10. DOI: 10.17487/RFC0959. URL: https://www.rfc-editor.org/info/rfc959 (cit. on pp. 4, 13).

[13] A. O. Freier, P. Karlton, and P. C. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. RFC 6101. 2011-08. DOI: 10.17487/RFC6101. URL: https://www.rfc-editor.org/info/rfc6101 (cit. on pp. 4, 11).

[14] P. Gouveia et al. "Kollaps: Decentralized and Dynamic Topology Emulation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387540. URL: https://doi.org/10.1145/3342195.3387540 (cit. on pp. 29, 30).

[15] C. Halme et al. "Performance analysis of modern QUIC implementations". In: (2021) (cit. on pp. 1, 23, 28).

[16] S. Hertelli, B. Jaeger, and J. Zirngibl. "Comparison of Different QUIC Implementations". In: *Network* 7 (2022) (cit. on p. 19).

[17] *Internet Protocol*. RFC 791. 1981-09. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791 (cit. on p. 4).

[18] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. 2021-05. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000 (cit. on pp. 15, 17).

[19] A. M. Kakhki et al. "Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols". In: *Proceedings of the 2017 Internet Measurement Conference*. 2017, pp. 290–303 (cit. on pp. 1, 22, 23).

[20] D. J. C. Klensin. *Simple Mail Transfer Protocol*. RFC 5321. 2008-10. DOI: 10.17487/RFC5321. URL: https://www.rfc-editor.org/info/rfc5321 (cit. on p. 13).

[21] S. Kumar, S. Dalal, and V. Dixit. "The OSI model: Overview on the seven layers of computer networks". In: *International Journal of Computer Science and Information Technology Research* 2.3 (2014), pp. 461–466 (cit. on p. 3).

[22] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach*. 7th. This is the 7th edition of the book. Pearson, 2016. ISBN: 978-0-13-359414-0 (cit. on pp. 3, 5, 6, 11, 24).

[23] J. Legatheaux. *Fundamentos de Redes de Computadores*. 1ª Edição Digital. Nova.FCT Editorial, 2018 (cit. on p. 13).

[24] R. Lychev et al. "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 (cit. on pp. 27, 28).

[25] R. Netravali et al. "Mahimahi: accurate record-and-replay for HTTP". In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, pp. 417–429. ISBN: 9781931971225 (cit. on pp. 1, 26).

[26] H. Nielsen et al. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. 1999-06. DOI: 10.17487/RFC2616. URL: https://www.rfc-editor.org/info/rfc2616 (cit. on pp. 4, 13).

[27] O. Okamoto, T. Sajima, and M. Maruyama. *Forwarding Media Access Control (MAC) Frames over Multiple Access Protocol over Synchronous Optical Network/Synchronous Digital Hierarchy (MAPOS)*. RFC 3422. 2002-11. DOI: 10.17487/RFC3422. URL: https://www.rfc-editor.org/info/rfc3422 (cit. on p. 4).

[28] S. Oran, A. Koçak, and M. Alkan. "Security Review and Performance Analysis of QUIC and TCP Protocols". In: *2022 15th International Conference on Information Security and Cryptography (ISCTURKEY)*. 2022, pp. 25–30. DOI: 10.1109/ISCTURKEY56345.2022.9931821 (cit. on pp. 1, 23).

[29] M. Seemann. *QUIC DEEP DIVE*. Paris P2P Festival. 2022. URL: https://www.youtube.com/watch?v=6SyDP7xKqZk (cit. on p. 15).

[30] B. Sharma and T. C. Aseri. "A comparative analysis of reliable and congestion-aware transport layer protocols for wireless sensor networks". In: *International Scholarly Research Notices* 2012 (2012) (cit. on p. 25).

[31] F. A. V. Sitepu. *Performance Evaluation of Various QUIC Implementations: Performance and Sustainability of QUIC Implementations on the Cloud*. 2022 (cit. on pp. 1, 26).

[32] *TCP 3-Way Handshake | Computer Networks*. https://workat.tech/core-cs/tutorial/tcp-three-way-handshake-in-computer-networks-yoo7331910lh. [Online; accessed 29-Dec-2023] (cit. on p. 5).

[33] *TCP Connection Termination*. [Online; accessed 12-Fev-2024]. URL: https://www.javatpoint.com/tcp-connection-termination (cit. on p. 9).

[34] M. Thomson. *Version-Independent Properties of QUIC*. RFC 8999. 2021-05. DOI: 10.17487/RFC8999. URL: https://www.rfc-editor.org/info/rfc8999 (cit. on p. 15).

[35] *Time to First Byte (TTFB)*. https://web.dev/articles/ttfb. [Online; accessed 6-Fev-2024] (cit. on p. 25).

[36] *Transmission Control Protocol*. RFC 793. 1981-09. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793 (cit. on pp. 4, 6).

[37] *User Datagram Protocol*. RFC 768. 1980-08. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768 (cit. on p. 4).

[38] *Using NetEm to Emulate Networks - SRT CookBook*. en. URL: https://srtlab.github.io/srt-cookbook/how-to-articles/using-netem-to-emulate-networks.html (cit. on p. 29).

[39] *Viewer HTTPS configuration - secure content delivery with Amazon Cloudfront*. URL: https://docs.aws.amazon.com/whitepapers/latest/secure-content-delivery-amazon-cloudfront/viewer-https-configuration.html (cit. on p. 12).

[40] K. V. Vishwanath et al. "ModelNet: Towards a datacenter emulation environment". In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. 2009, pp. 81–82. DOI: 10.1109/P2P.2009.5284497 (cit. on p. 29).

[41] *What is network latency? | AWS*. https://aws.amazon.com/what-is/latency/. [Online; accessed 2-Fev-2024] (cit. on p. 24).

[42] *What is network throughput? | AWS*. https://aws.amazon.com/compare/the-difference-between-throughput-and-latency/. [Online; accessed 2-Fev-2024] (cit. on p. 24).

[43] K. Yamamoto. *Implementing QUIC in Haskell*. Accessed: 11-2-2024 (cit. on p. 25).