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

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon

April, 2025

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

Examination Committee

Chair: Name of the committee chairperson
Full Professor, FCT-NOVA

Rapporteur: Name of a rapporteur
Associate Professor, Another University

Members: Another member of the committee
Full Professor, Another University

Yet another member of the committee
Assistant Professor, Another University

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2025

QuicSand: A Deep study on the Performance of Different QUIC Implementations

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.10) [31].

Dedicatory lorem ipsum.

Acknowledgements

I want to start by expressing my sincere gratitude to my advisor, Prof. João Leitão, for giving me all the support, patience and guidance for this work. Without his help and insights, this work would not have been possible to be complete. I would also feel honoured to thank him for being a remarkable teacher and mentor throughout my academic journey.

I have to praise my deepest gratitude to my family, my parents and friends for being my support, encouragement and motivation throughout this difficult and hard-working journey. I would also like to highlight all the support that I received form all colleagues that helped me, directly or indirectly, to guide me and gave all their different expertise that was crucial for the completion of this work.

Finally, I would also like to acknowledge the often unseen yet invaluable support from NOVA University of Lisbon, which provided me with every resource I needed to complete this work.

”

“You cannot teach a man anything; you can only help him discover it in himself.”

— Galileo, Somewhere in a book or speach
(Astronomer, physicist and engineer)

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 evolution 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 address this we have developed QuicSand, a novel dedicated tool that extensively test QUIC implementations, simplifying the testing process, helping users of systematically evaluate the performance of different implementations, using multiple different workloads and network conditions matching the needs and operation conditions of applications being developed by the user.

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, desenvolvemos o QuicSand, uma ferramenta inovadora e dedicada para avaliar e comparar extensivamente várias implementações do protocolo QUIC, simplificando o processo de teste, ajudando os utilizadores a serem capazes de avaliar rapidamente e de forma sistemática o desempenho das diferentes implementações, usando diversas cargas the trabalho e condições de rede correspondendo às necessidades e condições operacionais das applicações desenvolidas pelo utilizador.

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

Contents

List of Figures

List of Tables

Listings

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 a few 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). QUIC is an application-level protocol, which was mostly activated to allow for a speedy evolution, development, and innovation. This as led to a somewhat wide proliferation of new implementations and variants of QUIC that bring with them new features and new approaches to using and managing network and applicational resources.

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

To address this need, this work presents QuicSand, a network testing tool that uses a network emulation tool to simulate real-world network conditions to test and analyze the performance of different QUIC implementations. This leads to a deeper understanding of the behaviour of the implementations under different network conditions to help developers choose the best implementation for their applications.

This study not only to provides a new testing tool that automates the evaluation of the implementations but also to analyze their strenghts and weaknesses. Our results aimn at contributing for a more systematic and automated way to evaluate this increasingly popular transport protocol.

1.1 Objective

A key limitation of the current (so-called TCP/IP) network stack is the difficulty of innovating and deploying upgrades 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 avoids the challenges related with its deployment at the network stack within the kernel.

While a few works [20, 26, 34, 37, 39] have already compared the performance of different implementations and variants in specific 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 dissertation, we pushed the state of the art by developing a methodology and a tool that systematically tests QUIC implementations across different operational conditions, focusing on designing and implementing a containerized network testing tool that uses a network emulator to set a controlled environment for measuring performance metrics such as, throughput, round-trip time, cpu usage, among others.

Additionally, this work provides a methodology to automate the execution, logging and visualisation of test results, enabling large-scale scenario evaluation with minimal manual intervention. This research aims to create a generalized C API for QUIC implementations, facilitating the integration of new protocol implementations within the testing framework. This API is designed to be extensible and developer-friendly, enabling researchers and practitioners to incorporate and evaluate emerging QUIC implementations with minimal effort.

1.2 Contributions

The main contributions of this dissertation are the following:

1. **Development of QuicSand:** A containerized testing framework that enables repeatable and large-scale QUIC performance evaluations under diverse network conditions. It includes a general API to offer a common interface for different QUIC implementations, a set of client-server applications to test various workloads over QUIC, and a network emulator to simulate different network conditions.

2. **Network Scenarios and Workloads:** A set of network conditions and workloads that allow systematically evaluate of the performance of the different implementations incorporated in QuicSand in an automated way.

3. **Performance Evaluation and Results:** An analysis of results and a comparison of evaluated performance for different existing QUIC implementations.

1.3 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, identifying performance metrics that are relevant to this work;
- Chapter 3 presents the design and implementation of the QUICSand tool, starting with an overview of the tool, where the overall architecture and design is covered; then, it covers in more details the tool components and the implementation details; finishing with a brief overview over the main orchrestator component and workflow of QuicSand tool;
- Chapter 4 presents the methodology used to evaluate the performance of the different QUIC implementations, starting by the network conditions and workloads used, then it covers the methodology used to evaluate the performance of the different QUIC implementations, and finally the chapter ends with the results obtained from our evaluation of two different implementations (in C) that we have integrated into our tool;
- Chapter 5 is the chapter that concludes this work, where the main conclusions are presented, and some directions for future work are suggested.

2

Related Work

This chapter introduces the concepts and technologies that are relevant to this work. The chapter is structured as follows: in Section 2.1 we provide a brief overview of the Internet stack model, followed by a structured and precise explaination of the transport layer protocols and some general concepts about the session and application layer protocols; in Section 2.2 we describe in dept the QUIC protocol exposing its own characteristics and giving a overview of the different implementations that already exist; in Section 2.3 we gave a general insights about some studies that were already developed over the QUIC protocol performance; in Section 2.4 we explore and present some existing network tools; finally in Section 2.5 we discuss general ideas, concepts, and tools that have contributed to the development of this work.

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 [28] is a reference to how messages navigate between two points in the network. The model refers to a combination of 7 layers, each having a different role, as illustrated 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) [29]:

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

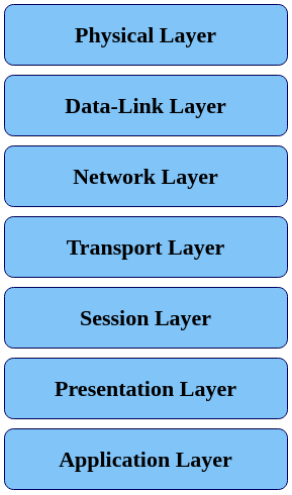


Figure 2.1: OSI Model

reliable delivery, depending on the protocol that manages the data delivery. Some examples of link-layer protocols are Ethernet, Wi-Fi, DOCSIS and PPP [9].

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 the data into packets. The layer also counts on a protocol named ARP [2] for the address resolution to know the media access control (MAC) [36] of the source and destination, transferring the data to the data-link layer. The network layer includes the Internet Protocol (IP) [23] 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 [44] and UDP [45], 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 communicating 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 [18] and TLS [12].

Application Layer - Layer 7 The application layer resides at the top of the OSI model and includes protocols such as HTTP [35], FTP [17], and DNS [13, 14]. 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) [29], 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 Handshake. 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. It 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).

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

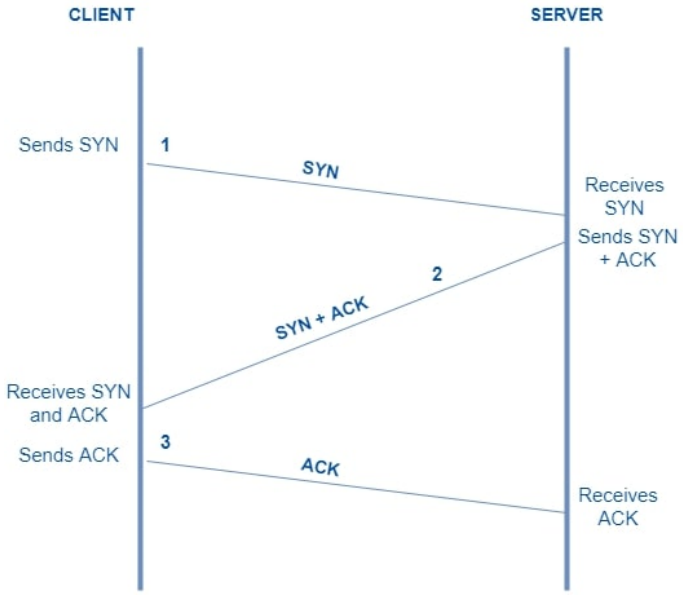


Figure 2.2: TCP Connection Establishment (Taken from [40])

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 is 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) is 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.

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 [44] 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;

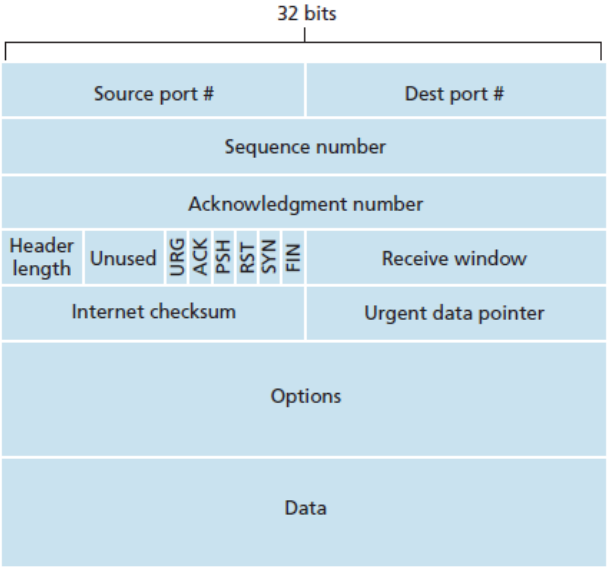


Figure 2.3: TCP Segment Structure (Taken from [29])

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 expected sequence number, the ACK is ignored.

The TCP also doubles the timeout interval after each timeout event to avoid premature timeouts. The value of *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 the 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, which is mentioned as the four-way handshake [41]. 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 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. It 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 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.

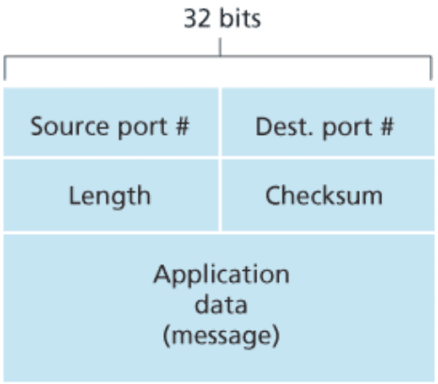


Figure 2.4: UDP Segment Structure (Taken from [29])

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.

2.1.2 Session Layer Protocols

In this section, we discuss a particularly relevant session layer protocol: the Transport Layer Security (TLS) protocol [12]. 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 [18], 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 [16] 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 [11] handshake is divided into several steps, as can be seen in Figure 2.5.

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

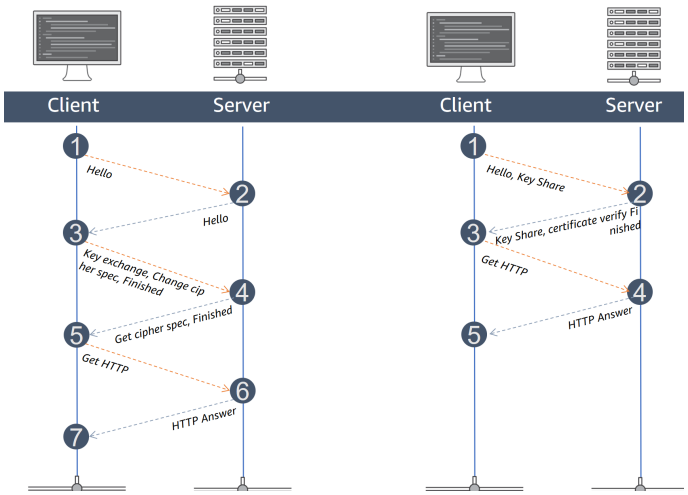


Figure 2.5: TLS 1.2 2-RTT and TLS 1.3 1-RTT Handshakes (Taken from [47])

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 [17] which supplies the transfer of files between two hosts, SMTP [27] for the transfer of e-mail messages, and SPDY, among others. This subsection has presented two protocols, the HTTP protocol [35] that is one of the most used protocols on the Internet and uses the QUIC protocol

in its latest version, and the SPDY protocol [5] 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) [6] 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) [30].

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 [8] 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 [8].

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 [42]. It will be complemented with the first version of the QUIC version presented in RFC9000 [24].

2.2.1 QUIC Version-Independent Properties

QUIC as defined in the RFC8999 [42] "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.

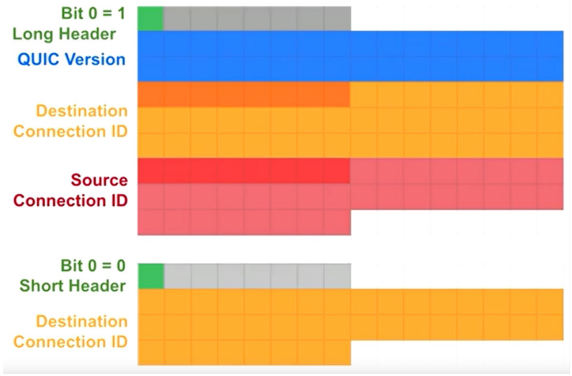


Figure 2.6: Long and Short Packets Header (Taken from [38])

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 [24]. 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 flow controlled, 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 amount 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 of starting to send 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 [21]: 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.

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, which 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

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

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]

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 discuss a set of previous performance analyses conducted on the QUIC protocol, drawing comparisons with both TCP and various implementations of the

[7] https://github.com/microsoft/msquic
[8] https://github.com/facebook/mvfst
[9] https://github.com/mozilla/neqo
[10] https://github.com/ngtcp2/ngtcp2
[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

protocol. The aim is to provide an 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.

This discussion also allows us to identify 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 different implementations.

2.3.1 QUIC vs TCP

According to some studies [26, 1], QUIC outperforms TCP in scenarios with 0-RTT connection establishment, recovering from packet loss, and also with scenarios where there are 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. In the study [26], it was observed that QUIC 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 the frequency of increasements than the first one, explaining 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 [37]. 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 study performed in 2021 [20]. 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 QUIC Performance Evaluation on High-Rate Links

In this study [25], the performance of QUIC was evaluated in high-rate links, ran in real hardware. They analyzed multiple QUIC implementations but their main focus was on LSQUIC and quiche implementations. The performance analysis was focused on the variation of the parameters inside the QUIC implementations, searching for the best results over that network scenario and the best performance in terms of goodput. The study concluded that an already optimized TCP is limiting QUIC for his development.

Although this study focused on testing multiple implementations, the tests were limited to single client-server connections, and the framework used was highly dependable on the simple client and server implementations that the QUIC implementations provided.

2.3.4 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 consider on our own work.

2.3.4.1 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, some of the nodal delays that a packet can be exposed to [29].

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 be 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 separation 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.

2.3.4.2 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 [49]. 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.

2.3.4.3 Average Throughput

Network throughput is the average volume of data that can travel in the network for a given file transfer for instance [50]. 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.

2.3.4.4 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 [51].

2.3.4.5 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 [4].

2.3.4.6 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 [43]. 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 [43].

2.3.4.7 Channel Utilization

The study [8] 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.

2.3.5 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 [34] 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 [7].

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* [39] experimental 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.

2.3.6 Discussion

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

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 develop QuicSand which gathers existing and novel features to guarantee a comprehensive and robust performance evaluation of the various QUIC implementations.

2.4 Network Emulation Tools

For a deep study of the performance of a network protocol and to compare the difference between implementations of QUIC protocol we depend on many network topologies and conditions to address the real-world scenarios and to have a reliable comparison among

the different implementations. To superscribe this aspect, it is necessary to evaluate a protocol implementation in a 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 [46] 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 [48] 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 [19]. 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 *Go Network Emulator*

Go Network Emulator (*GONE*) [3] is a scalable, container-based network emulation tool that allows users to create and deploy their custom network topologies and simulate different network conditions. GONE runs on a Docker-based environment and offers an accurate model of network operation by explicitly creating each node instead of algorithmic link aggregation or simplification. Additionally, the emulator can create topologies where the applications share network links.

Moreover, *GONE* contains an interactive command-line interface (*GONE-CLI*), which gives users real-time access to network parameters. Through this interface, users can dynamically introduce network constraints such as delay, packet loss, and bandwidth at the link level. It also allows users to create endpoints and connect network elements like routers, nodes, and bridges.

2.4.5 Discussion

In this section, the presented network emulation tools offer an extensive overview of some of the tools available for measuring the performance of QUIC implementations. Most tools range from centralised to decentralised approaches and from user-space to container-based emulation.

The choice of the network emulation tool was driven by the specific advantages that make this study scalable and flexible, wherein custom network structures can be built. Since only *Kollaps* and *GONE* are compatible with decentralized emulation, the decision was finally determined by elements needed to build the network scenarios necessary for the performance evaluation of QUIC implementations performance.

There was a consideration for *Kollaps* at first, but its network emulation approach is built on end-to-end compilers rather than explicit compilation of network constraints, which simulates each of the network components. It also does not permit the creation of scenarios in which traffic passes through a shared but unmanaged bottleneck. In contrast, by allowing users to specify precise traffic bottlenecks, *GONE* makes it a more appropriate option for the present research.

All tools come with trade-offs, and *GONE* has its downsides, too. Notably, the 1 Gbps is the maximum shared bandwidth between nodes. However, this constraint did not affect the aims of the performance evaluation, as the main emphasis is to create a congestion point in the network and evaluate the performance of QUIC under these conditions. A second limitation can be the latency that *GONE* creates when running in multiple machines, which can affect results. This limitation does not affect our work since the topologies were not complex, and the testing environment consisted of only one machine.

2.5 Summary

This chapter introduced key concepts related to this study, starting with an overview of network protocols. It covered transport layer protocols such as TCP and UDP as well as Layer protocols such as TLS and SPDY. SPDY was particularly crucial because it made an impact on QUIC itself.

Then, the chapter discussed QUIC protocol in detail, its main features, packet structure, connection setup, stream management, flow control, congestion control, and connection migration. It also examined various QUIC implementations and prior studies comparing QUIC and TCP and evaluating different QUIC libraries concerning different network conditions. Key performance metrics such as efficiency, productivity, human resource effectiveness and system performance metrics should also be widely discussed, such as delay, latency, throughput, packet loss, jitter, time to first byte, etc.

To guarantee realistic evaluation, the chapter also reviewed the tools of network emulation, comparing their features and limitations. *GONE* was selected because it had to implement network topologies with regulated bottlenecks, thus making it well-equipped for this study.

The next chapter will describe the QuicSand Tool in more detail, including its design, implementation, and usage in the QUIC performance evaluation.

3

QuicSand Tool

In this chapter we present our proposed solution whose objective is to simplify the evaluation of QUIC implementations that we named QuicSand [15]. This tool is achieved by combining multiple services and processes that together orchestrate serveral executions of QUIC implementations (precisily integrated in our tool) to provide results over the performance of the protocol implementations integration into the QuicSand API.

In the chapter, on a Section 3.1, we start with a general overview and detailed information on how the tool works, followed by the insights over each component, starting in the Section 3.2 by presenting the Quicsand C API explaining its general purpose and how can it be leverage to address new variants of QUIC implementations; in Section 3.3 we present the client-server applications that are used to test the performance of the protocol simulating real-life application workloads; in Section 3.4 we present the integration of the tool with the *Go Network Emulator*, explaining how the tool can be used to test the protocol in different network scenarios; finally the Section 3.5 explains the orchestration service that is used to manage the different services and processes that are part of the tool, finshing with a summary of the chapter in the Section 3.6.

3.1 Overview

The QuicSand tool is designed to simplify the evaluation process of the different QUIC implementations. To achieve that purpose multiple services processes were combined to orchastrate this process in a stable and reliable way.

As discussed in Section 1.1, the main goal of the tool is to provide a complete performance testing environment with different workloads, network scenarios, and considering several relevant metrics to evaluate the performance of a protocol. To address this goal, the tool can be divided by the following components: the QuicSand API, the QuicSand client-server applications, the integration with the *Go Network Emulator*, and the orchestration service. Every single component is essential to the tool, and they are all interconnected to provide the user with an adequate evaluation environment.

The tool architecture, illustrated in Figure 3.1, begins by running a script, that can

be called the orchestrator, which is responsible for managing the test environment and receiving all application-related files. The orchestrator first removes any existing Docker containers and network configurations that may have been created during previous runs. This ensures a clean environment for each test execution.

The orchestrator analyzes the *tests.yaml* after cleaning up the environment. The file includes the tests to be executed over a QUIC implementation. Based on these details, the orchestrator builds the implementation image from an Ubuntu-based system, as QuicSand was built for this OS. Once the test image is built, the orchestrator constructs the test environment.

A new Docker image is built for every client-server application according to the specific test being executed. Test arguments and workload configurations are passed to ensure the Docker image runs with the required parameters.

The orchestrator at this point reads the topologies file, which describes the network scenarios to be tested. To generate and configure these scenarios, the Topology Generator Service is used. It runs a script that takes the topology specifications from the test configuration and generates a shell script file, which is used to define the network scenario using *GONE* commands that are generated based ont this specification.

Once the test environment is fully configured, the experiment begins. The orchestrator executes the tests according to the defined workloads and network scenarios. Each result is extracted and stored in the client-server application in a file for analysis after the conclusion of the experience. This structure approach helps to ensure every test runs in a controlled and unique context, enabling a reliable evaluation of performance and reproducibility of results. The tests continue until all of the scenarios have been executed using the process described above.

3.2 Quicsand API

In this section, we dive over the QuicSand API, a general C API for QUIC implementations that will allow the integration of new variants for being tested with QuicSand. The API is composed of a set of structures and functions that allow the user to create their one client-server applications and more imporatntly to use QuicSand own client-server applications tool uses to perform the various tests, in particular to generate different types of interactions and workloads that are, therefore, independent of the QUIC implementation being tested.

3.2.1 Structures and Context Functions

The API is composed of a set of structures that represent the objects that the implementations need to have full control over the application. Since API is modeled to be used with many different implementations, the structures given by the API are generic pointers, as can be seen in the code on Listing 3.1, since they need to be allocated inside their API to address the different structure sizes and parameters that the implementations might have.

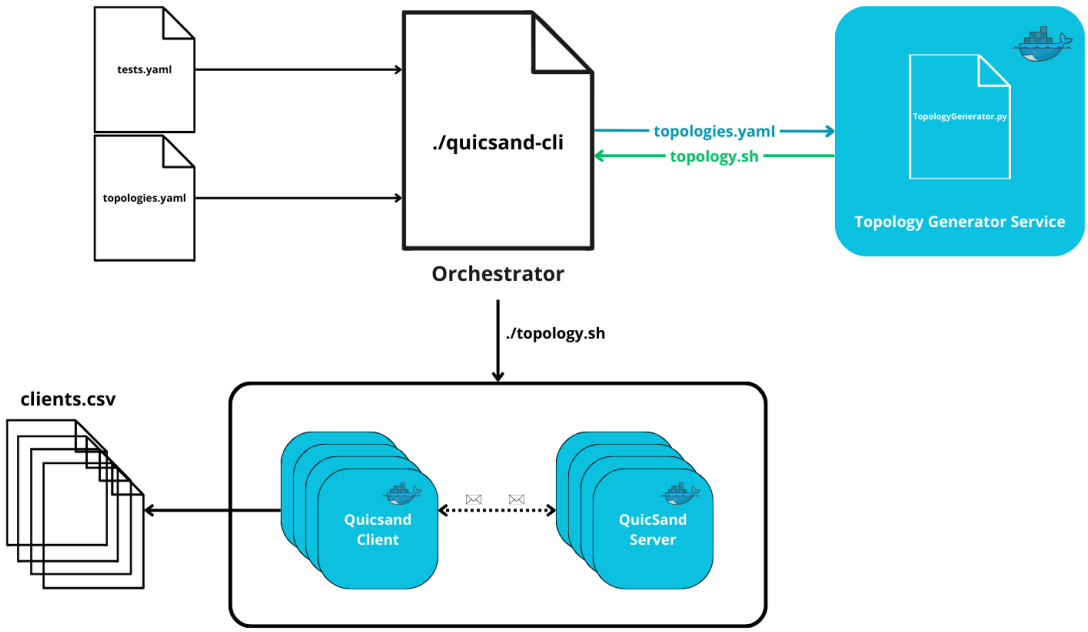


Figure 3.1: QuicSand Architecture

```
1    typedef void *context_t;
2    typedef void *connection_t;
```

Listing 3.1: Structures

The API also provides functions to create and destroy the different implementations' contexts. The context is the main structure that holds the information about the implementation, the certificates, the keys, the set of connections and streams that are open, among other things. The main point is to provide the user a way to create and destroy the context that is being used by a concrete QUIC the implementation, as can be seen in the code on Listing 3.2.

```
1    context_t create_quic_context(char *cert_path, char *key_path);
2
3    void destroy_quic_context(context_t context);
```

Listing 3.2: Context functions

3.2.2 Connection and Stream Functions

The connection and stream functions are in part similar, since they are both used to open, for the client side, and accepted in the server side. The idea with the availability of connection functions is to provide the structure that will be used to open streams and close connections. The connection functions are shown in the code on Listing 3.3 and should contain information about active connections and streams currently in use by the

application being executed.

```
1    // client
2    connection_t open_connection(context_t context, char* ip, int port);
3
4    //server
5    connection_t accept_connection(context_t context);
6
7    int close_connection(context_t context, connection_t connection);
```

Listing 3.3: Connection functions

Although stream functions are also used to execute operations, open and accept similarly to connections, they are slightly more complex. These functions return an integer value representing a file descriptor, as shown in the code on Listing 3.4. They are specifically designed to open Unix sockets, facilitating the communication between the client-server applications and the QuicSand API implementation.

This design choice simplifies data transmission and stream management for different implementations. By using Unix sockets, the client-server application can communicate with the QuicSand API to send, receive, and close streams in simple and implementation-independent way.

```
1    //client
2    int open_stream(context_t context, connection_t connection);
3
4    //server
5    int accept_stream(context_t context, connection_t connection);
```

Listing 3.4: Stream Functions

3.2.3 Server Functions

The functions shown in the code on Listing 3.5 are responsible for binding the server to a specific address and port and setting the server in listening mode. Since every implementation requires a mechanism to bind to a designated address and port to receive incoming connections, this function is essential to allow test applications of QuicSand to be agnostic to the QUIC implementation being tested. Additionally, the *set_listen* function switches the server into listening mode, thus enabling it to receive client connections. These are the key functions that ensure servers handle incoming connections.

```
1    int bind_addr(context_t context, char* ip, int port);
2
3    int set_listen(context_t context);
```

Listing 3.5: Server functions

3.2.4 Connection Statistics

The final function and structure provided by the API are related with the connection statistics. This structure is intended to return connection statistics to the application, as is kept simple to make sure it is easy to integrate across different implementations. If a specific implementation cannot produce statistics for some particular set of metrics, the corresponding values should be set to -1.

The code presented in the code in Listing 3.6, includes both the structure and function, which are the essessial comparatators for QuicSand. It uses statistics to present performance results of the protocol that are critical for the acessement of implementation performance.

```
1   typedef struct statistics
2   {
3       ssize_t min_rtt;
4       ssize_t max_rtt;
5       ssize_t avg_rtt;
6       ssize_t total_sent_packets;
7       ssize_t total_received_packets;
8       ssize_t total_lost_packets;
9       ssize_t total_retransmitted_packets;
10      ssize_t total_sent_bytes;
11      ssize_t total_received_bytes;
12  } statistics_t;
13
14  int get_connection_statistics(context_t context, connection_t connection,
        statistics_t *stats);
```

Listing 3.6: Connection Statistics

3.2.5 Integration of Implementations

Integrating multiple QUIC implementations into the same framework is challenging and requires careful design to adjust the differences between the different implementations' design models while maintaining a consistent interface. In our tool, we have integrated MsQuic [32] and Quiche [10] QUIC implementations, using our general QuicSand API that abstracts the differences between both implementations to expose a single and simple API for applications.

To address this issue, we first start to align both implementations into the same design model. MsQuic is an event-driven implementation that relies on callback functions to process QUIC events, such as incoming data, connection and stream state changes, among a few others. On the other hand, Quiche uses an API, including functions that create packets, read packets, set configurations and others, but without leading with the sockets' incoming and outgoing packets, the state of the connections and others that make these two implementations very different. To deal with this, we utilized *libev*, a high-performance event loop library in Quiche that will mitigate the differences between both implementations, making Quiche use a similar event-driven model.

To facilitate the interaction between the event-driven implementations' threads and the application threads that utilize our API functions, we used the POSIX threads library (pthread). This ensures that the QUIC event threads can effectively communicate with the main application logic while avoiding concurrency issues. Synchronization is achieved using locks, unlocks, and queues, for example, accepting connections and accepting streams, where the QUIC callbacks implementations receive the connections and streams, creating their structure that is saved in the queue structure and not immediately returned. This approach guarantees that no connections or streams are lost, ensuring their processing order. These structures are therefore enqueued by the accept functions of the server, ensuring synchronization between threads.

To enable switching between different QUIC implementations at compile time, we incorporate CMake build definitions along with preprocessor macros `#ifdef`. This approach allows us to use a single file to support multiple implementations without code duplication. During compilation, a CMake flag determines which QUIC implementation will be included, and the corresponding sections of code are activated using conditional compilation. For example, a macro such as MSQUIC or QUICHE indicates which implementation logic is compiled. This strategy provides flexibility, as the same application code can be compiled for different QUIC implementations without requiring extensive modifications.

By designing our integration with this approach, we ensure that both MsQuic and Quiche can be used within the same testing framework, allowing for comparative evaluations under identical network conditions. This QuicSand API abstraction, event-driven adaptation and selecting implementations at compile time ensures that our tool can incorporate in the future more implementations with minimal modification, providing extensibility for our tool.

3.3 QuicSand Client-Server Applications

Client-server applications are a key component in the system, acting as the main mechanism to exercise different communication patterns and workloads over different QUIC imlementations taking advantage of the QuicSand APU and the test scenarios. These applications fall into four types: request-response, upload, download and streaming. These categories represent the most common interaction patterns on Web Applications, which are the main use cases for QUIC.

Due to the need to reset the clients, servers, and network configurations each time a new workload and network scenario are tested, we decided against implementing a single, unified client-server application supporting all types of communication. Instead, we opted for separate, specialized applications for each type. This approach reduces complexity, making the applications more lightweight and efficient, while also minimizing erros and bugs.

After every client finalizes the test the connections statisctics are collected and stored into a file for further analysis. The client finishes by closing the connection and the destroying the context in every application type.

3.3.1 Request-Response

The request-response application is the least complex type out of all four types because all it needs is a client sending a request and the server responding.

Each request from the client is received by the server via a separate stream. This application is particularly useful for measuring the round-trip time (RTT) at both the protocol and application levels. Since the amount of data exchanged remains relatively balanced, it serves as a reliable method to analyze latency and performance. Tests that use this application can execute for a user-defined duration. The workflow of the request-response application is illustrated in Fig. 3.2.

Additionally, the average application-level round-trip time is computed by recording the timestamp before sending the request and immediately after receiving the response. These values are stored and processed to calculate the mean RTT. This round-trip time includes all protocol and application-level overheads, but since every implementation follows the same logic, the results remain comparable across different QUIC implementations.

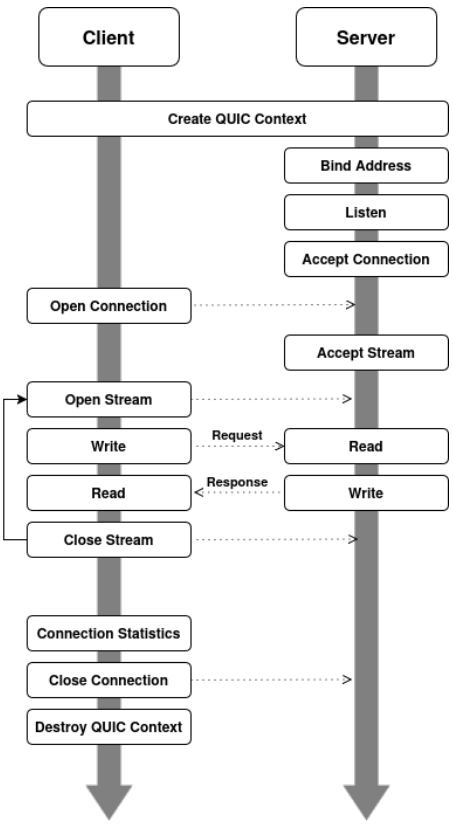


Figure 3.2: Request-Response application workflow

3.3.2 Upload

The upload application is different from the request-response model since, on average, the client transmits much more data than it receives. The process begins with the client opening a particular file specified in the application arguments. The client then calculates the file size and sends this information to the server before the file data starts to be transferred, as illustrated in Fig. 3.3.

When the server receives the file size, it prepares to receive the data stream. Once the transfer has completed, the client waits to receive an acknowledgment from the server indicating the file has been received successfully. The session ends with the client shutting the stream, fetching performance metrics, and closing the connection.

This application is meant to measure throughput from the client to the server, getting insights into how the underlying transport protocol transfers the data under different network scenarios.

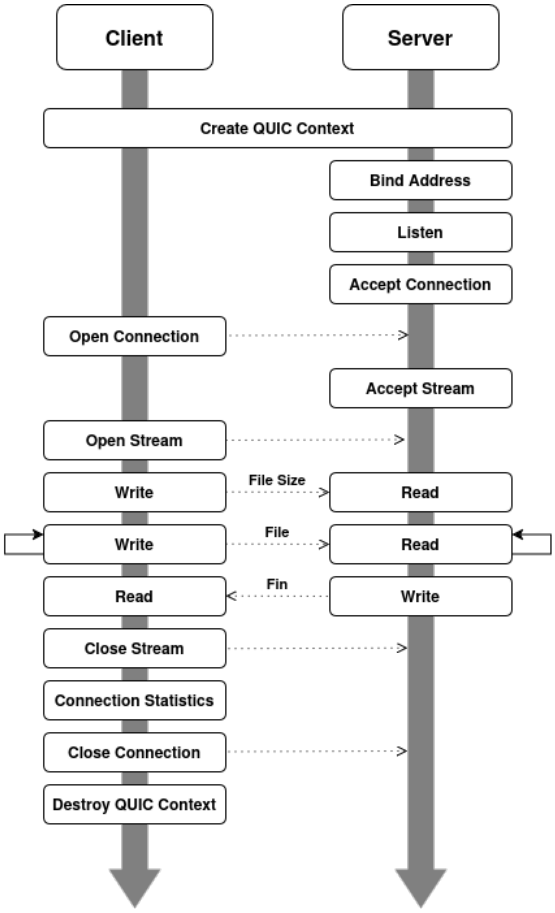


Figure 3.3: Upload application workflow

3.3.3 Download

The download application follows the inverse logic of the upload application. The client begins by sending a request for a specific file from the server. The server processes the

request and, before initiating the transfer, sends the file size to the client. The client then proceeds with the download, as illustrated in Fig. 3.4.

Once the file transfer is complete, the client terminates the session by closing the stream and performing the same post-processing operations as in the upload scenario.

In this case, throughput is measured in the opposite direction (server to client). Since multiple clients can request downloads simultaneously, this application helps analyze how the server-side implementation of the protocol manages concurrent data transfers.

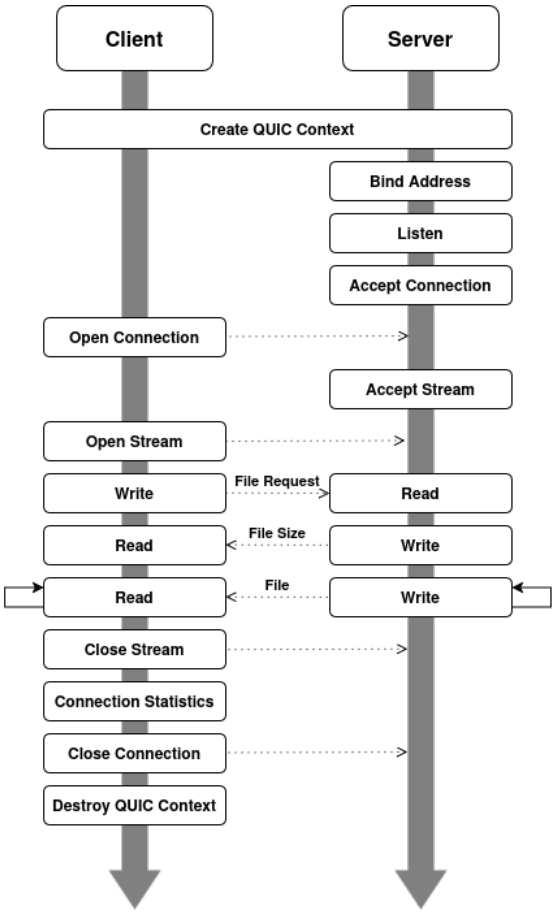


Figure 3.4: Download application workflow

3.3.4 Streaming

The streaming application is different from the download application. When you download, you get prior information about the size of the file(and hence the amount of data to receive); with streaming, you data being sent to the form the server to the client over a controlled rate. The difference between the two lies in how the data flow is regulated: rather than transmitting data as fast as possible, transmission is controlled by the application configured preset bitrate.

The client initiates by sending a message to the server, which indicates the required bitrate received as a client application argument. Then, the server starts streaming data at

the specified rate, as illustrated in Fig 3.5.

When the test time expires, the server closes the stream, which tells the client that no further data will be sent. The client closes the stream, and the session is terminated after gathering client's connection statistics metrics to further evaluate the performance of the implementation.

This application simulates real-world streaming workloads in which data have to be sent up at a constant rate to guarantee playback smoothness, bandwidth efficiency, and protocol-level flow control.

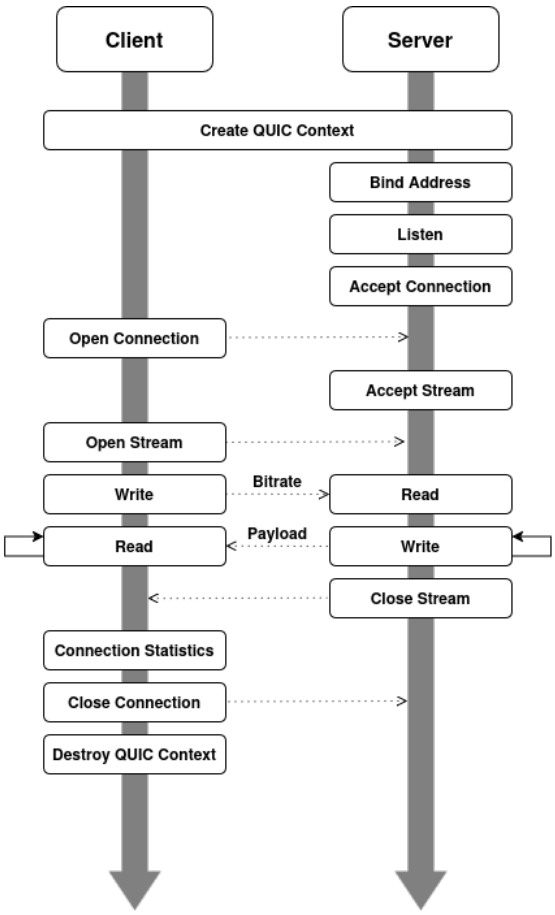


Figure 3.5: Streaming application workflow

3.4 Integration with *Go Network Emulator*

In this Section, we explain how the QuicSand tool integrates *GONE* as a network emulator to provide a deep analysis of the implementations over multiple network scenarios.

3.4.1 Configuration Topologies File

The configuration topologies file is also a YAML file that represents the network scenarios to be tested. The topologies are represented with a name and relevant constraints, such

as latency, bandwidth, number of clients and servers, and the docker image that is going to be used to run the tests created on the orchestration script.

An example of the configuration topologies file is shown in the code in Annex I on Listing I.1. The file starts with the sequence of topologies. Each topology is defined by the name and a sequence of key-value pairs that define the experience, that should contain: the latency (in ms), the bandwidth (in Mbps value), the image (the name of the docker image that the topology will execute), the number_of_clients and the number_of_servers that the topology will contain. This topology example represents a one-to-one communication, using the docker image with the name *quicsand*, which applies link-level constraints of 50 ms of latency and 500 Mbps of bandwidth.

The YAML configuration syntax used is both simple and structured, that provides readability and ease of use. This structured approach is used by the Topology Generator Service to create the experimental setup, explained in the next subsection.

3.4.2 Topology Generator Service

Although this service is not part of *GONE*, it is directly related to the network emulator. *GONE* provides several features that are beyond the scope of this work due to its complexity, such as the ability to create more complex networks, apply jitter, and packet loss. These scenarios, when combined with the selected test cases, would generate a massive number of tests, making it difficult to manage and analyze all of those independent experiments within the timeframe of this work.

So, to further get a better and more efficient way to create the selected scenarios to test the implementation that are capable of doing a deep analysis, the Topology Generator Service was created. This service consists of a Python script inside a lightweight Docker container that reads the topologies file and generates the shell script that is used to define the network scenario following the *GONE-CLI* commands syntax and to set the docker containers that run the QuicSand client-server applications with the test named to execute in that scenario.

The following code snippet in Annex I on Listing I.3 shows an example of a shell script generated by the Topology Generator Service. This script starts with client and server creation commands where the Docker containers are created, then bridges are created for both client and server as they are the components that link clients and server to other network components. Following it comes the creation of the two routers and the connection between clients to routers and server to routers. This step needs two commands: one that connects the client to the client bridge created before and the second that connects the client bridge with one of the routers, and the same step is equally applied to the server and the other router. Finally, the command that connects routers with the network constraints specified for the executing topology, followed by the command that propagates the routing rules between the routers and finishes with the unpause commands that, as the name indicates, unpause the Docker containers to start the experiment.

3.4.3 Network Orchestration

Network orchestration is now managed by the *GONE* network emulator, which generates network scenarios as defined by the Topology Generator Service. The emulator, initialized during the environment setup, executes commands specified in the shell script. These commands establish all endpoints, network components, and their interconnections.

To ensure that clients connect to their respective servers only after the setup is complete, *GONE* pauses all containers upon creation. The tool then sequentially resumes execution, first unpausing the servers, followed by the clients—marking the official start of the experiment. Once all clients have completed their tasks, performance metrics are collected, and the network emulator is restarted to prepare for the next scenario.

3.5 Orchestration Service

The orchestration service relies on a script that contains every service and process that the tool uses to perform the tests. Its main goal is to manage the test environment by controlling all the services and guaranteeing a safe and clean environment for each test execution. It also guarantees that the services are correctly deployed and used in a synchronized way. At the end of a test, the metrics are collected and stored into a directory that owns the specification of the testing scenario and workload.

The script receives two file paths as arguments the configuration of the tests and the configuration of the topologies. This configuration files are used to define the workloads and the network scenarios that the orchestration service might use to perform the tests.

3.5.1 Configuration Tests File

The configuration tests file is a YAML file that contains the details about the tests to be executed and the QUIC implementation being tested. The file is divided into two main sections: the configuration itself, where it is defined which implementation is going to be tested, and the testing workloads, where the user can define the workloads that are going to be used in the tests. An example of the configuration test file is shown in the code in Annex I on Listing I.2.

A test is defined by assigning a name to it within the tests section. Each test is categorized based on the type of network operation it performs, which may include request-response, upload, download, or streaming.

In the request-response test, the test duration is specified in seconds. The data size is defined in bytes and is scaled by a predefined factor on the server side. This means that if the data size is set to 10 bytes and the factor is 100, the server will generate a response of 1000 bytes.

To designate a source of data for upload and download tests, file paths are given. In an upload test, the file path indicates where the file is stored on the client side. On the other hand, the file for the download test specifies the location of the file on the server side,

whereby the client begins by sending a request to the server to serve a file that corresponds to that path. Regardless of the direction, the file path is transmitted by the client.

The streaming test is defined similarly to the request-response test in terms of duration, which is specified in seconds. Additionally, the bitrate is defined in kilobits per second (Kbps), allowing precise control over the data transmission rate.

3.6 Summary

In this chapter, we presented the QuicSand tool, a solution designed to simplify the evaluation of QUIC implementations. The tool is composed of multiple services and processes that work together to provide a complete performance testing environment. The QuicSand API is a C API that allows users to create their client-server applications and use QuicSand's applications to test the performance of the protocol. The integration with the *Go Network Emulator* allows the testing of the protocol under a variety of network conditions, and the orchestration service controls the various services and processes involved in the tool. The tool is designed to be flexible and easy to use, evaluating the various conditions in which QUIC implementations perform.

In the following chapter, we show our test methodology and our experimental results. We also discuss the implications of the results and give insights for the protocol under various workloads and network scenarios.

4

Methodology and Results

4.1 Methodology

This section details the methodology with which we evaluate the performance of QUIC under different network conditions and workloads using QuicSand. The experimental setup, network scenarios, and performance metrics are discussed to give a complete idea of the evaluation process. This defines the bounds and conditions by which QUIC can be tested. This study looks to provide more in-depth insights into the protocol's behavior and performance characteristics, as well as to validate the usability functionality of our tool.

4.1.1 Experimental Setup

All experiments were performed in one machine in our university cluster to guarantee accurate results. The system is equipped with two AMD EPYC 7343 precessors with a total of 32 cores. The node has 128 GiB of DDR4 3200MHz memory, 450 GB SSD, and two network interfaces with 10 Gbps each.

The machine runs a Debian 12 (Bookwork) and guarantees stability and compatibility to use the QuicSand tool requirements (Docker, Linux kernel). The machines run in a controlled and isolated environment with the necessary computational resources to ensure accurate results.

4.1.2 Network Scenarios

To evaluate the performance of the QUIC protocol, a series of controlled network scenarios were created to simulate a range of real-world environments. These scenarios aim to determine how QUIC implementations perform in different network topologies and variable network conditions, for example, latency and bandwidth restrictions.

Network Topologies. We looked at two patterns of topologies where clients vary from a single instance to a large-scale setup with up to 100 concurrent clients. This range allows us to examine QUIC implementation performance over various scales, from single communications to such high-concurrency environments where the protocol's efficiency

of handling multiple simultaneous connections can be evaluated. The scalability of the server side was tested in two configurations:

- A **single-server deployment**, where multiple clients communicate with a single server.
- A **multi-server deployment**, where the number of servers matches the number of clients, results in one-to-one connections.

In the *n*-*n* topology (Fig. 4.1), n clients, where $n \in \{10, 25, 50, 100\}$, create a dedicated connection to n servers, forming a one-to-one mapping from clients and servers. This particular topology is focused on testing the performance of various QUIC implementations across diverse competing flows. By simulating simultaneous data movement on a common network infrastructure, we can assess the behavior of each implementation when dealing with congestion, as bandwidth is divided between multiple flows sharing a bottleneck point.

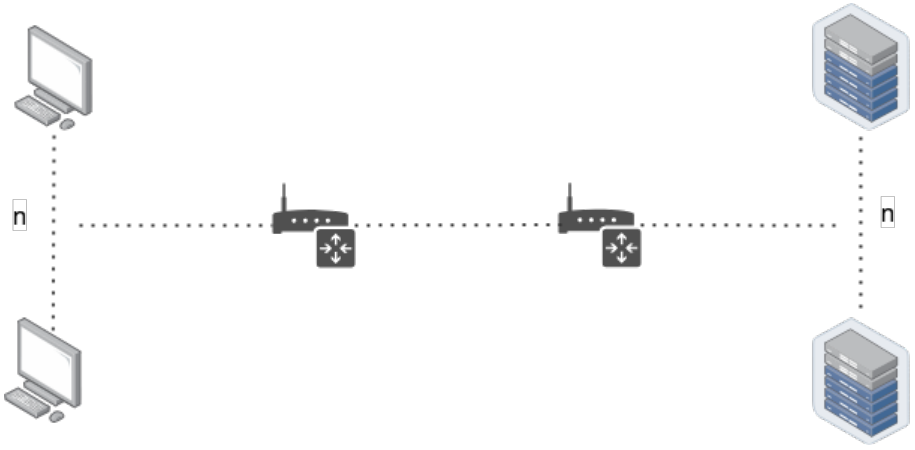


Figure 4.1: Network scenario where multiple clients connect to multiple servers.

In contrast, the n-1 topology (Figure 4.2), where n clients are connecting to a single server. This setup not only adds several competing flows but also highlights the server-side concurrency effects when managing concurrent connections. Evaluating QUIC implementations in this environment enables us to understand their capacity to successfully multiplex streams, control congestion, and aggregate resource contention at the server level. This topology is specifically important and relevant to real-life scenarios where one server has to serve multiple clients at the same time.

Within these network topologies, we did not carry out an explicit scenario in which one client connects to one server without any competing flows. This architecture removes contention for server-side resources and bandwidth and gives a baseline measurement of the protocol's raw performance. By analyzing this scenario, we can evaluate if a QUIC implementation meets the expected performance within an isolated environment, avoiding any interference caused by other network flows.

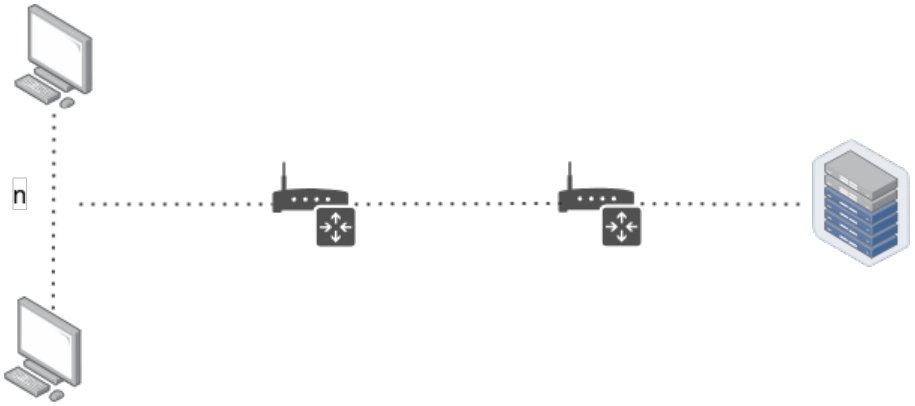


Figure 4.2: Network scenario where multiple clients connect to a single server.

Network Contraints. For network performance analysis, we considered that latency plays a crucial role in protocol efficiency, particularly in transport-layer protocols optimized for low-latency communication. The selected latency values of 50 ms (low), 200 ms (medium), and 400 ms (high) correspond to typical real-world scenarios, such as local area networks, intercontinental connections, and highly congested or long-distance links. These values represent the total round-trip time (RTT) of the link, as *GONE* uses these to divide the values representing the total round-trip time (RTT) of the link values by two. Hence, the one-way latency from the client to the server is 25, 100 and 200 ms, respectively. These numbers were selected based on research on common network latency measurements [33], further described by Stefano Gridelli.

Similarly, bandwidth constraints affect how efficiently QUIC can utilize network resources. The scenarios include bandwidth configurations of 100 Mbps, 500 Mbps, and 1000 Mbps to cover a spectrum of network capacities, ranging from limited connections to high-speed data transfers, as these values are common bandwidth values in current network environments.

4.1.3 Workloads

Several different kinds of workloads were intended to simulate real-world use scenarios, including request-response protocols, bulk data movement, and streaming applications. These workloads have been chosen with best practices on KVM for IBM Z [22], covering a broad spectrum of application behaviors, including small, frequent transactions and sustained high throughput data exchanges.

The **request-response** workload includes a variety of configurations ranging from small requests to solicit responses of arbitrary sizes. In these configurations, 10-byte for 1000-byte responses, along with larger combinations like 1000-byte requests, with 1000000-byte responses. It is by changing these request and response scales that it is analysed how QUIC manages small control messages against more meaningful data payloads. This workload mimics typical web interactions (e.g., API calls) or lightweight messaging apps, where latency and overhead reduction are important performance factors.

Both the **upload and download** scenarios were considered for evaluating continuous data transfer scenarios with file sizes of 100KB to 1GB. This choice allows testing QUIC's efficiency across a range of scenarios, from small files to large media file transfers, for example.

As well as data transfers, we included some streaming workloads to validate QUIC's appropriateness for real-time media transport. The selected bitrates, ranging from 200 Kbps to 20 Mbps, align with common streaming resolutions, from low-quality mobile streams to high-definition video content. Since QUIC is increasingly used in video streaming services, particularly by platforms like YouTube, it is essential to evaluate its ability to maintain stable throughput, adapt to network variations, and minimize buffering under different network constraints.

The goal of this study is to give a comprehensive overview by designing these network scenarios and workloads. A thorough examination of QUIC's metrics in various environments that demonstrates its strengths and possible weaknesses in real-world usage. The selected test parameters are in coherence with previous studies in QUIC performance evaluation, providing that the findings are significant to the contribution to the discussions on transport protocol optimization.

4.1.4 Performance Metrics

To assess the performance of QUIC in various network conditions, we focus on a subset of performance metrics that provide meaningful insights while introducing minimal measurement overhead. The selected metrics include:

- Average Round-Trip Time (RTT) - This metric captures the fundamental delay between sending a request and receiving a response, which is crucial for evaluating transport protocol efficiency. RTT is widely used in QUIC performance analysis as it directly influences congestion control decisions and data transfer speeds. He's also part of the QUIC protocol itself, as the handshake process is designed to minimize RTT and establish secure connections quickly.

- Average Throughput - This metric represents the data transmission rate, which means the amount of data that it transferred within the time of the experiment. It is important to understand the behaviour of QUIC implementations under different congestion control algorithms and network conditions.

- Largest Memory Chunk Size Used - This metric provides insights into memory consumption because it helps understand memory management by implementations, especially when it comes to dealing with lots of concurrent connections.

- CPU Time Used - Measuring CPU time allows us to measure the computation cost of various QUIC implementations. This helps in understanding how efficiently a implementation utilizes system calls and resources during the experiment.

Those referred metrics are collected in the client applications during the experiments to minimize the additional overhead. We have chosen this approach as it gives the insights we need by extracting from the applications directly using C-based system libraries without additional outside complex mechanisms. These metrics are CPU time and maximum memory chunk. Small data readers and writers are inherently monitored by the operating system, and accessing them does not impose significant overhead, which makes them practical for performance evaluation.

These four metrics create a balance of efficiency and insights, such that they ensure that performance measurements are correct without using unnecessary overhead. We understand key metrics such as RTT, throughput, memory usage, and CPU utilization, analysing every single one across varying network conditions while preserving a lightweight measurement.

4.1.5 Implementations

In this work, we chose to assess the performance of two QUIC implementations: MsQuic and Quiche. These implementations were chosen due to their popularity, and the fact that both are written in C allows them to be integrated into the QuicSand API framework. While MsQuic and Quiche give their client and server applications examples, these were not working for the purpose. Our need to control specific types of workload between clients and servers isn't fully captured by the examples presented by the implementations because they are too simple. Consequently, we created individual client and server applications conforming to the structure of the QuicSand API proposed earlier.

Initial Configurations. The main task was to provide consistency on the same testing conditions in the same framework. To achieve this, we launched both implementations matching parameters — buffer and window sizes, initial data sizes, and other necessary settings, as described in the table below:

Configuration	MsQuic	Quiche
Initial Congestion Window	*10	*10
Maximum Congestion Window	*100	*100
Initial Data Size	*1200	1200
Maximum Streams	*0	10000
Maximum Stream Data (Bidi Local)	*1048576	1048576
Maximum Stream Data (Bidi Remote)	*1048576	1048576
Maximum Stream Window	*16777216	16777216
Maximum Connection Window	*25165824	25165824
Congestion Algorithm	CUBIC	CUBIC
Idle Timeout	0 ms	0 ms
TLS Certificate Validation	Disabled (client mode)	Disabled (client mode)

Table 4.1: Configuration parameters for MsQuic and Quiche implementations. Default values are marked with * when not explicitly set.

As we can see from the table, the parameters were adjusted to preserve similar testing conditions. The biggest difference is that the number of default values for every parameter favouring MsQuic was intentional. Quiche has fewer default values by design, whereas most parameters in MsQuic were unchanged from their defaults. By aligning these two components, a meaningful and fair evaluation of both implementations is possible.

Concurrency. Both implementations are very different regarding concurrency in their approach. MsQuic supports thread-safe usage that allows it to handle multiple connections together in the same process. This gives efficient resource usage and better performance in high concurrency environments. On the other hand, Quiche is not thread-safe by default and so requires external synchronization to control multiple connections. It offers a user more raw control over the object, but this also adds additional effort to maintain connection thread safety and efficiency.

4.2 Results

This section provides the results for the performance evaluation of both MsQuic [32] and Quiche [10] implementations of QUIC with a range of workloads and network conditions. The performance metrics explained earlier are examined to produce assumptions about how implementations operate over different network scenarios, showing their advantages and possible limitations. We take the new results across configurations and use them to see patterns and trends that reveal the protocol's performance characteristics and behavior.

4.2.1 Results Selection Criteria

The results presented in this section are selected based on how relevant they are to the research and how they reveal QUIC's behavior under various conditions. Since our tool is designed to give a performance evaluation analysis of QUIC implementations, the results need to be selected in how well they represent the performance on controlled diverse network environments and workloads. So, the results were selected and limited to the following criteria:

- **Consistency:** Results that show consistent behavior over multiple configurations and workloads were prioritized as they represent the implementation's robustness and performance across varied environments.

- **Scalability:** Results that highlight the implementation's ability to scale with increasing client loads or network constraints were considered, as they provide insights into its performance in high-concurrency environments.

- **Redundant Results:** Graphics with redundant information and that do not present new behaviours of the protocol were excluded.

- **Representativeness:** Results that are representative of the performance of the protocol under multiple conditions and that represent the implementation's strengths and limitations.

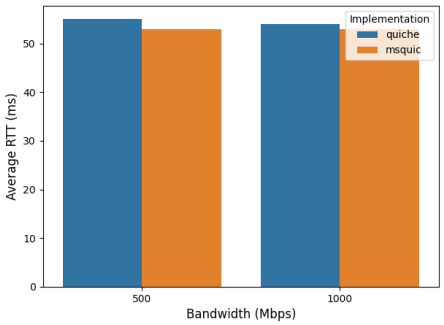
Using these criteria, we present results which provide an insight into QUIC's effectiveness in various scenarios, showcasing its advantages as well as its potential in real-world applications. The results selected convey some significant information about how each implementation performs across different network conditions and workloads, helping our understanding of the performance of the protocol.

Throughout the analysis, we are going to emphasise the performance metrics comparison over the workloads, as the graphics contain more information and are easier to understand. Nonetheless, every important insight over the performance of an implementation will be presented independently.

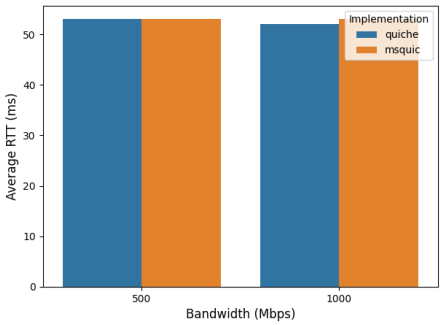
4.2.2 Average Round-Trip Time (RTT)

The average round-trip time (RTT) is one of the most important metrics for evaluating transport protocols performance as it can influence the user's experience and, in our case, is calculated by averaging the values of all clients in the test experiment. In this subsection, we are going to show how RTT performs under multiple network contraints and topologies.

Single Client, Single Server. We start by delving into single client-server topologies, as it represents the most high-performance scenario because the connection and the environment are completely isolated. As shown in Figure 4.3a, Quiche underperforms compared to MsQuic, especially under 500 Mbps of bandwidth. In Figure 4.3b, the case is slightly different as Quiche outperforms Msquic under 1000 Mbps of bandwidth. Besides, these slight differences, we can deduce that both protocols perform well under this scenario, as expected, because it is a situation where neither the workload nor network constraints are pronounced.



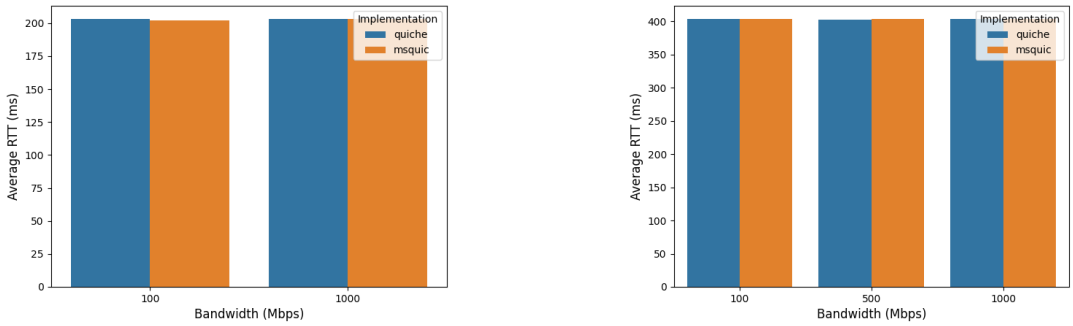
(a) Request: 10 bytes, Response: 1000 bytes



(b) Request : 10 bytes, Response: 10000 bytes

Figure 4.3: Average RTT (ms) vs Bandwidth (Mbps) for Latency 25 (ms), 1 Client and 1 Server.

In Figure 4.4 are represented experiments under the same workload and topologies, where the only varied contriants were the latency and the response size, corresponding to 100 ms and 10000 bytes in Figure 4.4a and 200 ms and 1000 bytes in Figure 4.4b. As can be seen, the values of the RTT remain even across both implementation as expected, because of the non-demanding workload. These scenarios represent all single client-server topology scenarios, as no significant change was noticeable.

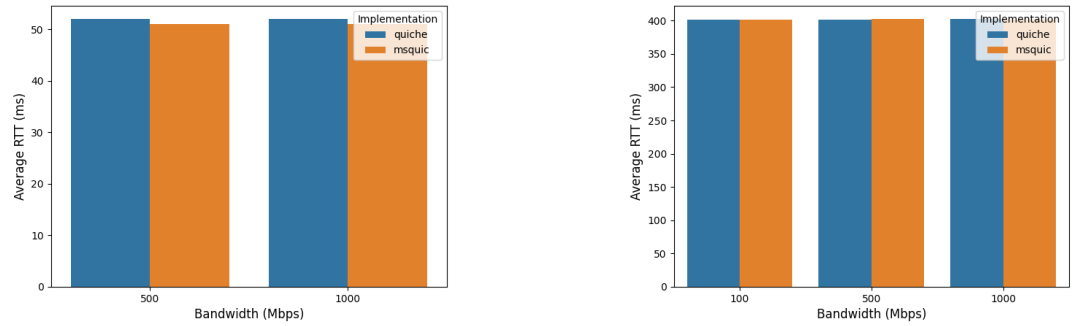


(a) Latency: 100 ms

(b) Latency: 200 ms

Figure 4.4: Average RTT (ms) vs Bandwidth (Mbps) for 1 Client, and 1 Server with 10 (bytes) Request and 10000 (bytes) Response.

Multiple Clients, Multiple Servers. In the next figures, an experiment over a 10-to-10 client-server topology where each client connects with a dedicated server, having a shared bottleneck point in the network. The results show that the values of RTT remain similar to the single client-server setup. As RTT is almost the same across implementations, this configuration is considered a significant example to measure the performance of schemes under such conditions. These results are shown in Figures 4.5a and 4.5b, showcasing the similar 25 ms and 200 ms latency request-response dynamics between both implementation links.



(a) Latency: 25 ms

(b) Latency: 200 ms

Figure 4.5: Average RTT (ms) vs Bandwidth (Mbps) for 10 Clients, and 10 Servers with 10 (bytes) Requests and 10000 (bytes) Responses.

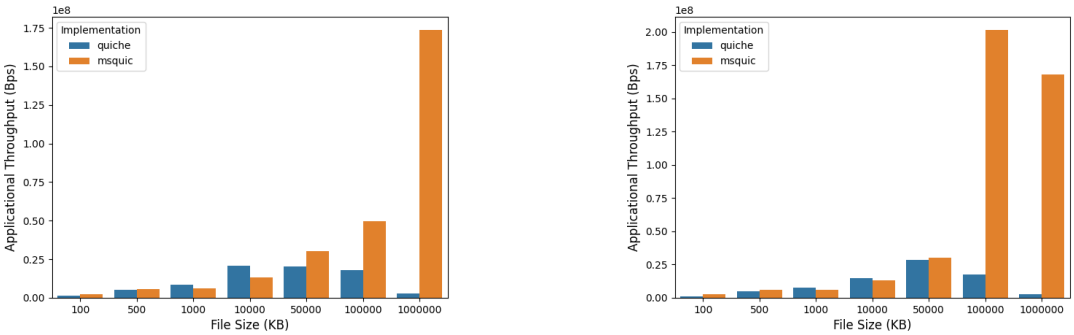
4.2.3 Applicational Average Throughput

This metric provides us with a more accurate comparison and perspective on the throughput of the QUIC implementations, as it does not rely on QUIC implementations' connection statistics. If we were to choose the protocol, the total bytes sent over the network would not be fair because implementations can differ on the number of packets sent over the network over the same data payload size. To address this, instead of using the total bytes sent by the protocol, we chose to calculate the total time that the test took to transfer the data.

In these experiments, we tested the application throughput in three types of Web Application interactions: upload, download, and streaming. The metric represents outgoing throughput in the upload and incoming throughput in the download and streaming.

Upload Workload. The upload workload consists of the data transfer of a file from the client to the server. In each test, just one file is transferred, ranging from 1 KB to 1 GB. The Figures that illustrate the results of the performance of this workload compare the average applicational throughput with the different file sizes, as they contain relevant information of the implementation performance under this metric.

Single Client, Single Server. The Figure 4.6a and Figure 4.6b illustrate results for network scenarios under 25 ms of latency with 500 Mbps and 1000 Mbps, respectively, where Quiche outperforms MsQuic for files with smaller sizes, in this case under 10MB. However, as the file sizes increase, MsQuic achieves superior throughput values and outperforming Quiche that presents very low performance values, proving that the implementations benefits from the high bandwidth availability, reaching a maximum average throughput of 200 MBps as represented in the Figure 4.6b compared to the maximum average throughput value of Quiche that rounds over 25 MBps.



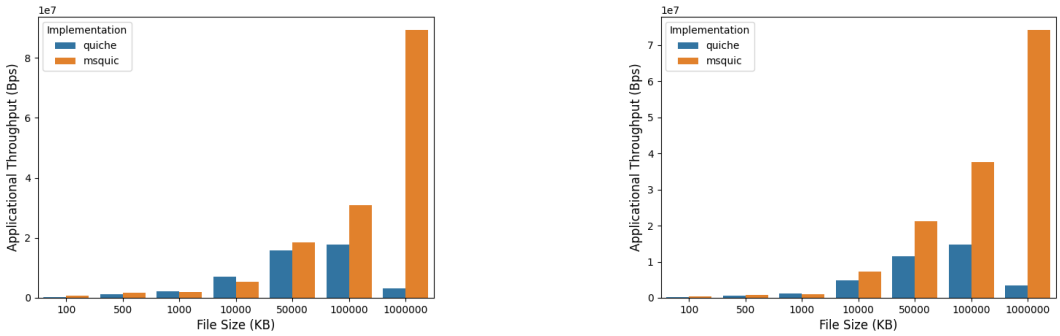
(a) Bandwidth: 500 Mbps

(b) Bandwidth: 1000 Mbps

Figure 4.6: Applicational Average Throughput (Bps) vs File Size (MB) for Latency 25 (ms), 1 Client, and 1 server.

We also analyzed the results under 100 ms and 200 ms latency conditions, as shown in Figure 4.7. The trends observed remain consistent with those from the 25 ms latency

scenario, with two key differences. First, the performance gap between MsQuic and Quiche widens slightly as latency increases, reinforcing MsQuic's ability to better handle higher delay environments. Second, under higher latency conditions, MsQuic surpasses Quiche even for the 10 MB file size, a behavior not observed in the 25 ms latency scenario.

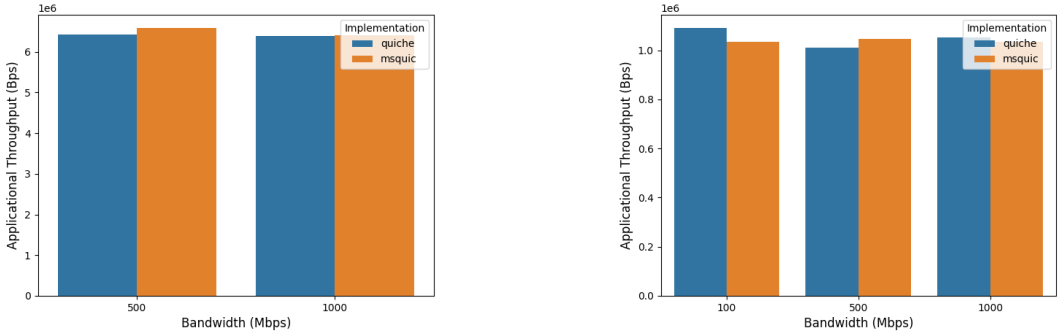


(a) Latency: 100 ms

(b) Latency: 200 ms

Figure 4.7: Applicational Average Throughput (Bps) vs File Size (MB) for 1 Client, and 1 Server.

Multiple Clients, Multiple Servers. For multiple clients connecting to multiple servers, we decided to show how the throughput behaves in a 10-to-10 client-server topology varies with bandwidth and file size. Since the more critical file size is the 10MB one, we decided to show how the bandwidth impacts the throughput for these workload experiments. The results are presented in Figure 4.8a and Figure 4.8b, where in the 25 ms latency scenario, MsQuic slightly outperforms Quiche across the 500Mbps bandwidth, on the other hand, Quiche outperforms MsQuic in the 1000Mbps and 100Mbps bandwidth over 200 ms latency. Since the values are not significantly different, we can conclude that both implementations are well-suited for handling multiple concurrent connections in a many-to-many transferring 10MBytes file.



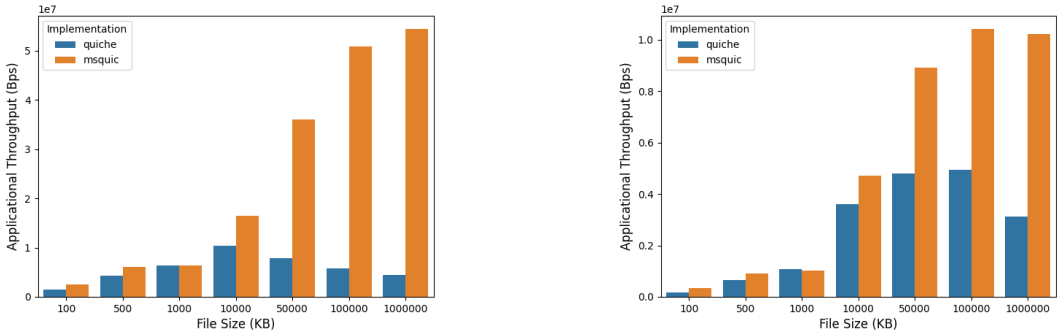
(a) Latency: 25 ms

(b) Latency: 200 ms

Figure 4.8: Applicational Average Throughput (Bps) vs Bandwidth (Mbps) for 10 Clients, and 10 Servers with 10 MB File.

Analyzing now the graphics over the file size, we can see that the throughput increases

with the file size, as expected. The results for the 25 ms and 200 ms latency scenarios are presented in Figure 4.9a and Figure 4.9b, respectively. These results show that the throughput values increase with larger file sizes in MsQuic, where the protocol matches the client's needs and the network constraints. On the other hand, Quiche has a more stable throughput, where the values are not significantly different over the file sizes. We also conclude that MsQuic is more efficient in handling uploads of larger files overall, as it can better utilize available bandwidth and optimize data transfer speeds.



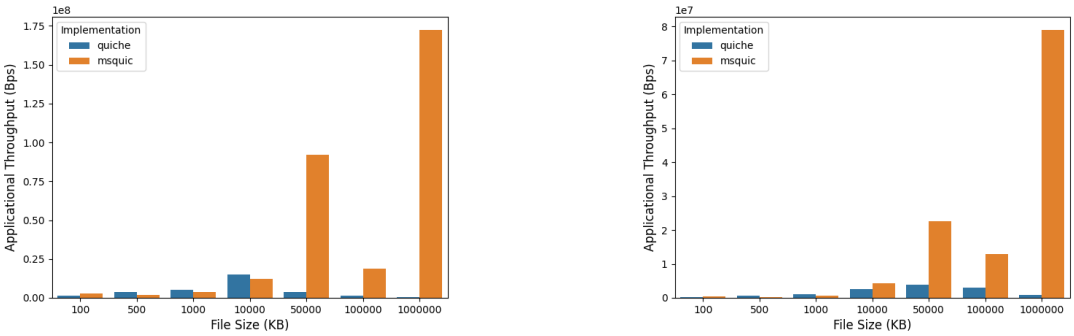
(a) Latency: 25 ms

(b) Latency: 200 ms

Figure 4.9: Applicational Average Throughput (Bps) vs File Size (MB) for 10 Clients, and 10 Servers.

Download Workload. The download interaction will give us an understanding on how implementations' incomming throughput varies with the different topologies and file sizes.

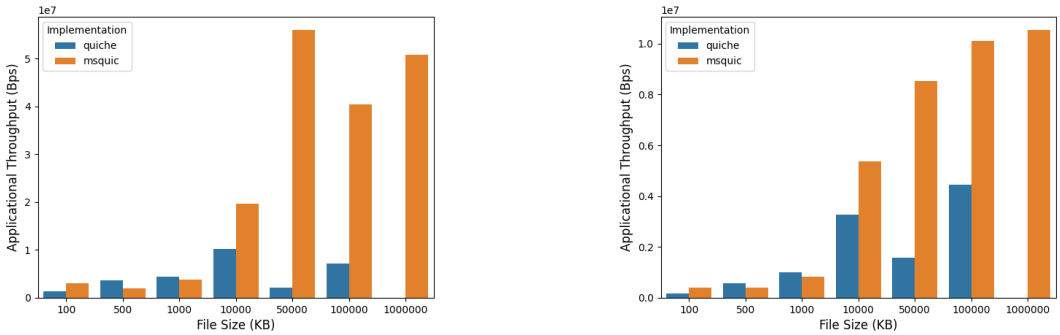
Single Client, Single Server. In this 1 to 1 topology, we decided to explore the same network scenarios presented in other workloads before, one with lower bandwidth (100 Mbps) and higher latency (200 ms) (Fig. 4.10b), and one with a higher bandwidth (1000 Mbps) and lower latency (25 ms) (Fig. 4.10a). In both graphics, the performance of the throughput is inconsistent for both implementations. In Figure 4.10a, Quiche still outperforms MsQuic for file sizes under 50 MB that match the upload applicational average throughput results. On the other hand, MsQuic still outperforms Quiche for higher volumes of files, but with an inconsistent growth, where at file size with 100 MB, the average applicational throughput showed some inconsistent results, dropping to a smaller value. We can also verify that the throughput values are relatively higher in the favourable network condition scenario (Fig. 4.10a).



(a) Latency: 25 ms, Bandwidth: 1000 Mbps (b) Latency: 200 ms, Bandwidth: 100 Mbps

Figure 4.10: Incomming Applicational Average Throughput (Bps) vs File Size (MB) for 1 Client, and 1 Server.

Multiple Clients, Multiple Servers. For multiple servers interacting with multiple clients in one-to-one connections, we decided to show results for 10 to 10 topologies to maintain the consistency in the results presented and to ensure a better and comprehensive evaluation. Figures 4.11 illustrate similar patterns compared to the results of the one-to-one topologies (Fig. 4.10), as the same inconsistency is presented. As expected, the values of the throughputs in both images dropped compared to the one-to-one scenarios because of the multiple concurrent flows. However, we can observe that the outperformance of Quiche over MsQuic has dropped compared to the previous topologies scenarios, as in both graphics we can see that MsQuic has a very considerable advantage over Quiche's incoming applicational average throughput for files with sizes above 1MB. We can conclude that there is a trend where, for files with smaller sizes and good network conditions, Quiche is slightly better than MsQuic compared with scenarios with worse bandwidth or bigger files where the incoming applicational average throughput has demonstrated to be substantially better.



(a) Latency: 25 ms, Bandwidth: 1000 Mbps (b) Latency: 200 ms, Bandwidth: 100 Mbps

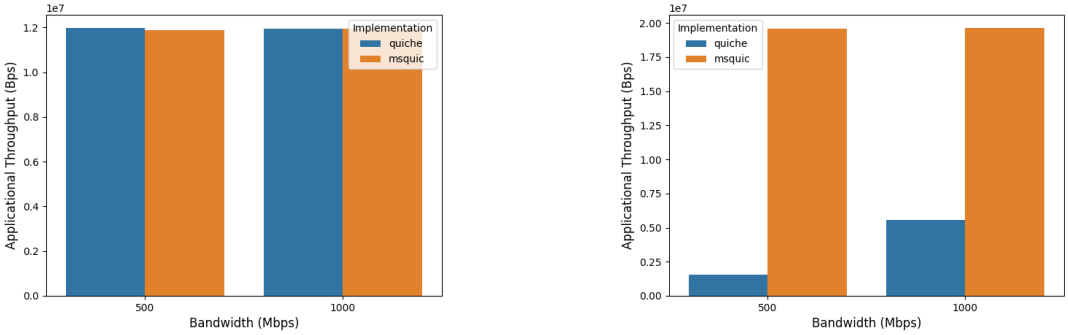
Figure 4.11: Incomming Applicational Average Throughput (Bps) vs File Size (MB) for 10 Clients, and 10 Servers.

Streaming Workload. Despite the streaming workload being similar to upload, the

results are indeed interesting due to streaming being a data transfer that controls the data flow to match the client's bitrate workload speed.

In this workload, we found it more relevant to illustrate how throughput varies with bandwidth, as these metrics are closely related. We began by analyzing two single-client, single-server scenarios, where the client's bitrate was set to 12 Mbps and 20 Mbps. The results are presented in Figure 4.12.

Figure 4.12a demonstrates that both protocols exhibit similar performance at 12 Mbps, aligning with the trends observed at lower bitrates. Conversely, Figure 4.12b shows that MsQuic outperforms Quiche at 20 Mbps, where a noticeable impact of lower bandwidth on Quiche's performance can be observed, whether this effect wasn't verified in MsQuic.

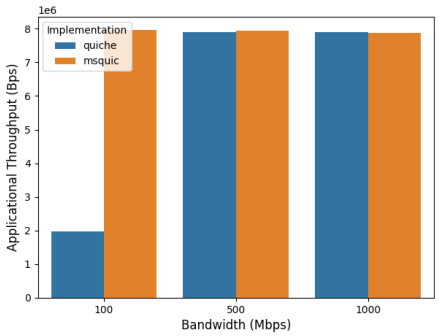


(a) Bitrate: 12000 Kbps

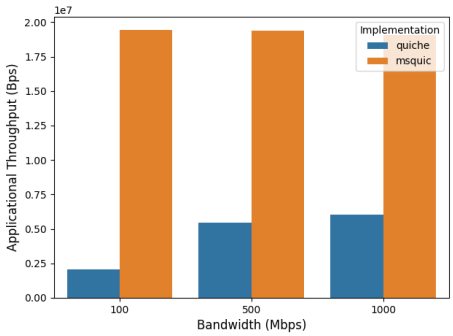
(b) Bitrate: 20000 Kbps

Figure 4.12: Applicational Average Throughput (Bps) vs Bandwidth (Mbps) for Latency 25 (ms), 1 Client, and 1 Server.

These next two graphics illustrate how the throughput varies with the bitrate and the bandwidth for 200 ms latency under 8Mbps and 20Mbps bitrate scenarios. We chose 8Mbps of bitrate because it is the first that achieves significant results. As can be seen in Figure 4.13a, the MsQuic performs well in every bandwidth, otherwise, Quiche has a significant drop in throughput when the bandwidth is 100Mbps. In Figure 4.13b, the results are similar to the 25ms latency scenario, where MsQuic outperforms Quiche in every bandwidth. It is also noticeable that the throughput in the Quiche protocol has a pattern, as the throughput decreases with the bandwidth decrease.



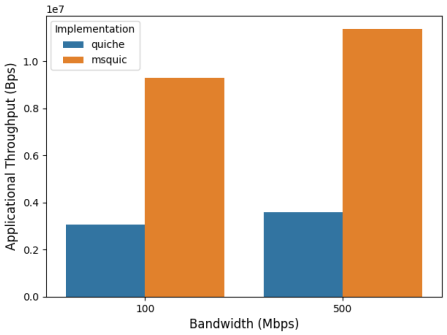
(a) Latency: 200 ms, Bitrate: 8000 Kbps



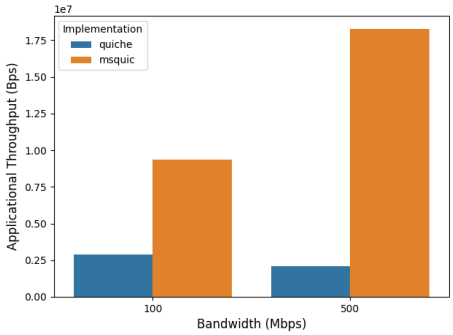
(b) Latency: 200 ms, Bitrate: 20000 Kbps

Figure 4.13: Applicational Average Throughput (Bps) vs Bandwidth (Mbps) for 1 Client, and 1 Server.

For the 10-to-10 client-server topology, the most relevant findings are presented in Figure 4.14. The throughput evaluation in this scenario confirms that MsQuic consistently outperforms Quiche across all bandwidth and bitrate configurations. Unlike the single-client, single-server setup, where each protocol operates in isolation, this scenario introduces the additional complexity of handling concurrent flows. Despite this, the results demonstrate that MsQuic effectively manages multiple connections, with clients collectively approaching the network's maximum capacity. Specifically, for a 100 Mbps link shared among 10 clients, each client achieves close to the expected 10 Mbps throughput, showcasing efficient bandwidth utilization.



(a) Latency: 100 ms, Bitrate: 12000 Kbps



(b) Latency: 100 ms, Bitrate: 20000 Kbps

Figure 4.14: Applicational Average Throughput (Bps) vs Bandwidth (Mbps) for 10 Clients, and 10 Servers.

We found it particularly insightful to highlight the evaluation of throughput in the 10-client, 10-server scenario with an 8000 Kbps bitrate under 100 ms latency. The results, presented in Figure 4.15, reveal an unexpected drop in throughput for the Quiche protocol when the bandwidth is set to 500 Mbps. This behavior is anomalous, as a higher available bandwidth should not negatively impact throughput. Ideally, the protocol should sustain the expected 8000 Kbps throughput, yet the observed performance deviates from this

expectation.

This finding is particularly noteworthy as it suggests a potential limitation in Quiche's ability to handle multiple concurrent connections in high-bandwidth environments. The issue is likely rooted in its flow and congestion control mechanisms, as similar behavior is consistently observed across other test scenarios.

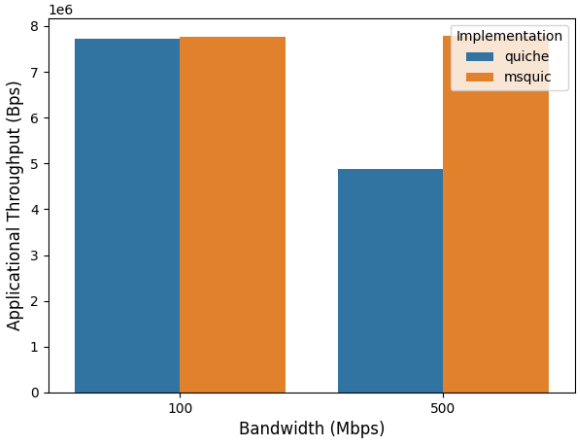


Figure 4.15: Applicational Average Throughput (Bps) vs Bandwidth (Mbps) for 10 Clients, and 10 Servers with 8000 (Kbps) bitrate

4.2.4 Memory Usage

Memory usage is a metric that helps understanding how QUIC implementations manage their memory. In this study, we decided to measure the larger memory chunk size used by both implementations in the different workloads and network scenarios, as these metrics can give developers insights on which implementation is best well-suited for their machines and networks' resources.

Single Client, Single Server. We have tested the memory usage for single client-server topologies under multiple network conditions and workloads. We decided to present and analyse the results with the same scenarios of the average round-time time (RTT) and average applicational throughput to provide consistency and to relate results with these previously analysed metrics.

In the network scenario, where latency is 25 ms with 500 Mbps as the link-level constraints, as illustrated in Figures 4.16 and 4.17, Quiche demonstrates a tendency to hold smaller chunks of memory compared to Msquic. Besides the fact that Figure 4.16 demonstrates a consistent value for the request and response workload, the same is not verified for the upload workload, illustrated in Figure 4.17, showing an inconsistent growth of the largest max memory chunk with the increase of the file sizes. These behaviours were expected, as Quiche is a lightweight non-thread-safe implementation compared to MsQuic.

For workloads operating under favorable network conditions, where MsQuic does not demonstrate a clear advantage over Quiche, the results maintain the same trend, where Quiche consistently outperforms MsQuic by consuming less memory.

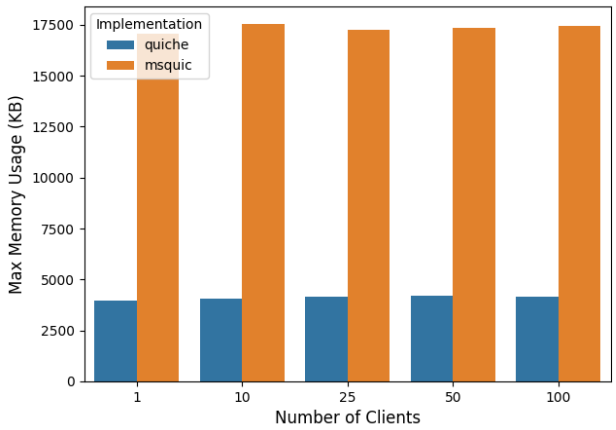


Figure 4.16: Largest Memory Chunk Size (bytes) vs Number of Clients for Latency 25 (ms), Bandwidth 500 (Mbps), 1 Server, and 10 (bytes) Requests and 1000 (bytes) Responses

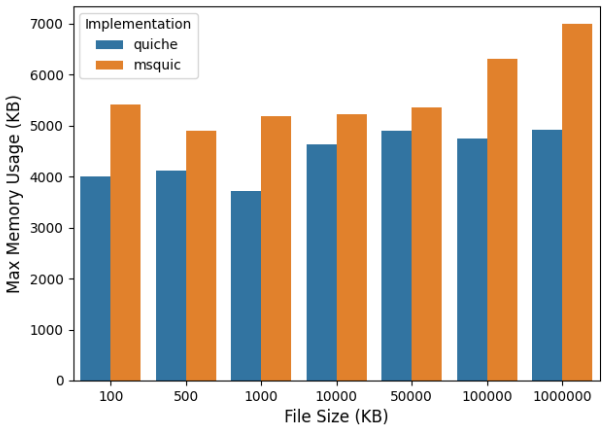


Figure 4.17: Largest Memory Chunk Size (Bytes) vs File Size (MB) for Latency 25 (ms), Bandwith 500 (Mbps), 1 Client, and 1 Server

Challenges arise when an implementation starts underperforming due to the network conditions in combination with the workload demands. Figure 4.18 illustrates a scenario where the Quiche implementation starts underperforming without reaching the client's bitrate requirements, leading to inefficient memory management under different streaming workloads. The results demonstrate that Quiche, at certain points, struggles to manage memory, especially for higher volumes of data transfer.

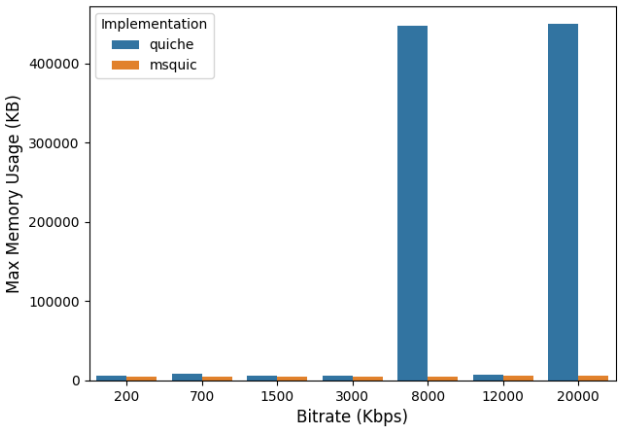


Figure 4.18: Largest Memory Chunk Size (Bytes) vs Bitrate (Kbps) for Latency 25 (ms), Bandwidth 500 (Mbps), 1 Client and 1 Server

Multiple Clients, Multiple Servers.

The complexity of memory management increases with a growing number of clients and servers, as concurrent flows add worse performance of the implementations and postriori some retention on the endpoints. In these cases, effectively dealing with multiple concurrent connections becomes key in maintaining performance. Memory allocation and deallocation need to be well optimized, as it leads to the wastage of resources consumption, and eventuality that can have effects on the overall stability of the system.

As shown in Figure 4.19, under concurrent flows, Quiche has a much lower performance compared with MsQuic on the client side. The inefficiency might be from its inability to serve multiple concurrent connections, leading to excessive contention. While analyzing Quiche as a standalone, Quiche has performance degradation at higher bitrates in the streaming workload. This is a pattern seen in other workloads as well, emphasizing that Quiche fails to manage its memory efficiently in the face of higher volume of data transmissions.

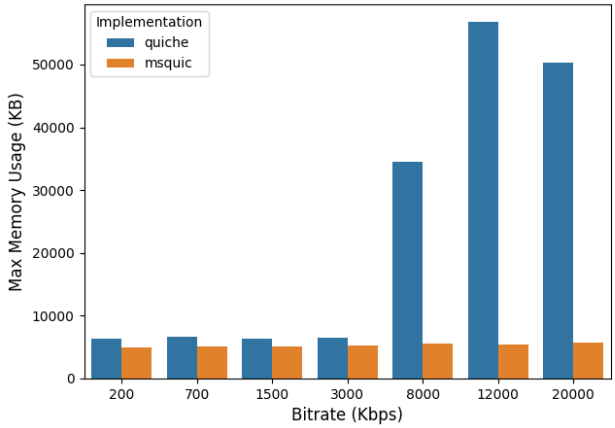
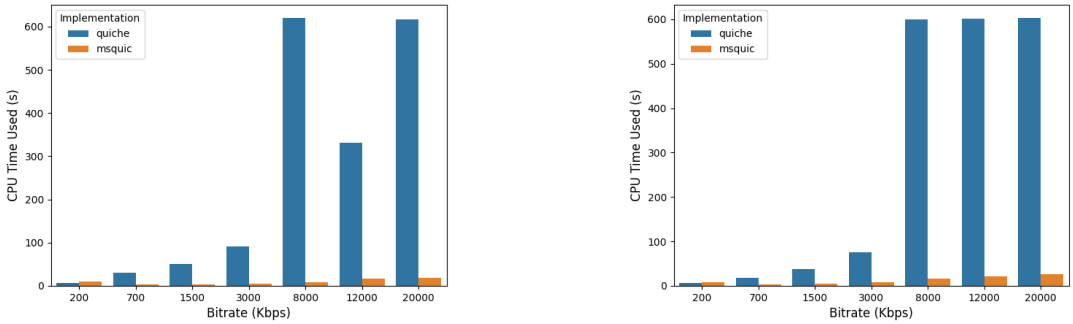


Figure 4.19: Largest Memory Chunk Size (bytes) vs Bitrate (Kbps) for Latency 25 (ms), Bandwidth 500 (Mbps), 10 Clients, and 10 Servers

4.2.5 CPU Usage

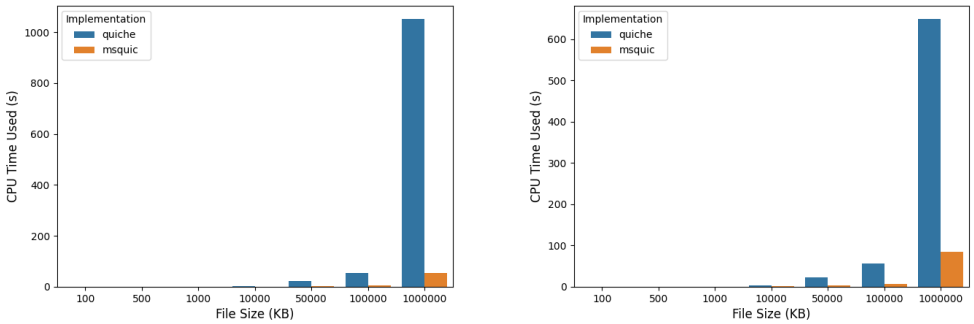
The CPU usage metric gives an idea of the computational overhead of a QUIC implementaion. It is mainly quantified based on CPU time used by different QUIC implementations under varied workloads and network scenarios. By CPU used time analysis, it allows us to quantify how each implementation handles their protocol operations, packet processing and congestion control mechanisms. High CPU used time indicates poor resource management, while low CPU used time implies that they function at a high-performance level and scale. We chose to calculate CPU time used by the implementations in an experiment because it provides information about how the implementations manage the CPU resources.

To provide a clearer understanding of CPU efficiency, we rearrange the order of workload analysis. This reordering is motivated by the fact that MsQuic consistently outperforms Quiche in streaming, upload, and download workloads across nearly all scenarios, demonstrating lower CPU time usage. Figures 4.20 and 4.21 illustrate the CPU usage for streaming and upload workloads, respectively, under 25 ms latency and 500 Mbps bandwidth conditions for both the 1-to-1 and 10-to-10 topologies. Other figures are omitted here, as the results are not significantly different from the ones presented.



(a) 1 Client, 1 Server (b) 10 Clients, 10 Servers

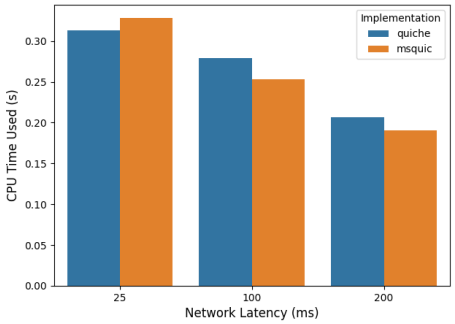
Figure 4.20: CPU Usage for Streaming Workloads



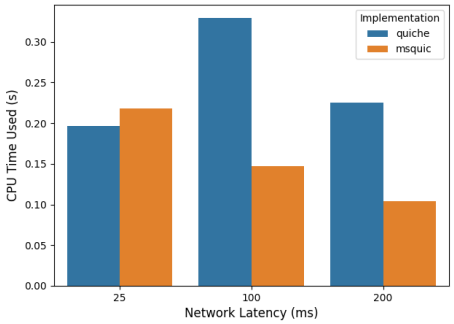
(a) 1 Client, 1 Server (b) 10 Clients, 10 Servers

Figure 4.21: CPU Usage for Upload Workloads

Conversely, in the request-response workload, a different trend is observed. MsQuic maintains its superior efficiency across all scenarios and workloads, except in the case of the first workload, where the request size is 10 bytes, with 25 ms latency and 1000 Mbps bandwidth, in a 1-to-1 client-server topology. In this particular instance, Quiche shows a slight advantage over MsQuic, as shown in Figures 4.22a and 4.22b. This behavior is likely due to the minimal data transfer requirements, which favor Quiche's lightweight implementation. However, MsQuic quickly regains its efficiency advantage at lower bandwidth scenarios, as illustrated in Figure 4.23, where MsQuic already outperforms Quiche in both workloads.

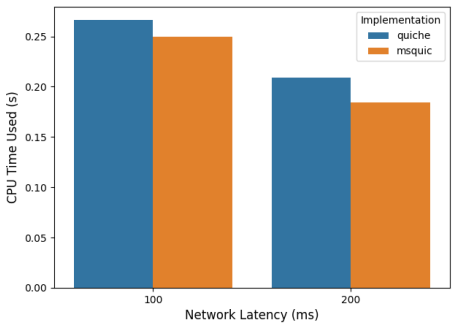


(a) Request size: 10 bytes, Response size: 1000 bytes

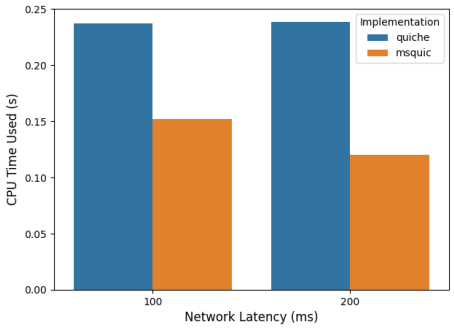


(b) Request size: 10 bytes, Response size: 10000 bytes

Figure 4.22: CPU Usage for Request-Response Workloads with Bandwidth 1000 (Mbps)



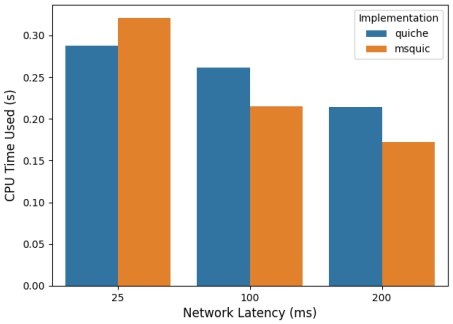
(a) Request size: 10 bytes, Response size: 1000 bytes



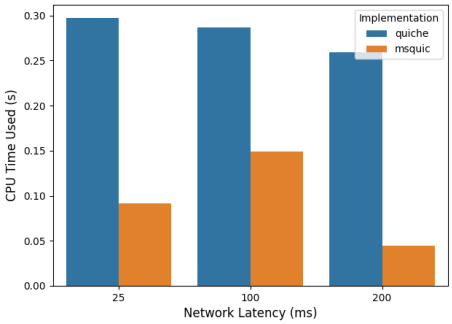
(b) Request size: 10 bytes, Response size: 10000 bytes

Figure 4.23: CPU Usage for Request-Response Workloads with Bandwidth 100 (Mbps)

In the 10-to-10 client-server topology, the results remain consistent with the previous findings: MsQuic maintains its efficiency advantage over Quiche across all scenarios and workloads, except in high-bandwidth (1000 Mbps) scenarios. In this specific case, Quiche outperforms MsQuic when handling a request size of 10 bytes and a response size of 1000 bytes, as shown in Figure 4.24a. However, as depicted in Figure 4.24b, and the result being representative of the ones not presented, Quiche no longer maintains an advantage in the remaining scenarios.



(a) Request size: 10 bytes, Response size: 1000 bytes



(b) Request size: 10 bytes, Response size: 10000 bytes

Figure 4.24: CPU Usage for Request-Response Workloads with Bandwidth 1000 (Mbps)

4.2.6 Extended MsQuic Performance Analysis in High-Concurrency Topologies

In this analysis, we have covered a set of scenarios that compared both implementations, where we chose the most interesting results that show more understanding of the implementations' performance in those scenarios. Besides these comparisons, lots of scenarios were not presented due to extreme performance disparities, in particular in many-to-one topologies where multiple clients connect to a single server, exposing how implementations handle concurrency.

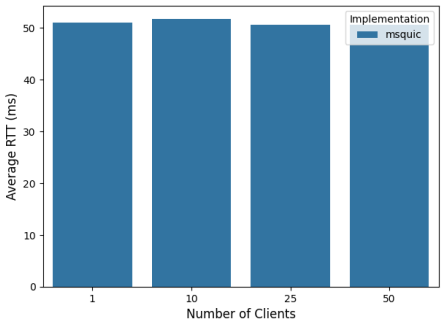
This subsection provides more insights on these topologies in the performance of MsQuic implementation that showed the best overall performance, and that includes thread-safe mechanisms. Additionally, this subsection explores QuicSand tool features on covering multiple different topologies.

Many-to-One Topologies. In these topologies, multiple clients connect to only one server to understand how optimized QUIC implementations are on handling concurrency in the endpoints. These topologies not only show how concurrency is handled on the server endpoint, but it maintains a common bottleneck point in the network, maintaining testing over concurrent flows in a network link.

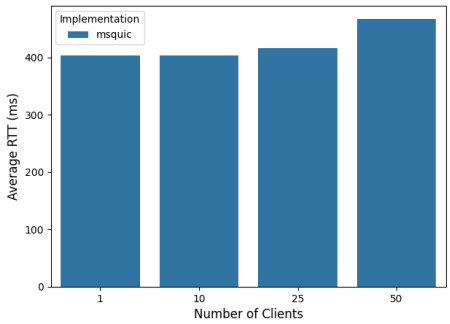
Request-Response Workload. The request-response workload is the least demanding scenario of the four types of workloads that QuicSand include. However, it maintains useful insights over implementation behaviour in these types of scenarios with multiple connections into a single server endpoint. Our results show how the number of clients (number of concurrent connections) impacts the performance metrics, including RTT, memory and CPU usage.

The impact in RTT is illustrated in Figure 4.25, where we present both a least constrained network scenario (25 ms of latency and 1000 Mbps bandwidth) and a more constrained network scenario (200 ms latency and 100 Mbps bandwidth). In both cases, the request size is set to 1000 bytes, while responses consist of 1,000,000 bytes, representing tests with

larger payloads. Figure 4.25a shows that RTT remains relatively stable as the number of clients increases, except in the more constrained scenario (Figure 4.25b), where an increase is observed when the client count reaches 25 and 50. This behavior is expected, as lower bandwidth and higher concurrency naturally contribute to increased RTT. The other results are not presented as the RTT is relatively stable even when increasing the number of clients.



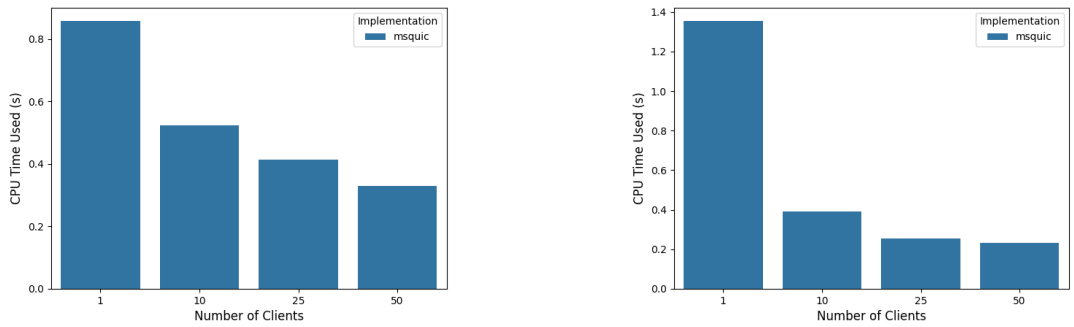
(a) Latency: 25 ms, Bandwidth: 1000 Mbps



(b) Latency: 200 ms, Bandwidth: 100 Mbps

Figure 4.25: Round-Trip Time (ms) vs Number of Clients for Request-Response Workloads with 1000 (bytes) Requests and 1000000 (bytes) Responses

Figure 4.26 presents the CPU usage under the similar network conditions as the RTT analysis, the least constrained scenario with 25 ms latency and 1000 Mbps bandwidth, and one more constrained scenario with 100 ms latency and 100 Mbps bandwidth. The result shows how the CPU used time is related to the increase of the clients connected to the server. Both figures highlight that the increase in the client number leads to a decrease in the CPU used time. This behaviour is expected because the request-response workload tests are controlled by the time of the tests, since the increase of the clients leads to a higher volume of data being sent over the network and, therefore, a decrease in the number of requests sent. So the CPU used time is expected to decrease since less CPU is used in the endpoints, as demonstrated in the Figure 4.26a and Figure 4.26b.



(a) Latency: 25 ms, Bandwidth: 1000 Mbps (b) Latency: 100 ms, Bandwidth: 100 Mbps

Figure 4.26: CPU Usage (ms) vs Number of Clients for Request-Response Workloads with 1000 (bytes) Requests and 1000000 (bytes) Responses

The values of the memory usage are consistent all over the scenarios, as the memory usage is stable across different numbers of clients. The results represent the memory usage in all case scenarios with the values fluctuating around 5000 KB, represented in Figure 4.27.

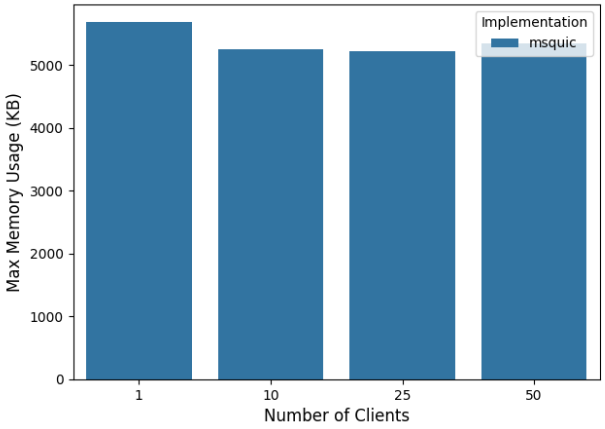
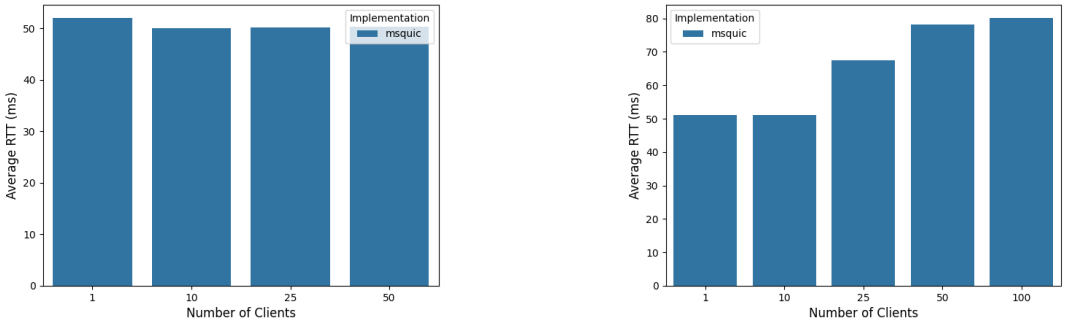


Figure 4.27: Largest Memory Chunk Size Used (bytes) vs Number of Clients for Request-Response Workloads with 1000 (bytes) Requests and 1000000 (bytes) Responses

Streaming Workload. The streaming workload is a more demanding network scenario, as clients receive data from the servers at a continuous rate. This workload is particularly challenging in many-to-one topologies since it requires clients and the server to deal with concurrent connections and higher volumes of data. Our analysis revealed information on the performance of the MsQuic implementation in the RTT and incoming applicational throughput metrics.

The RTT analysis, presented in Figure 4.28, compares the RTT with the number of clients connected to the server under different bitrate conditions. Both figures represent experiments that used favorable network conditions, with 25 ms latency and 1000 Mbps bandwidth. In Figure 4.28a, where the bitrate is set to 200 Kbps, the RTT remains relatively

stable independently of the number of clients connected to the server, getting used to favourable network conditions. However, in Figure 4.28b, where the bitrate is significantly higher at 20000 Kbps, the RTT increases as more clients are added, highlighting the impact of higher bitrates on the RTT.

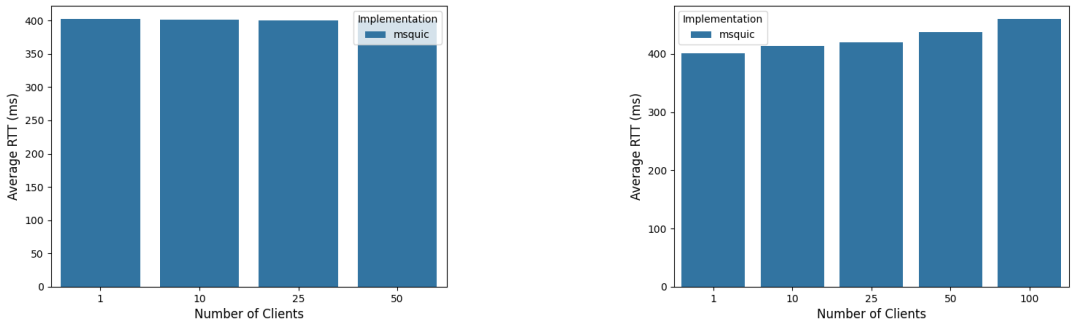


(a) Bitrate: 200 Kbps

(b) Bitrate: 20000 Kbps

Figure 4.28: Round-Trip Time (ms) vs Number of Clients for Streaming Workloads with Latency 25 (ms) and Bandwidth 1000 (Mbps)

Figure 4.29 presents two graphics where the RTT was analised for a more constrained network scenario, with 200 ms latency and 100 Mbps bandwidth, comparing 200 Kbps and 20000 Kbps bitrates. The results follow a similar pattern comparing with the scenario presented before (Fig 4.28b), where RTT remains stable for lower bitrates but increases as number of clients grow for higher bitrates. However, the rate of RTT increase is less pronounced compared to the previous scenario, as the already constrained network conditions impose more limitations.



(a) Bitrate: 200 Kbps

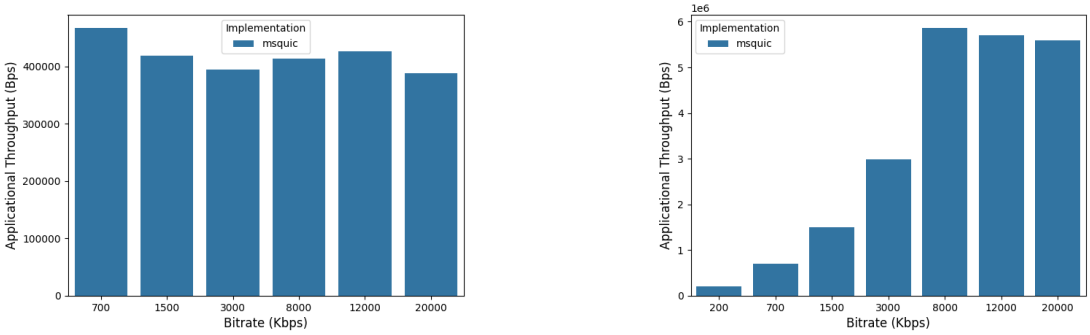
(b) Bitrate: 20000 Kbps

Figure 4.29: Round-Trip Time (ms) vs Number of Clients for Streaming Workloads with Latency 200 (ms) and Bandwidth 100 (Mbps)

For the incoming applicational average throughput, we selected the figures that tested the streaming workload under 25 ms of latency and 1000 Mbps of bandwidth. Different from other metrics we decided to compare the different topologies, the many-to-one topology and the many-to-many, where we decided to present the experiments that were

executed under 100 clients connecting to a single server and 100 clients connecting to 100 servers.

Figure 4.30a shows how applicational throughput varies with different bitrates, where we can see that the value of the throughput is relatively constant under the different bitrates, fluctuating around 400 and 500 KBps. On the other hand, in Figure 4.30b, we can see the impact of the bitrate on throughput. In these cases, the throughput achieved higher values compared to the 100 clients to a single server scenario, reaching values close to 6000 KBps in the tests that were executed over a required bitrate higher than 8000 Kbps. We can conclude that the throughput of the execution tests over the many-to-one scenario was highly compromised by the concurrency on the server, leading to a considerably lower performance.



(a) 1 Server

(b) 100 Servers

Figure 4.30: Applicational Average Throughput (Bps) vs Bitrate for Streaming Workloads with Latency 25 (ms) and Bandwidth 1000 (Mbps)

This final throughput analysis highlights how the MsQuic implementation successfully maintains the proposed bitrate throughput up to 8000 Kbps, beyond which throughput starts to decline. This behavior aligns with expectations, as the network conditions become insufficient to support the increasing data bitrates. As illustrated in Figure 4.31, the network constraints, particularly the 200 ms latency and 100 Mbps bandwidth, directly impact the overall throughput performance of the protocol, reinforcing the role of network limitations in QUIC's efficiency under high-bitrate streaming workloads.

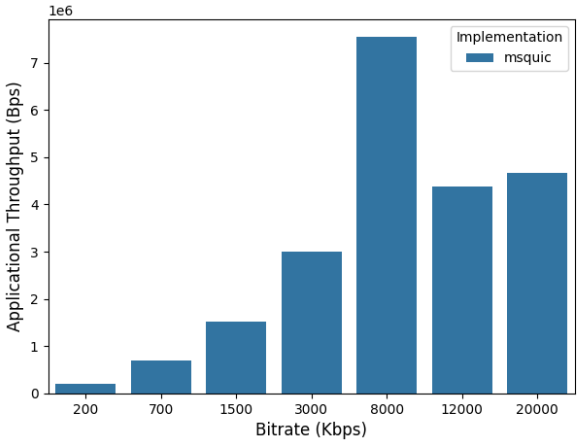


Figure 4.31: Applicational Average Throughput (Bps) vs Bitrate for Streaming Workloads with Latency 200 (ms) and Bandwidth 100 (Mbps)

Lastly, since the results for CPU usage, memory usage, and upload/download workloads do not introduce any insights beyond those already discussed in previous sections, they are omitted from this analysis. Including them would be redundant, as their patterns and behaviors follow the same trends observed in earlier evaluations.

4.2.7 Discussion

The results presented in this work demonstrate the effectiveness of QuicSand in systematically evaluating QUIC implementations under diverse network conditions. The tool's structured designed provided an ease approach to test the performance of the implementations under equally workloads and network conditions, where the only requirements needed for the tool to execute the experiments is the configuration tests and topologies file, ensuring that the only parameter changed during the tests execution was the name of the implementation, to guarantee results for both implementations.

By combining every single QuicSand feature, we can validate that the tool lies in its ease of use and extensibility, allowing researchers to quickly adapt to new workloads and network conditions through simple configuration files without modifying the tool's logic.

Beyond the experiments presented by this work, QuicSand's extensible design allows integration of new features, such as protocol configurations, inclusion of new QUIC implementations and inclusion of new network constraints with minimal effort. This systematic approach and automation establishes QuicSand as a tool for future investigations into QUIC implementations' performance evaluation. investigations into QUIC implementations' performance evaluation.

4.3 Summary

In this chapter, we presented the methodology of the selected MsQuic and Quiche QUIC implementations included in our tool to test in different workloads and network scenarios. It included the explanation of those workloads and network scenarios, and additionally, the performance metrics selected to evaluate the performance of the QUIC implementations and the experimental setup where the tests were executed.

In the chapter, we also presented results, where we started by presenting the criteria in which the presented graphics and experiments were selected, then we discussed the obtained results comparing both MsQuic and Quiche QUIC implementations, finishing with a more detailed discussion over MsQuic performance.

5

Conclusion and Future Work

This chapter presents the conclusions from the results obtained in the previous chapter and future work that can be made over this study. Section 5.1 summarizes the main findings of the study, highlighting the insights into QUIC performance under different network conditions. Section 5.2 presents the future work that can be made to improve the QuicSand tool and the overall research of the performance of QUIC implementations.

5.1 Conclusion

The emergence of new QUIC implementations come with the difficulty to test and compare multiple different implementations to know wich one best suites the apllicational needs under different real-world network conditions. One of the most critical challenges for developers is find a good way to address all these problems in a systematiically, automated and easy way of deployment to test and evaluate QUIC implementations under multiple scenarios.

In this study, we have developed a tool that systematically and in an (mostly) automated way tests QUIC implementations under multiple network conditions and considering several relevant performance metrics such as RTT, throughput, CPU usage and memory usage, revealing implementation strengths and limitations. We compared two C implementations MsQuic and Quiche. MsQuic has shown throughout the study to outperform in most cases but also presents its weaknesses. For instance, MsQuic showed to perform better in terms of CPU usage compared to Quiche, but in smaller request and response workloads and some network conditions, Quiche demonstrated an advantage over MsQuic. Nevertheless, Msquic has an overall better performance in every single metric evaluated and consistent results over different workloads. On the other hand, Quiche showed a worse performance and more inconsistency throughout the performance evaluation.

QuicSand has emerged as an interesting tool to measure QUIC performance in a controlled network environment. Testing on the cloud also provides a flexible, scalable and automated testing environment; the tool empowers researchers to perform in-depth performance analysis on different QUIC implementations in diverse network conditions.

The modular architecture of the tool, its extensible approach and ease of use enables scientists to test diverse performance metrics and scenarios. The results of this study show that the tool is effective in assessing QUIC performance and suggest the need for more efforts to be done in this field. The results obtained were generated from a fully automated system that shows the tool's capability of performing large-scale tests with few human-assisted assessments. A key feature of the technology is this automation capability of the tool, allowing researchers to assess tool performance rapidly.

During the development and testing of the tool, the following challenges arose: problems with network emulation, the integration of QUIC implementations, and the development of general-purpose client-server applications. MsQuic had the benefit of being much simpler to integrate due to being a higher-level API and has inbuilt optimized ways for threading and concurrency. In contrast, we had a harder time integrating Quiche as its lower-level API requires a more detailed implementation where the developer needs to address more design choices. This can impact the accuracy and consistency of the performance evaluation. Nevertheless, this work contributes to a better understanding of how QUIC behaves in various network environments and demonstrates QuicSand s opportunities to further explore this area.

Comparing the two implementations, MsQuic is a much more developer-friendly option, which provides a high-level API for maximum potential concurrency and resource management. Quiche itself, on the other hand, is a lower-level API, leading developers to have more design choices and flexibility. For applications that require both high performance and a large amount of concurrency without the need to put in a lot of effort, MsQuic is the better choice. However, for situations where a low-overhead solution is needed for small workloads, Quiche might be more appropriate on resource-constrained system

Finally, this work has helped better understand QUIC performance and to achieve its goal of providing an automated and extensible QUIC performance evaluation tool in heterogeneous networking environments. While goals were reached, many improvements can take the tool to the next level, as we will present in the next section.

5.2 Future Work

While this research has been looking into the performance of MsQuic and Quiche and the development of a systematic and automated testing tool, providing some insights under controlled network conditions, there are still many aspects to explore and improve.

An important avenue for future work is improving the scalability of the testing framework. The tool shall be optimized in handling network scenarios for more complex, bigger topologies with more client and server simulations running at the same time. This includes mechanisms such as resource allocation and dynamic scaling that can minimize performance degradation in large-scale setups, either via automated optimization or via adding hardware resources dynamically.

An additional improvement over the performance in Quiche's server concurrency model. An exploration into different concurrency models may lead to better performance. Using a more complex strategy for setting QUIC protocol parameters can help to fit emerging technologies and understand how they behave under different network scenarios.

Another thing that could be improved is the accuracy of network emulation. The current test infrastructure uses *GONE*, Docker Swarm, and containerized deployments for network emulation. Future work could build on the current effort with added real-world scenarios. Inclusion of multi-failure conditions, e.g., random packet corruption, packet loss, jitter variations, and asymmetric bandwidth limitations to more accurately model real network conditions. Additionally, improving automation in the execution of tests, collection of test logs, and analysis would facilitate their large-scale testing and also decrease human effort.

This study is limited to the controlled, containerized environments, but future work could generalise tests to operational deployments. Instead, tests would be running inside a cloud environment, which would contribute to understanding QUIC's flexibility outside simulated environments, enabling insights into how different implementations behave in a real network.

Improving data visualisation and benchmarking the results presentation of the performance metrics, real-time visualizations, and more statistical comparisons could also improve the research work. Improvement in log formatting and interactive dashboards would also offer detailed visibility into the protocol behavior, which simplifies the test results interpretation.

Additional effort can be made to expand the QuicSand tool in other programming languages beyond C. While the current implementation offers extensibility for C-based implementations, other implementations can't be added due to the programming language barrier. These will add robustness to the tool and additional performance evaluations across QUIC implementations that use different programming languages.

By filling gaps in these areas, future work can refine the capabilities and generalize the robustness of the testing tool, helping to provide a more complete overview of QUIC. These improvements would not only increase the protocol evaluation but also facilitate the real-world application in scenarios with strict performance requirements.

Bibliography

[1] A. L. et al. "The QUIC Transport Protocol: Design and Deployment". In: *ACM SIGCOMM* (2017). DOI: 10.1145/3098822.3098842. URL: https://research.google/pubs/the-quic-transport-protocol-design-and-internet-scale-deployment/ (cit. on p. 24).

[2] *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. 5).

[3] D. Antunes. "Towards Generic and Scalable Network Emulation". MA thesis. Faculdade de Ciências da Universidade de Lisboa, 2025 (cit. on p. 30).

[4] M. S. Baptista and J. M. V. Sobral. "Jitter in IP networks: A Cauchy approach". In: *International Journal of Network Management* 20.4 (2010), pp. 268–282. DOI: 10.1002/nem.773. URL: https://doi.org/10.1002/nem.773 (cit. on p. 26).

[5] 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. 15).

[6] 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. 15).

[7] R. Bruenig. *A performance measurement tool for QUIC similar to iperf*. https://github.com/rbruenig/qperf. [Online; accessed 9-Fev-2024] (cit. on p. 27).

[8] 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. 16, 27).

[9] 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. 5).

[10] Cloudflare. *quiche*. Accessed: 2025-04-02. 2020. URL: https://github.com/cloudflare/quiche (cit. on pp. 36, 50).

[11] 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. 13).

[12] 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. 5, 12).

[13] *Domain names - concepts and facilities*. RFC 1034. 1987-11. DOI: 10.17487/RFC1034. URL: https://www.rfc-editor.org/info/rfc1034 (cit. on p. 5).

[14] *Domain names - implementation and specification*. RFC 1035. 1987-11. DOI: 10.17487/RFC1035. URL: https://www.rfc-editor.org/info/rfc1035 (cit. on p. 5).

[15] T. Duarte. *QuicSand: A Deep Study on the Performance of Different QUIC Implementations*. Accessed: 2025-04-02. 2025. URL: https://github.com/TiagoDuarte25/quicsand-c (cit. on p. 32).

[16] 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. 13).

[17] *File Transfer Protocol*. RFC 959. 1985-10. DOI: 10.17487/RFC0959. URL: https://www.rfc-editor.org/info/rfc959 (cit. on pp. 5, 14).

[18] 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. 5, 12).

[19] 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 p. 29).

[20] C. Halme et al. "Performance analysis of modern QUIC implementations". In: (2021) (cit. on pp. 2, 24).

[21] S. Hertelli, B. Jaeger, and J. Zirngibl. "Comparison of Different QUIC Implementations". In: *Network* 7 (2022) (cit. on p. 20).

[22] IBM Z Performance Team. *KVM Network Performance - Best Practices and Tuning Recommendations*. Last Updated: 2023-11-28. 2018-04. URL: https://www.ibm.com/docs/en/linux-on-systems?topic=kvm-network-performance-best-practices-tuning-recommendations (cit. on p. 47).

[23] *Internet Protocol*. RFC 791. 1981-09. DOI: 10.17487/RFC0791. URL: https://www.rfc-editor.org/info/rfc791 (cit. on p. 5).

[24] 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. 16, 18).

[25] B. Jaeger et al. "QUIC on the Highway: Evaluating Performance on High-rate Links". In: *2023 IFIP Networking Conference (IFIP Networking)*. 2023, pp. 1–9. DOI: 10.23919/IFIPNetworking57963.2023.10186365 (cit. on p. 25).

[26] 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. 2, 24).

[27] 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. 14).

[28] 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. 4).

[29] 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. 4, 6, 8, 12, 25).

[30] J. Legatheaux. *Fundamentos de Redes de Computadores*. 1ª Edição Digital. Nova.FCT Editorial, 2018 (cit. on p. 15).

[31] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[32] Microsoft. *MsQuic*. Accessed: 2025-04-02. 2020. URL: https://github.com/microsoft/msquic (cit. on pp. 36, 50).

[33] NetBeez. *Network Performance Analysis: A Deep Dive*. Accessed: 2025-03-24. URL: https://netbeez.net/blog/network-performance-analysis/ (cit. on p. 47).

[34] 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. 2, 27).

[35] 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. 5, 14).

[36] 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. 5).

[37] 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. 2, 24).

[38] M. Seemann. *QUIC DEEP DIVE*. Paris P2P Festival. 2022. URL: https://www.youtube.com/watch?v=6SyDP7xKqZk (cit. on p. 17).

[39] F. A. V. Sitepu. *Performance Evaluation of Various QUIC Implementations: Performance and Sustainability of QUIC Implementations on the Cloud*. 2022 (cit. on pp. 2, 27).

[40] *TCP 3-Way Handshake | Computer Networks*. https://workat.tech/core-cs/tutorial/tcp-three-way-handshake-in-computer-networks-yoo73319l0lh. [Online; accessed 29-Dec-2023] (cit. on p. 7).

[41] *TCP Connection Termination*. [Online; accessed 12-Fev-2024]. URL: https://www.javatpoint.com/tcp-connection-termination (cit. on p. 10).

[42] 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. 16).

[43] *Time to First Byte (TTFB)*. https://web.dev/articles/ttfb. [Online; accessed 6-Fev-2024] (cit. on p. 27).

[44] *Transmission Control Protocol*. RFC 793. 1981-09. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793 (cit. on pp. 5, 7).

[45] *User Datagram Protocol*. RFC 768. 1980-08. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768 (cit. on p. 5).

[46] *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).

[47] *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. 14).

[48] 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).

[49] *What is network latency? | AWS*. https://aws.amazon.com/what-is/latency/. [Online; accessed 2-Fev-2024] (cit. on p. 26).

[50] *What is network throughput? | AWS*. https://aws.amazon.com/compare/the-difference-between-throughput-and-latency/. [Online; accessed 2-Fev-2024] (cit. on p. 26).

[51] K. Yamamoto. *Implementing QUIC in Haskell*. Accessed: 11-2-2024 (cit. on p. 26).

I

Annex 1: Configuration Files and Topology Examples

I.1 Configuration Files

```
topologies:
  topology1:
    latency: 50
    bandwidth: 500M
    image: quicsand
    number_of_clients: 1
    number_of_servers: 1
```

Listing I.1: Configuration Topologies File

```
configs:
  globalVariables:
    implementation: msquic

tests:
  test1:
    type: request-response
    duration: 30
    data_size: 10
    factor: 100

  test2:
    type: upload
    file_path: /home/user/file.txt

  test3:
    type: download
    file_path: /home/user/file.txt

```

```
    test4:
      type: streaming
      duration: 30
      bitrate: 10
```

Listing I.2: Configuration Tests File

I.2 Topology Example

```
#!/bin/bash

# Create quicsand servers
gone-cli node -- docker run --rm -d \
                            --network gone_net \
                            --ip 10.1.1.1 \
                            --name server1 quicsand

# Create quicsand clients
gone-cli node -- docker run --rm -d \
                            --network gone_net \
                            -v $(pwd)/tmp_result:/result \
                            --name client1 quicsand 10.1.1.1
                                client1

# Create bridges clients
gone-cli bridge bridge-client1

# Create bridges servers
gone-cli bridge bridge-server1

# Create routers
gone-cli router router-left
gone-cli router router-right

# Connect clients to routers
gone-cli connect -w 500M -n client1 bridge-client1
gone-cli connect -w 500M -b bridge-client1 router-left

# Connect servers to routers
gone-cli connect -w 500M -n server1 bridge-server1
gone-cli connect -w 500M -b bridge-server1 router-right

# Connect routers
gone-cli connect -l 50 -w 500M -r router-left router-right

```

```
36      # Propagate routing rules
37      gone-cli propagate router-left
38
39      # Unpause servers
40      gone-cli unpause server1
41
42      # Unpause all nodes
43      gone-cli unpause -a
```

Listing I.3: Simple Topology Example