Chair of Network Architectures and Services
School of Computation, Information, and Technology
Technical University of Munich

Technical University of Munich

School of Computation, Information, and Technology

Informatics

Master's Thesis in Informatics

Analysis of Performance Limitations in QUIC Implementations

Marcel Kempf

Technical University of Munich

School of Computation, Information, and Technology

Informatics

Master's Thesis in Informatics

Analysis of Performance Limitations in QUIC Implementations

Analyse von Leistungseinschränkungen bei QUIC Implementierungen

Author:	Marcel Kempf
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Benedikt Jaeger
	Johannes Zirngibl
Date:	December 15, 2022

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, December 15, 2022

Location, Date	Signature

Abstract

The QUIC transport protocol, standardized in 2021, aims to provide a secure, reliable, and fast connection between two endpoints. Despite all its benefits, previous research has shown that the performance of QUIC is not always as good as expected. The QUIC implementations are often identified as the cause of its suboptimal performance rather than any shortcomings in protocol design. However, QUIC implementations have not yet been analyzed in detail to identify the performance limitations and potential areas for improvement.

In this thesis, we present a configurable measurement framework to analyze the performance of QUIC implementations. Our analysis reveals that the components *packet IO* and *crypto* are the most CPU-intensive, contributing up to 75 % of the total CPU utilization on the server. However, only the sending and receiving of packets turned out to limit performance caused by unused kernel optimizations for more efficient packet processing. The *connection management* component is also identified as a performance-limiting factor. We reveal implementation issues with congestion control algorithms and shallow buffers as causes for performance problems. The BBR implementation of *LSQUIC* achieves less than 40 % of the goodput of CUBIC. In conclusion, our work provides a deeper understanding of the performance limitations of QUIC implementations and offers suggestions for improving their performance in future work. By addressing the issues identified in our study, it may be possible to enhance performance and realize the full potential of QUIC as a secure, reliable, and fast transport protocol.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Transmission Control Protocol (TCP) was first standardized in 1981 and has been in use for more than 40 years. TCP uses sequence numbers, flow control, and congestion control to ensure that data is transmitted without getting reordered, damaged, or lost. This makes TCP a reliable transport protocol and is the reason why it was chosen for most applications on the Internet. The Hypertext Transfer Protocol (HTTP) as the application layer protocol most commonly used for web traffic also relies on TCP until its version 2, which was the latest version until June 2022. TCP has received changes, updates, and new features over the years as the Internet and its requirements for transport protocols have evolved. Some issues with TCP in combination with Transport Layer Security (TLS) for web traffic, such as the costly handshake and the Head-of-Line blocking problem, have not been addressed over the years, as it would affect backward compatibility. [1]

The transport protocol QUIC was standardized by the Internet Engineering Task Force (IETF) in 2021. It was initially designed by Google to provide a solid base for web traffic with HTTP/3, which it finally became with the standardization of HTTP/3 in June 2022 [2]. Google aimed to develop a new transport protocol that is faster and more robust than TCP [3], [4]. QUIC features swift connection establishment, stream multiplexing, congestion as well as flow control, and built-in security with TLS [5]–[7]. QUIC solves many of the drawbacks of TCP such as the costly handshake. New features, such as connection migration, were added to QUIC to meet the latest requirements of a transport protocol. To offer compatibility with existing middleboxes and kernels, QUIC operates in the user space, on top of the User Datagram Protocol (UDP). This also allows for a fast development cycle as QUIC can be updated without touching the kernel.

Related work evaluated the performance of QUIC under different network conditions, partly comparing it with TCP [8]–[11]. It turned out that TCP was able to perform better than QUIC in many cases. The common finding is that poor performance is caused by the implementations rather than the protocol design. Misconfigurations, implementation design choices, and bugs are considered responsible for the observed performance issues. The fact that QUIC is implemented in user space and covers features from transport to application layer gives QUIC implementations a high impact on the overall performance of connections.
However, large parts of the related work analyze QUIC's performance only superficially, comparing different implementations under different network conditions. Additionally, related work was mainly published before the standardization, analyzing preliminary implementations of an IETF QUIC draft. At this point, many QUIC implementations had not yet fully implemented all components.

1.1 Goals

QUIC was developed and introduced as a better alternative to TCP. However, there are results showing that QUIC performs worse than TCP in certain cases.
The goal of this thesis is to analyze the performance of QUIC implementations in detail, focusing on the different components of the implementations.

We want to answer the following research questions:

How do we need to design a measurement framework to analyze the performance of transport protocols?

To analyze the performance of transport protocols, a measurement environment that conducts measurements under identical conditions with different implementations is required. This measurement environment should offer a wide range of capabilities for collecting metrics and comparing them afterward.
We present a configurable measurement setup to execute experiments in a reproducible way while being able to modify the environment slightly and collect useful metrics. This paves the way for standardized approaches to compare different QUIC implementations or other transport protocols using a unified set of performance metrics.

Which parts of QUIC implementations limit performance the most?

QUIC implementations can be divided into several components, as they perform many different tasks. By breaking down QUIC implementations into smaller parts and analyzing them individually, we look for performance limitations from the implementation side. The impact of different components and discovered limitations on the overall performance of QUIC connections is evaluated. This in-depth analysis brings us closer to answering whether QUIC is a viable competitor for TCP.

Which configurations influence the performance of QUIC implementations?

By analyzing bottleneck components, we want to find out which configurations worsen or improve the performance of QUIC implementations.
We perform measurements with a basic TCP/TLS stack to compare them with measurement results from a QUIC implementation. After identifying performance limitations in QUIC implementations, we achieve better performance through optimizations where possible.

1.2 Outline

The following chapter provides basic knowledge about QUIC and performance evaluation of transport protocols. Key aspects of QUIC that are important for subsequent evaluation are briefly introduced. Related work on QUIC performance analysis, comparison of different implementations, and performance evaluation of other protocols are presented in Chapter 3. Chapter 4 describes the network setup and the framework used to execute the measurements. The underlying *QUIC Interop Runner* as well as additional tools used to collect metrics and evaluate the performance are briefly introduced. In Chapter 5, we present, analyze, and discuss the results of the most relevant measurements. After clustering QUIC implementations into components, those of most significant importance for performance are analyzed. Finally, we conclude our results and outline future work in Chapter 6.

Chapter 2

Background

This chapter provides basic knowledge about the QUIC protocol and performance analysis of network protocols. After introducing QUIC's design and the most important features, we shortly describe how QUIC packets are constructed and how acknowledgments are handled. Section 2.2 introduces metrics for measuring network performance and introduces flame graphs as a tool for analyzing the central processing unit (CPU) usage. In Section 2.3, we present the buffers involved in a QUIC connection.

2.1 QUIC

QUIC is a general-purpose transport protocol focusing on fast and secure connections, based on the UDP. It was originally developed by Google and is now standardized by the IETF in RFC 9000 [5]. The motivation was to replace the TCP/TLS stack used by the HTTP with a single protocol that is more efficient and secure [3]. The protocol stacks are compared in Figure 2.1. QUIC uses UDP to achieve backward compatibility with existing network infrastructure. However, it is connection-oriented, reliable, and provides flow and congestion control, just like TCP. QUIC implements those transport features in the user space.

2.1.1 Features

Encryption and Authentication: QUIC uses TLS 1.3 for encryption and authentication [6]. The whole QUIC packet consisting of the header and payload is authenticated. After the handshake has been completed, the payload and large parts of the header are additionally encrypted.

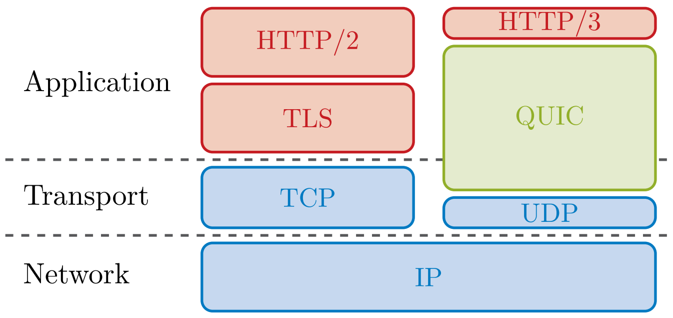

Figure 2.1: QUIC and TCP protocol stacks for HTTP, from [12].

Stream Multiplexing: A stream is a uni- or bidirectional data flow in an existing connection between two endpoints. Multiple QUIC streams can run simultaneously in the same connection and are independent of each other. One primary reason for stream multiplexing with independent streams is to avoid Head-of-Line blocking. QUIC frames of the type `STREAM` are used to open, close, or send data on a stream. Each `STREAM` frame contains a stream identifier, uniquely identifying the stream within a connection by a 62-bit integer. The offset field in `STREAM` frames indicates the position of the data in the stream and is used to reassemble the received data in the correct order.

Congestion and Flow Control: QUIC implements several mechanisms to control the amount of data sent by the endpoints. These mechanisms are inspired by and adapted from TCP. RFC 9002 describes how the mechanisms should be implemented in QUIC [7]. A congestion control algorithm similar to TCP's Reno is described, but other algorithms can be used as well.

User Space Implementation: QUIC is implemented in user space to allow faster development and easy deployment on existing machines. No changes to the kernel are required to install or update QUIC on a machine. A disadvantage is the lack of kernel optimizations and the higher amount of context switches, which can lead to performance issues.

2.1.2 Packets and Frames

QUIC sends packets encapsulated in UDP datagrams. Each QUIC packet consists of a header and payload. Depending on the packet type, the header can either be a short or long header. The long header contains special fields only necessary during connection establishment. After initial connection establishment, the short header is used for all packets.

The payload consists of one or multiple QUIC frames. One frame must always fit into one QUIC packet. There are different types of frames, e.g., `STREAM`, `ACK`, `PING`, or `NEW_CONNECTION_ID`.

2.1.3 Acknowledgments

Acknowledgments in QUIC are sent in `ACK` frames. A QUIC packet is called *ack-eliciting* if it contains at least one frame other than `ACK`, `PADDING`, or `CONNECTION_CLOSE`. While endpoints acknowledge all received packets, `ACK` frames are only sent after receiving an ack-eliciting packet. The `max_ack_delay` transport parameter defines the maximum time the receiver is allowed to wait before sending an `ACK` frame. An `ACK` frame contains so-called ACK ranges, acknowledging multiple packets at once. According to RFC 9000, determining the acknowledgment frequency is a trade-off and may affect the performance of the protocol [5]. The IETF proposed a new approach to determine the acknowledgment frequency in a draft for an extension [13].
The performance influence of different ACK frequencies is analyzed further in Section 5.2.2.

2.2 Performance Analysis

The performance of a network protocol is often defined as the quality of the service it provides. In the case of QUIC implementations, we can either evaluate the performance of the transmission (e.g. throughput) or the performance of the protocol implementation (e.g. CPU usage). The next section describes the different performance metrics and how they can be used to evaluate performance.

2.2.1 Performance Metrics

Several metrics can be used to measure transport protocol performance. If the measurement is performed with a web server and a client requesting a whole website consisting of several files, the page load time (PLT) is often used. It is defined as the time between the first request and the last response. PLT is especially useful to evaluate user experience, as it is a metric that users intuitively gather themselves. If the measurement setup does not include transmissions of large files, requests per second is another interesting metric as it indicates the server's performance under high load. However, if the measurement involves the transmission of large files, the throughput is important, i.e. the amount of data that can be transmitted in a given time. A metric to measure throughput is packets per second (pps). Since packets do not always have the same size, using data rate is a more detailed metric. To not consider retransmissions and the size of data used for connection management, the goodput metric offers a slightly varying definition. The goodput is the size of the file(s) transferred divided by the time it took to transfer the data.

To measure the performance of a single application like the server, the CPU usage is often used. Memory usage can also be measured as a metric for application performance.

2.2.2 Flame Graphs

To find bottlenecks or performance issues in an implementation of a server or client, a flame graph can be used. A flame graph visualizes the CPU usage in the form of stack traces in a captured time interval. The graph can be created using a capture that contains the call stack of the functions executed by the CPU at different timestamps.

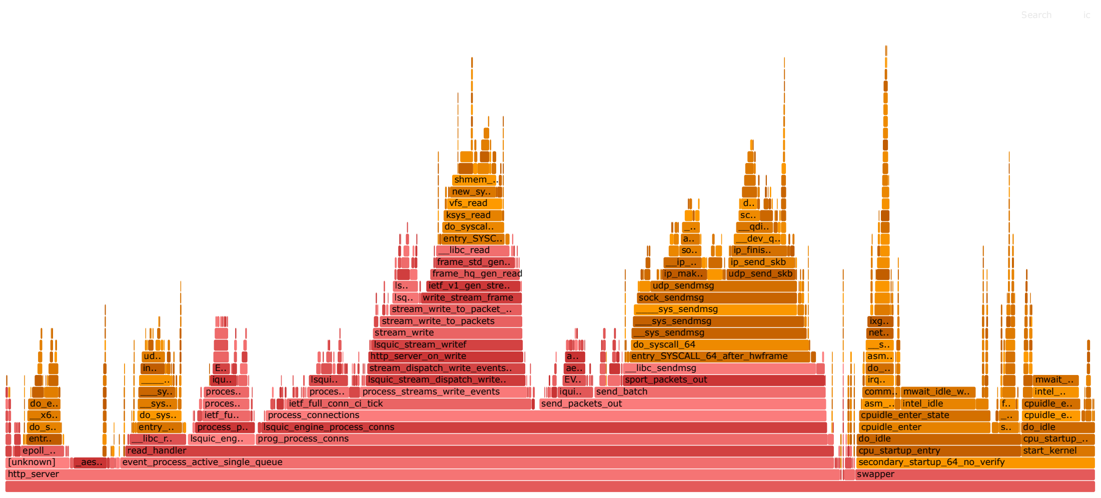

FIGURE 2.2: A sample flame graph of a *LSQUIC* server transferring a 10 GiB file in 100 s.

A sample flame graph is shown in Figure 2.2. The data used to create this flame graph was collected during a transmission of a 10 GiB file with the QUIC implementation *LSQUIC* on the server. While the x-axis of a flame graph represents the stack profile population, the y-axis represents the stack depth. The width of a bar, therefore, represents the amount of time spent in this specific function (and all functions above). The color of a bar usually has no specific meaning. We decided to use shades of orange for all functions running in the kernel and shades of red for all functions running in the user space.

2.3 Buffers

During the transmission of data, packets are not always processed immediately. Several buffers at different positions are involved.

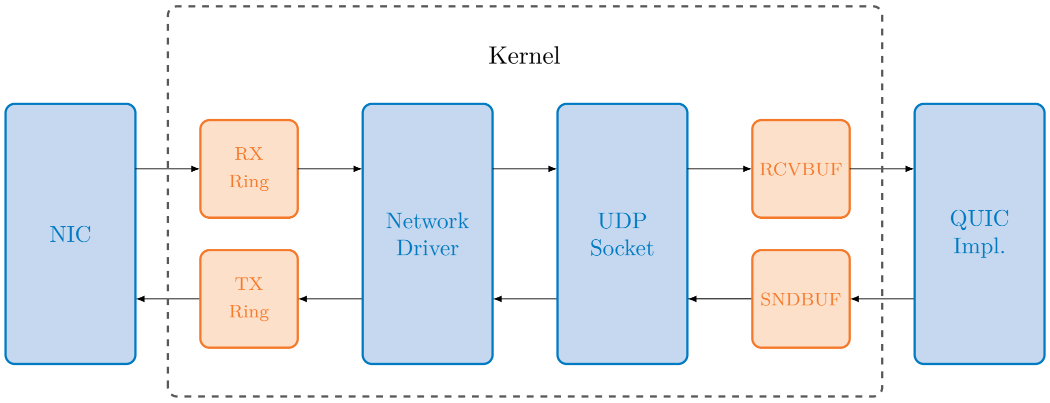

Figure 2.3: Important buffers involved in sending and receiving with UDP sockets. This illustration is simplified, as NIC internal buffers and the qdisc layers are excluded.

Figure 2.3 shows the buffers involved in the transmission of data. While the upper arrows show the flow of incoming packets, the lower arrows show the flow of outgoing packets. For simplification, not all steps and buffers involved are shown in the illustration.
After initial processing, the network interface card (NIC) uses Direct Memory Access (DMA) to store received packets in the RX ring buffer. DMA is a technique to access the system memory without the involvement of the CPU. The kernel will now use interrupts to fetch the received packets from the RX ring buffer. The packets will be processed by the kernel and the headers of layers 2, 3, and 4 will be removed. The layer 4 payload is now stored in the socket receive buffer (RCVBUF). The QUIC implementation, reading data from the socket, will remove and further process the layer 4 payload from the socket receive buffer. If the RX ring buffer or the socket receive buffer are full, incoming packets are dropped. [14], [15]
The sending process involves a socket send buffer (SNDBUF) and a TX ring buffer. The buffers and the involved parts of the transmission act similarly to the receiving process. One important difference is that no data is dropped if the socket send buffer is full. In this case, the system call will block or fail until the buffer is not full anymore. By choosing the size of the socket send buffer accordingly to the available bandwidth and size of the later buffers involved in sending, the amount of dropped packets during the sending process is usually low or even zero. [14], [15]
The impact of buffer size on performance, including the effects of both undersized and oversized buffers, is evaluated in Section 5.4.3.

Chapter 3

Related Work

This chapter provides an overview of related work in the field of QUIC and network performance analysis.

Yang et al. [16] examined different QUIC implementations and analyzed the performance of different components. The implementations *quant*, *quicly*, *picoquic*, and *mvfst* were used. Their testbed setup consisted of two servers connected with two 10 Gbit/s links. While Server 1 was running the QUIC server and client, Server 2 was running a network simulator to simulate different network conditions, e.g. packet loss, latency, and reordering. Traffic is routed from Server 1 to Server 2 via the first link and back to Server 1 via the second link. It was explored in detail how network conditions like packet loss and reordering affect QUIC performance. The performance was measured by looking at throughput and CPU time. They found that the main bottlenecks are the kernel network stack, the crypto component of the respective QUIC implementation, and packet reordering.

A paper by Rochet et al. [9] explores the benefits of tighter coupling of TCP and TLS when used together. They designed and implemented their own protocol called *TCPLS*, which is based on *picotls* and uses TLS 1.3. Transport features comparable to QUIC, such as streams, connection migration, and HOL blocking avoidance, have been implemented through TLS Encrypted Extensions. TCP Fast Open [17] allows a quick handshake to be achieved in many connections. TCPLS was able to reach a throughput twice that of *quicly*, the fastest of the QUIC implementations tested. In their measurements, throughput and packets per second (pps) were used as metrics. The QUIC implementations *mvfst* and *MsQuic* were also evaluated in addition to *quicly*. The authors found that it is possible to add modern transport features to TCP without also requiring any changes to the kernel. They claim that TCPLS is a viable alternative to QUIC, as it performs substantially better than QUIC in their measurements.

Yu et al. [8] conducted a performance analysis of different QUIC production endpoints hosted by Google, Facebook, and Cloudflare. Their measurements included the transfer of entire websites from the production endpoints to their client running different QUIC implementations. They consider the analysis of QUIC performance within testbeds to be unrepresentative since the kernels are not optimized for QUIC in such cases. The metrics used are the bytes acknowledged over time, as well as the page load time (PLT). The introduction of additional packet loss and delay gave them extra insight, as they were unable to change the server configurations. They concluded that performance differences are mainly caused by differences in the implementations or server configurations, not by inherent properties of QUIC itself.

A blog post by Marc Richards [18] compares the Linux kernel network stack with a kernel bypass network stack using Data Plane Development Kit (DPDK), a library to achieve faster packet processing. He used TCP only; QUIC or UDP were not analyzed. Several improvements were made to the Linux Kernel's network stack to achieve better performance. Performance was measured in requests per second. The results show that the kernel-bypass network stack outperformed the unoptimized kernel network stack by a factor of 4.2 and the optimized kernel network stack by a factor of 1.5. He found several advantages and disadvantages to using both the optimized kernel and the kernel-bypass network stack.

A paper by Mishra et al. [10] evaluates and compares congestion control of different QUIC implementations. The QUIC implementations *mvfst*, *chromium*, *MsQuic*, and *quiche* were used. They implemented a tool called *QUICbench* to perform their measurements. They used the kernel implementation of the different congestion control algorithms as a reference. Their testbed setup consisted of two servers connected with a 1 Gbit/s link. Significant performance differences were found between the different congestion control algorithms in QUIC implementations and the reference implementations. The authors concluded that the performance is mainly affected by implementation differences, not by differences between QUIC and TCP. They also planned to evaluate more QUIC implementations in the future.

Chapter 4

Implementation

This chapter presents the used measurement setup including the testbed, the measurement framework, and the tools used to collect metrics. Additionally, the procedure of result parsing and postprocessing of the collected logfiles is described.

4.1 Testbed Setup

All measurements were performed in the chair's testbed on bare metal machines. We decided not to use any kind of virtualization, containerization, or emulation to avoid any overhead introduced by the additional layers. All measurements were performed on the same network topology, consisting of one client, one server, and a management node, as illustrated in Figure 4.1. The client and server are connected via a 10 Gbit/s link. The management node is connected to client and server with a link of a separate network card, allowing the management node to fully control client and server. The management node uses the Plain Orchestrating Service (POS) [19] to control other nodes. POS can execute arbitrary commands on other nodes, reboot them, and change the live operating system.

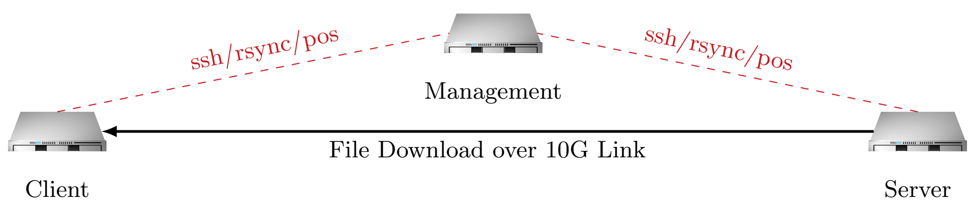

Figure 4.1: Network topology of our testbed setup.

Both machines are running Debian 11 as a live system not involving any hard disks. If not stated otherwise, client and server were equipped with an Intel® Xeon® E5-1650 v3 CPU and an Intel® 10G X550T network card.

4.2 QUIC Implementations

We considered different open-source QUIC implementations listed by the IETF QUIC Working Group [20]. All implementations that are not actively maintained or do not support the current version of the QUIC protocol were excluded from our choice. As we are interested in performance measurements, we decided to mainly use *LSQUIC*, which is implemented in C and focuses on performance. More advantages of using *LSQUIC* are the presence of example implementations for a HTTP client and server and the continuous development of the library. Support for extensions such as the delayed ACKs extension [13] and the datagram extension [21] was added soon after the publication of the corresponding drafts. Zirngibl et al. [12] showed that *LSQUIC* is also among the most popular QUIC implementations deployed in May 2021.
To ensure consistency, we use the same version of *LSQUIC* for all our measurements. The lsquic commit hash `f3657f1` was chosen, as it was the latest at the time this decision was made.

4.3 QUIC Interop Runner

To execute measurements and collect data, we used the *QUIC Interop Runner*, a framework initially developed to run interoperability tests between QUIC implementations [22]. The *QUIC Interop Runner* supports multiple test cases, checking the tested implementation for compliance of functionalities in interoperation, and measurements, measuring the performance of different server-client pairs.
The original *QUIC Interop Runner* uses *Docker*, a software platform for OS-level virtualization. Server and client are run in *Docker* containers on the same machine. We use a modified version of the *QUIC Interop Runner* operating both endpoints on dedicated hosts. The modified *QUIC Interop Runner* uses `poslib` to control the testbed, transfer files, and execute commands on the testbed hosts [19]. We mainly use the provided measurement called `goodput`, in which the client requests a large file from the server and measures the time it takes to download the file. As a metric for performance, we primarily use goodput. More metrics are collected with the tools described in the following section.
Using the *QUIC Interop Runner* has the advantage that our measurements can easily be reproduced, also with other QUIC implementations.

Listing 4.1: Sample `config.yml` for the *QUIC Interop Runner*

```yaml
testbed: testbed/testbed_uniswap-solana.json
server: lsquic
client: lsquic
test: goodput
repetitions: 20
filesize: 8192
implementation_directory: ./out
client_prerunscript:
  - pre-post-scripts/start-pidstat.sh
  - pre-post-scripts/run-netstat.sh
  - pre-post-scripts/set-rcvbuf.sh
client_postrunscript:
  - pre-post-scripts/stop-pidstat.sh
  - pre-post-scripts/run-netstat.sh
  - pre-post-scripts/set-buffers-default.sh
server_prerunscript:
  - pre-post-scripts/start-pidstat.sh
  - pre-post-scripts/run-netstat.sh
server_postrunscript:
  - pre-post-scripts/stop-pidstat.sh
  - pre-post-scripts/run-netstat.sh
client_implementation_params:
  - CC_ALGO=1
  - rmem_value=1703936
server_implementation_params:
  - CC_ALGO=1
```

We extended the *QUIC Interop Runner* at several points to improve its functionality for performance measurements. First, we added support for running `perf` (explained in the following section) on the server and client. Minor changes were made to support modified versions from implementations, as we use multiple modified versions of *LSQUIC* for our measurements in Sections 5.3 and 5.5. We also added the functionality to pass variables to the client or server. Additional tools like `ethtool` and `netstat` to collect more metrics were integrated and are available for optional use.
To make the configuration of the *QUIC Interop Runner* more flexible and measurements reproducible, we added support for `yaml` configuration files. A sample configuration file is shown in Listing 4.1.

4.4 Tools

In this section, we briefly introduce tools used to capture packets, metrics, and statistics during measurements. Tools to modify the measurement conditions are also described.

`Ifstat` is a tool to monitor network traffic and report network interface activity. We use it to log current bandwidth usage of the client's network interface in both directions every second.

`Perf` is a Linux profiler, also known as perf_events. It is event-oriented and can capture performance counters, tracepoints, and hardware events [23]. The perf command features various subcommands, e.g. `perf record` to record a profile, `perf report` to display a recorded profile, `perf script` to display the trace output of a captured profile, and `perf stat` to collect performance counter statistics [24].
We use the `perf record` and `perf script` subcommands for this thesis.

`Pidstat` is a command to monitor running tasks in the Linux kernel [25]. While `pidstat` supports various statistics, we use it to monitor the CPU utilization of a process.

`Tcpdump` is a network packet analyzer. We use it to capture packets on the network interface used for the measurement at our client and save them in a packet capture (PCAP) file. Together with the encryption keys, this allows us to decrypt the captured packets for further analysis.

`Ethtool` is a tool to query network interface statistics and control network driver settings. The number of packets received, transmitted, and dropped is included in the statistics [26]. Detailed information on queues is also provided. The amount of dropped packets includes only the ones dropped at the ring buffers shown in Figure 2.3. We use `ethtool` to log the detailed statistics of both used network interfaces before and after transmission.

`Netstat` is another tool for collecting and displaying network statistics. In comparison to `ethtool`, `netstat` provides statistics from the kernel instead of the network interface [27]. It also provides detailed information on the UDP and TCP connections, which `ethtool` and `ifstat` do not support. We use `netstat` to count the number of packet drops in the kernel's UDP send and receive buffers.

`tc` is a tool provided by the Linux kernel for managing traffic control settings. It can shape or schedule outgoing traffic. `tc` supports the use of various different queueing disciplines (qdiscs) to manage the traffic [28]. `NetEm` is used in some measurements to emulate the characteristics of a wide area network (WAN) to outgoing packets. `NetEm` is capable of emulating packet loss, delay, reordering, or limited bandwidth [29]. While we use `NetEm` to emulate delay, `tbf` is used to emulate limited bandwidth as it scales better with larger bandwidth limits [30].

4.5 Analysis

To further analyze the data from measurements, we developed a collection of tools that refactors the output of the tools we use to collect metrics.
Perf is started before the transmission begins and stopped after the transmission has been completed, as we do not attach perf to the specific Process Identifier (PID) of the server or client. Therefore, we trim the perf output to result in a shorter capture, reaching from the first to the last occurrence of the server or client. We perform this step to achieve more precise information about the CPU runtime of other processes like ssh. After this step, we use a pre-made script to collapse the perf output [31]. This step is necessary to achieve a more readable output and to create flame graphs. The script converts a list of multiline stacks to one-line semicolon-separated stacks followed by the sum of all samples of every stack. From this output, we now generate a flame graph.
We use another script to read the output of the *QUIC Interop Runner* (stored in `result.json`) and the output of all tools used to collect metrics during the measurement. The script was initially developed by Kevin Ploch [32] and was extended by us to support more tools and metrics. The script performs postprocessing steps to the output files of the different tools and stores all data in a `csv` file, each line representing a repetition of a measurement.

One essential part we implemented in the parsing script is the categorization of call stacks into different components of QUIC implementations. To get a broad overview of the significance concerning the CPU time of different components, we map each line of the collapsed stack output from before to a category like `CRYPTO` or `PACKET_IO`. In selecting the different categories, we have followed the categories used by Yang et al. [16] in their CPU usage breakdown. We decided to add one more category, `GENERAL`, to cover all functions that do not fit into the other categories. All functions considered negligible due to their small contribution to the overall CPU usage are assigned to this category as well. We also decided to merge the category covering the connection setup and teardown and the category covering acknowledgments into one category, `CONN_MGMT`. Since we wanted to separate the assembly and disassembly of packets, we have added the new category `MARSHALLING`. This results in the following categories: `PACKET_IO`, `IO`, `CONN_MGMT`, `CRYPTO`, `MARSHALLING`, and `GENERAL`. Table 4.1 shows how our categorization works for a perf output recorded on an *LSQUIC* server. The first data row (`SUM_SAMPLES`) shows the total number of samples in the perf output having the *LSQUIC* server application in the call stack. The second section provides a brief overview of the different categories and their CPU time. In the third section, the individual components are broken down further to simplify the search for bottlenecks.

Table 4.1: CPU utilization of an *LSQUIC* server broken down to different parts and functions. The samples were collected on the *LSQUIC* server transferring a 8 GiB file to the client. The measurement was performed 20 times and the samples were summed up.

	Category	Samples	Samples Share
	SUM_SAMPLES	265 911	100.00 %
Distribution	PACKET_IO	160 368	60.31 %
	CRYPTO	46 789	17.60 %
	IO	27 358	10.29 %
	CONN_MGMT	23 731	8.92 %
	MARSHALLING	3778	1.42 %
	GENERAL	3887	1.46 %
Detailed Distribution	PACKET_IO (sendmsg)	101 674	38.24 %
	PACKET_IO (recvmsg)	1936	0.73 %
	PACKET_IO (asm_common_interrupt)	20 177	7.59 %
	PACKET_IO (read from stream)	122	0.05 %
	PACKET_IO (write to stream)	24 704	9.29 %
	PACKET_IO (function: send_packets_out)	5695	2.14 %
	PACKET_IO (IP find next hop)	5877	2.21 %
	PACKET_IO (read one packet)	183	0.07 %
	CRYPTO (AES)	26 554	9.99 %
	CRYPTO (AEAD)	12 882	4.84 %
	CRYPTO (encrypt packet)	4968	1.87 %
	CRYPTO (decrypt packet)	169	0.06 %
	CRYPTO (apply header protection)	1966	0.74 %
	CRYPTO (strip header protection)	168	0.06 %
	CRYPTO (RSA)	60	0.02 %
	CRYPTO (X25519)	22	0.01 %
	IO (read)	26 459	9.95 %
	IO (write (ksys))	2	0.00 %
	IO (write (fs))	30	0.01 %
	IO (epoll)	783	0.29 %
	IO (other)	84	0.03 %
	CONN_MGMT (process incoming ack)	7719	2.90 %
	CONN_MGMT (other)	16 012	6.02 %
	MARSHALLING (process packet)	248	0.09 %
	MARSHALLING (parse packet)	368	0.14 %
	MARSHALLING (build ack)	105	0.04 %
	MARSHALLING (function: sport_packets_out)	3057	1.15 %
	GENERAL (process incoming packet)	711	0.27 %
	GENERAL (read_handler)	2153	0.81 %
	GENERAL (other)	1023	0.38 %

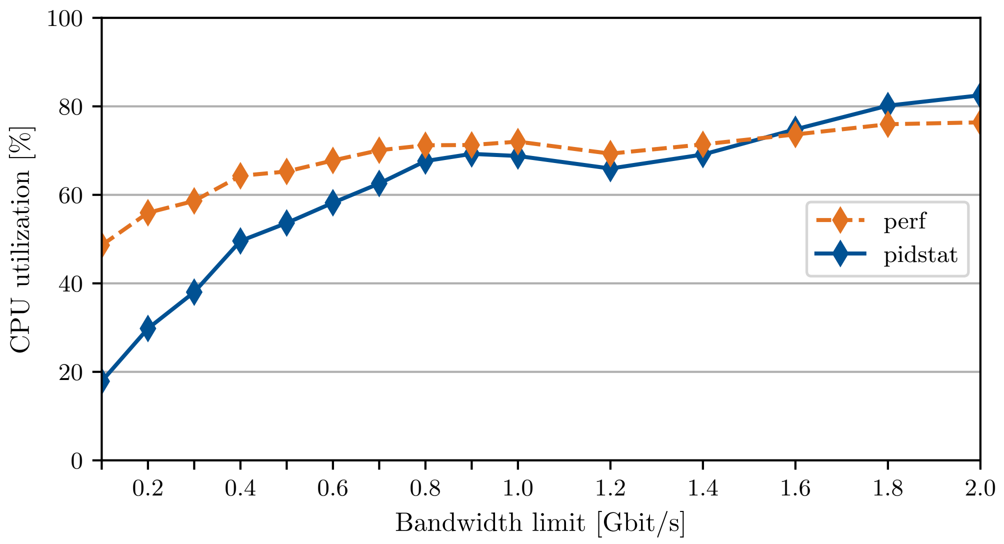

Figure 4.2: CPU utilization of an *LSQUIC* server determined with `perf` and `pidstat` compared for different bandwidth limits.

To measure the CPU utilization of the complete server and client applications, we tested two different methods. The first method employes the `pidstat` tool to sample the CPU utilization of a running process and record it in a file every second during runtime. Afterward, the average of all samples is calculated to determine the average CPU utilization of the process. The second method utilizes the trimmed output of the `perf` tool also used for the CPU usage breakdown. To calculate the average CPU utilization, the number of samples including the server or client application in the call stack is divided by the total number of samples.

Figure 4.2 presents a comparison of the results obtained using both methods for different bandwidth limits. While both methods produce comparable and plausible results, the method using `pidstat` yields slightly lower values for smaller bandwidths. We observed that with an increasing CPU utilization value, the deviation between the two methods decreases. We assume that the bandwidth limit does not affect the deviation between our two methods, but the CPU utilization value itself.
As no ground truth is available for the CPU utilization, we are unable to tell if the method using `pidstat` underestimates the CPU utilization for smaller values or if the method using `perf` overestimates the CPU utilization for larger values. We decided to use the method using `pidstat` to determine the CPU utilization for further evaluation. The main reason is that `pidstat` has significantly less impact on the system during measurements than `perf`.

Chapter 5

Evaluation

Using the measurement framework described in detail in Chapter 4, we performed measurements with different configurations. We modified the *LSQUIC* server and client applications to change between different cipher suites or congestion control algorithms, for example. In the following sections, we will present the results and look closely at different parts of QUIC implementations and general performance issues. If not stated otherwise, all measurements were performed by transmitting a single 8 GiB file from the server to the client. The bandwidth or delay of the network was not limited for most measurements. For every measurement, 20 repetitions were performed and the average was calculated.
We performed a baseline measurement with the previously described setup and without changing anything from the default *LSQUIC* settings. We achieved a goodput of 1715 ± 908 Mbit/s. The CPU utilization of the client was at 75 ± 9 %, while the server was at 77 ± 22 %. Especially the standard deviation of the server's CPU utilization seems high, which we want to find possible reasons for in the following sections. The standard deviation of the measured goodput is likewise significantly higher than expected.

5.1 Components

To get a better understanding of the performance of QUIC implementations, we use `perf` and our categorization tool described in Chapter 4 to break the CPU usage down into different categories. An explanation of the decision on the categories used is given in Section 4.5. In Table 4.1, we provide a detailed overview of the composition of each category.

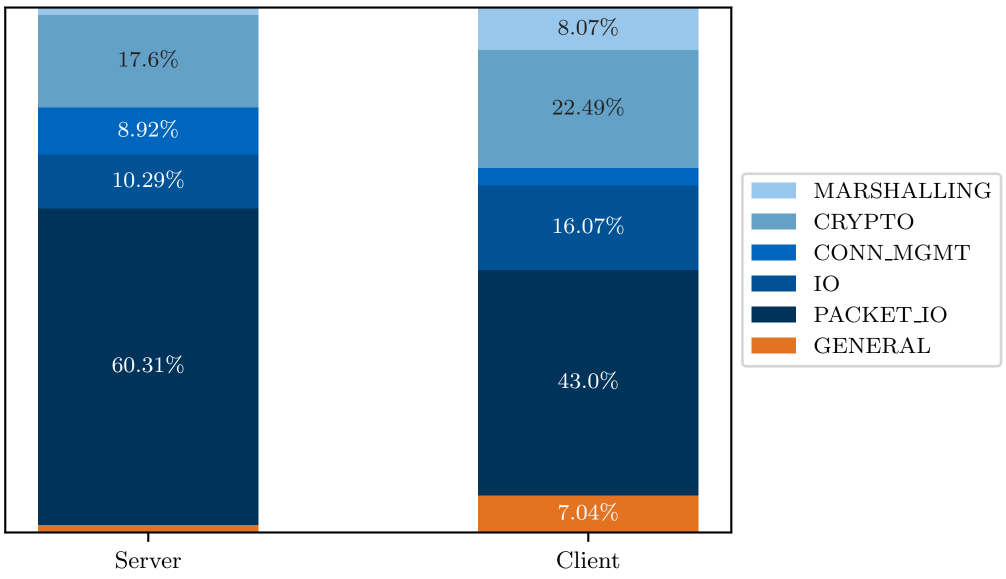

Figure 5.1: CPU usage of client and server categorized into the different components.

We use the following categories:

- `PACKET_IO`: send and receive on socket, stream read/write, receive interrupts
- `IO`: read and write operations on the filesystem
- `CONN_MGMT`: send control, acknowledgment processing
- `CRYPTO`: cryptographic operations, e.g., encryption and decryption
- `MARSHALLING`: marshalling and unmarshalling of packets
- `GENERAL`: all remaining function calls that do not fit into the other categories

Figure 5.1 shows the categorized CPU usage of *LSQUIC*'s server and client. The percentages of the different components are relative to the total CPU utilization of the *LSQUIC* binary at the corresponding endpoint. The height of both bars equals 100 % of the CPU usage. We observed that `PACKET_IO` is the most time-consuming component for both *LSQUIC*'s server and client, contributing more than 40 % of the total CPU time on the client and more than 60 % on the server. The second most time-consuming component is `CRYPTO`, followed by `IO`.
Since QUIC performance is often degraded by the CPU (as shown in the following experiments), the components `PACKET_IO` and `CRYPTO` are particularly important to focus on.

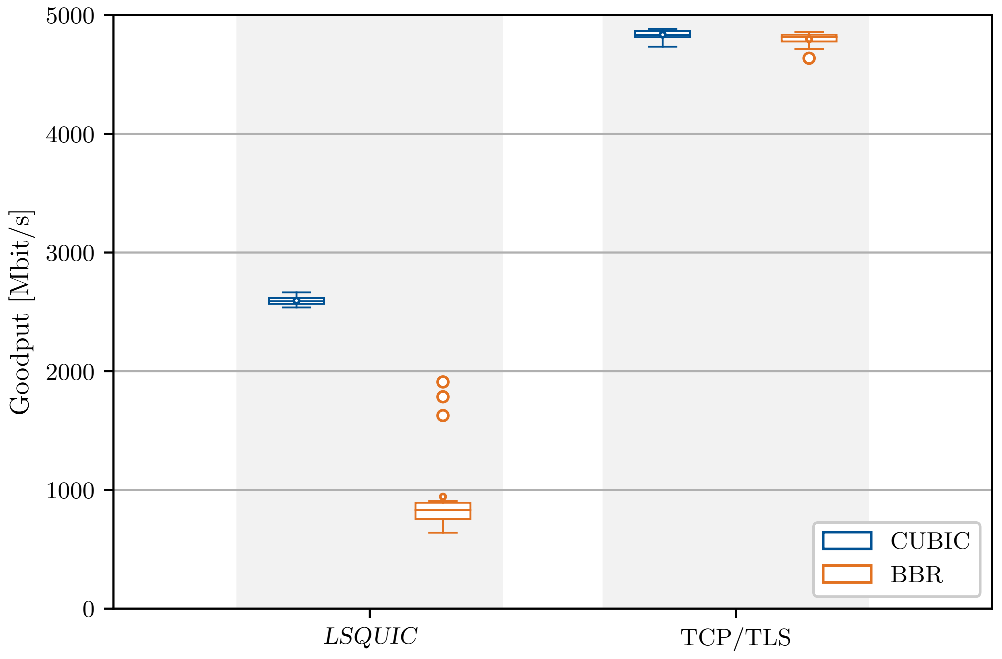

Figure 5.2: CUBIC and BBR goodput comparison for *LSQUIC* and TCP/TLS stack using `nginx` and `wget`. All four measurements were performed with 20 repetitions.

5.2 Connection Management

Although the connection management is not CPU intensive, it can heavily influence the performance of a QUIC implementation by controlling the sending rate of the endpoints. In this section, we will look at the influence on the performance of different congestion control algorithms and acknowledgment frequencies.

5.2.1 Congestion Control

Since UDP has no congestion control, QUIC implementations provide congestion control algorithms in the user space. Most QUIC implementations use existing congestion control algorithms from TCP, as they are predictable and tested for many years [10]. *LSQUIC* provides support for the loss-based algorithm CUBIC, the model-based algorithm Bottleneck Bandwidth and Round-trip propagation time (BBR), and an adaptive algorithm that selects the best algorithm based on the round trip time (RTT). By default, the adaptive algorithm is used, which selects BBR if the RTT is greater than 1.5 ms or CUBIC otherwise.

In the left part of Figure 5.2, the goodput of transmissions with different congestion control algorithms implemented in lsquic is compared. We performed 20 repetitions for

every configuration.
Each box reaches from the first to the third quartile, while the median is marked by a line. The mean average is marked by a dot. The upper whisker extends from the upper edge of the box to the highest datum below 1.5 times the inter-quartile range (IQR). The lower whisker is positioned similarly below the box. Outliers are marked with a circle. These settings apply to all following box plots.
It can be seen that the CUBIC implementation in *LSQUIC* does not only achieve 2.5 times the goodput of BBR but also produces less deviation between the measurements. The BBR implementation of *LSQUIC* can also achieve a goodput close to 2 Gbit/s as the outliers show. However, the average goodput of almost 1 Gbit/s achieved with BBR is significantly lower than the 2.6 Gbit/s achieved with CUBIC.
When using the adaptive congestion control algorithm, the RTT estimation after connection establishment decides on the congestion control algorithm used. As the RTT estimation of *LSQUIC* in our setup was always around the threshold of 1.5 ms, the adaptive algorithm selected BBR for most of the measurements but sometimes CUBIC. We identified this as the reason for the high deviation in goodput and CPU utilization in our baseline measurement.

As the goodput difference between both congestion control algorithms is greater than 1.5 Gbit/s in average, we also compared the performance of the CUBIC and BBR implementations in *LSQUIC* with the respective implementations in TCP combined with TLS for encryption. Our TCP/TLS stack uses `nginx` as server and `wget` as client application. The results are shown in the right part of Figure 5.2. It can be seen that the performance difference between the CUBIC and BBR implementations in our TCP/TLS stack is much smaller than in *LSQUIC*. The goodput difference of 40 Mbit/s is smaller than the standard deviation of both measurements. We can therefore assume that the TCP implementations of CUBIC and BBR achieve a similar goodput in our measurement setup. We expect the BBR implementation of *LSQUIC* to have issues in the implementation causing the sending rate to be lower than it could be. Mishra et al. [10] also found that other QUIC implementations have significant performance differences in their implementations of congestion control algorithms.
Congestion control algorithms, especially BBR, heavily rely on the RTT for calculating their sending rate. We supposed that the RTT of 1.5 ms in our measurement setup might be too low for the BBR implementation to work properly. We decided to increase the RTT by 10 ms using NetEm to emulate a delay on outgoing packets. Since asymmetric networks can lead to complications, we have added a 5 ms delay for outbound packets on both the client and the server. We used the following command to apply the delay:

Listing 5.1: Command used to add delay

```
tc qdisc add dev enp2s0f0 root netem delay 5ms limit 100000
```

Table 5.1: Goodput and packet drops for different congestion control algorithms with variable client receive buffer size and RTT. All metrics are average values of 5 repetitions rounded to integers.

Configuration			Results	
CCA	Delay[a] [ms]	BUF[b] [KiB]	Goodput [Mbit/s]	Drops[c]
BBR	0	208	848	50
BBR	0	1664	904	0
BBR	5	208	643	2 140 772
BBR	5	1664	1798	173
CUBIC	0	208	2621	12 657
CUBIC	0	1664	2770	622
CUBIC	5	208	521	2501
CUBIC	5	1664	1977	1179

[a] Delay for outgoing packets added at server and client, [b] Size of the client's UDP receive buffer
[c] number of packets dropped by the client's kernel due to insufficient receive buffer size

The default value for `limit` is 1000, meaning that a maximum of 1000 packets will be held back in the queue waiting to get sent [29]. In our first test with delay emulation, we observed more retransmissions than expected. We calculated the minimal NetEm buffer size for our network using the following equation:

$$\begin{aligned}\text{minimum limit} &= \frac{\text{delay} \cdot \text{available bandwidth}}{\text{packet size}}\\ &= \frac{5\,\text{ms} \cdot 10\,\text{Gbit/s}}{1500\,\text{B}}\\ &= \frac{0.005\,\text{s} \cdot 1\,250\,000\,\text{kB/s}}{1.5\,\text{kB}}\\ &= 4166,\overline{6}\end{aligned}$$

To be on the safe side, we used a limit of 100 000 packets, as we performed measurements also with more delay and wanted to avoid packet loss caused by delay emulation.

Table 5.1 shows the results of our measurements with delay emulation. Scholz et al. [33] showed that many retransmissions happen with BBR in combination with shallow buffers. This is why we decided to perform additional measurements with an increased receive buffer at the client. The performance influence of buffers is evaluated in more

detail in Section 5.4.3.
The results confirm our assumption that the RTT of 1.5 ms is too low for BBR to work properly. The calculation of the bandwidth delay product (BDP) and the sending rate seem to be influenced. The resulting sending rate is low enough to avoid packet drops at the client's receive buffer. Increasing the client's receive buffer does not significantly increase goodput with BBR, but adding a 5 ms delay to outbound packets on server and client leads to a doubled goodput of 1798 Mbit/s in average. If a delay of 5 ms is added at server and client and the client's receive buffer has the default size of 208 KiB, more than two million packets are dropped. The resulting retransmission rate of 27 % on average confirms the findings of Scholz et al. [33].

After reviewing these results, we decided to prefer the CUBIC implementation in future *LSQUIC* measurements over the BBR implementation, as it performs better and is more stable under our default conditions. As the CPU utilization at the server is at 98.7 % on average, we assume that the server's CPU is the bottleneck preventing CUBIC to achieve higher goodput.

5.2.2 Acknowledgments

As already mentioned in RFC 9000 Section 13.2.2 [5], the acknowledgment frequency is an important factor for the performance of QUIC. The acknowledgment frequency is defined as the ratio of the number of acknowledgments sent by the client to the number of packets received by the client. Sending delayed or fewer acknowledgments can negatively affect performance since the other endpoint needs them to adjust the congestion window. Sending acknowledgments too often can also negatively affect performance since the packet transmission and processing cost of acknowledgments at both endpoints increases [5]. Jana Iyengar, a main author of RFC 9000, also found that the performance of QUIC is significantly decreased by the CPU costs caused by a higher acknowledgment frequency. This was identified to be a problem, especially in high bandwidth networks, where the CPU costs of acknowledgments are higher than the benefits of a higher acknowledgment frequency [34]. There are several papers analyzing the influence of different acknowledgment frequencies and strategies for QUIC or transport protocols in general [35]–[37]. They propose lowering or adjusting the acknowledgment frequency to achieve better performance in many cases. In this section, we will look at the influence of acknowledgment frequency on the performance of QUIC.

As it is not possible to change the acknowledgment frequency in *LSQUIC* without fundamentally modifying the source code, we looked at the already present deviation of the acknowledgment frequency in *LSQUIC*. We measured the acknowledgment frequency by counting the number of packets and bytes sent by the client. To ensure the correctness of this method, we used the `pcap` files and the TLS encryption keys from our measurements to decrypt the packets. It showed that more than 99.99 % of the packets sent by the client after the initial handshake contained `ACK` frames. The remaining packets were observed especially earlier in connections and contained just a padded `PING` frame. We can therefore make the simplifying assumption that all packets the client sends are acknowledgments.

By default, *LSQUIC* uses the delayed ACKs extension [13] that is an IETF draft as of December 7, 2022. This extension adds the functionality for a QUIC client or server to control the acknowledgment frequency of its peer. The `ACK_FREQUENCY` frame and the `IMMEDIATE_ACK` frames are used to control the acknowledgment policy of the peer.
As *LSQUIC* provides a way to disable the delayed ACKs extension, we compared the performance of *LSQUIC* with and without the extension enabled. As we observed a significant difference between the acknowledgment frequencies of *LSQUIC*'s CUBIC and BBR implementations, we also compared the performance of both implementations with and without the extension enabled. For both CUBIC and BBR, the goodput and the number of packets sent by the client did not change more than by one standard deviation. The delayed ACKs extension therefore does not seem to influence the performance of *LSQUIC* in our measurement setup. We ensured that the delayed ACKs extension was working by looking at the logs.

After reviewing results from other measurements and looking at the number of packets sent by the client, we noticed that the acknowledgment frequency is mainly influenced by the congestion control algorithm used and the network conditions. For the measurements providing data for Table 5.1, we observed acknowledgment frequencies from $\frac{1}{6}$ to $\frac{1}{250}$. The client in our measurements with BBR and without delay added sends the most packets, acknowledging every sixth packet on average. For the measurements with BBR and delay added, the acknowledgment frequency decreases to approximately $\frac{1}{200}$. Also when using CUBIC, the acknowledgment frequency changes drastically between our measurements, as can be observed in Figure 5.5. In the underlying measurements, the client received around six million packets from the server, resulting in an acknowledgment frequency between $\frac{1}{40}$ and $\frac{1}{250}$.

The calculated values from our measurements show that the acknowledgment frequency fluctuates significantly between different measurements. The acknowledgment frequency is mainly influenced by the RTT calculated by the endpoints. We were able to observe a correlation between the acknowledgment frequency and the achieved goodput. Fewer acknowledgments were usually linked to a higher goodput. The only measurement that did not follow this pattern was the measurement with BBR and a delay of 5 ms added at server and client. The reason is probably the unusually high retransmission rate of 27 %

Table 5.2: TLS 1.3 cipher suites supported by QUIC and *BoringSSL* [38].

IANA ID	Cipher Suite	QUIC	BoringSSL
`0x1301`	`TLS_AES_128_GCM_SHA256`	✓	✓
`0x1302`	`TLS_AES_256_GCM_SHA384`	✓	✓
`0x1303`	`TLS_CHACHA20_POLY1305_SHA256`	✓	✓
`0x1304`	`TLS_AES_128_CCM_SHA256`	✓	✗
`0x1305`	`TLS_AES_128_CCM_8_SHA256`	✗	✗

in this measurement. It can not be determined if a higher acknowledgment frequency leads to a higher goodput, if fewer acknowledgments are sent when a high goodput is achieved, or if both are the case. A deeper analysis of the *LSQUIC* source code showed that besides the smoothed round trip time (SRTT) and the `max_ack_delay` (explained in Chapter 2), no additional factors are seeming considered when queueing an outgoing acknowledgment.
As the receiving of packets is comparatively expensive for the server, the delayed ACKs extension has the potential of increasing performance. Especially for high-bandwidth networks, the acknowledgment frequency should be decreased, as the extension provides support for this. Depending on the network conditions, the acknowledgment frequency should be adjusted so that the trade-off between the CPU costs of sending and receiving acknowledgments and the benefits for connection management is optimized.

5.3 Crypto

QUIC uses TLS for encryption and authentication. The oldest TLS version that should be supported by QUIC implementations is the current version, TLS 1.3 [6]. In this section, we will take a closer look at the general impact of crypto on performance as well as the performance difference between cipher suites.
As QUIC implementations must not use a TLS version older than TLS 1.3, we only consider cipher suites supported by TLS 1.3. The last of the TLS 1.3 cipher suites listed in Table 5.2 is incompatible with QUIC. *BoringSSL*, the crypto library used by *LSQUIC*, supports only the first three cipher suites.

The first two cipher suites differ slightly in the parameters used for encryption and hashing. The third cipher suite uses the ChaCha 20 stream cipher and the Poly1305 message authentication code. In performance measurements, ChaCha20-Poly1305 is usually faster than AES-GCM, if the CPU does not support Advanced Encryption Standard New Instructions (AES-NI) [39].
We determined the utilized cipher suite by looking at the `ClientHello` and `ServerHello`

in `CRYPTO` frames of the `Initial` packets during the handshake. The `ClientHello` contains the list of cipher suites the client supports, in our case the first three cipher suites listed in Table 5.2. The `ServerHello` contains the cipher suite the server chose. The *LSQUIC* server and client behave as follows: If the client detects support for AES-NI on its CPU, it signalizes the preference of the AES-GCM cipher suites over the ChaCha20-Poly1305 cipher suite by ordering the cipher suites as above. If the client does not detect support for AES-NI, it prefers the ChaCha20-Poly1305 cipher suite by reordering the cipher suites so that the ChaCha20-Poly1305 cipher suite is the first in the list. The server chooses the first cipher suite from the list in the `ClientHello` that it also supports. If the client's CPU supports AES-NI, the client only supports AES cipher suites, and the server does not support AES-NI, it will choose the ChaCha20 cipher suite.

To determine the performance difference between the cipher suites, we changed the cipher suites available to the client while constructing the `ClientHello` message before building *BoringSSL*. We used the OpenSSL processor capabilities vector to manually and temporarily disable AES-NI support. This action is done by setting the environment variable `OPENSSL_ia32cap="~0x200000200000000"` [40]. We decided to exclude the `TLS_AES_256_GCM_SHA384` cipher suite from our experiments, as *LSQUIC* never selected it during a handshake in our past experiments.

Figure 5.3 shows the performance difference between the cipher suites. Every area with grey background contains the results of one measurement with 20 repetitions. While the first box plot in each area shows the goodput with the left y-axis, the second and third box plot show the CPU utilization of the respective endpoint with the right y-axis. Both y-axes do not start at zero to better visualize the results.
The highest goodput was achieved with the default configuration of *LSQUIC* in our test setup, where the `TLS_AES_128_GCM_SHA256` cipher suite is used. The second area shows that the performance is significantly worse if AES-NI is not available on both endpoints but server and client still agree on using an AES cipher suite. If the server or client does not support the `TLS_CHACHA20_POLY1305_SHA256` cipher suite, they have to use an AES cipher suite. Using the `TLS_CHACHA20_POLY1305_SHA256` cipher suite if AES-NI is not available is the better choice, which is how *LSQUIC* also does it. In this case, the achieved goodput is only slightly worse than with an AES cipher suite in combination with AES-NI, reaching around 2300 Mbit/s instead of 2600 Mbit/s *LSQUIC*'s `http_client` and `http_server` also try to use the `TLS_CHACHA20_POLY1305_SHA256` cipher suite if AES-NI is not available.

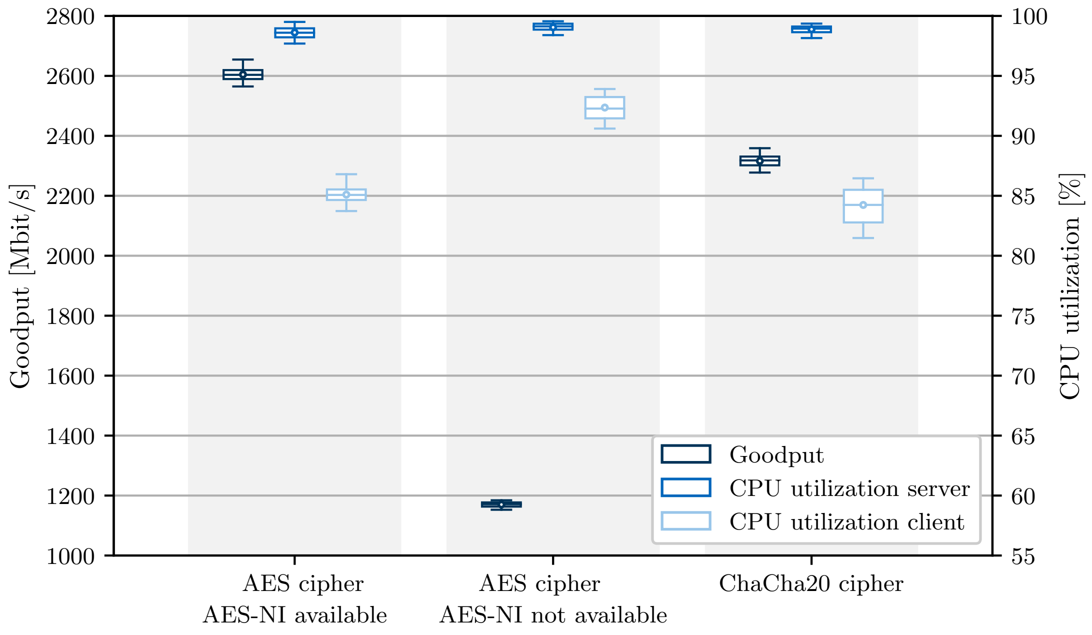

Figure 5.3: Performance of `TLS_AES_128_GCM_SHA256` and `TLS_CHACHA20_POLY1305_SHA256` compared with and without AES-NI support.

Table 5.3: Performance of `TLS_AES_128_GCM_SHA256` with different combinations of AES-NI support on client and server.

AES-NI on Server	AES-NI on Client	Goodput [Mbit/s]
✓	✓	2596.33 ± 31.28
✓	×	1303.03 ± 31.45
×	✓	1123.17 ± 11.26
×	×	1170.02 ± 8.90

In Table 5.3, the goodput in relation to the presence of AES-NI on client and server is shown. As expected, the highest goodput of 2596.33 Mbit/s on average was achieved with AES-NI available on both endpoints.

The goodput is significantly worse if AES-NI is not available at the client, reaching only 1303.03 Mbit/s on average. If AES-NI is not available on both endpoints, the goodput is decreased further by 130 Mbit/s on average. The reason is that encryption is more CPU intensive than decryption, as also observed in the next paragraph. The goodput if AES-NI is not available on the server but available on the client was approximately 50 Mbit/s lower compared to the measurement where both endpoints had no AES-NI

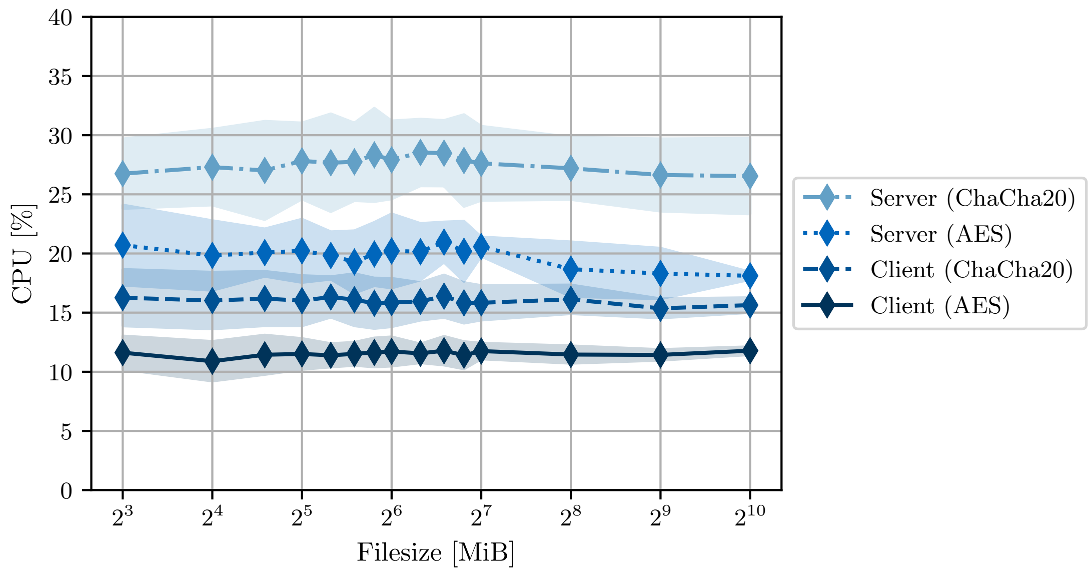

FIGURE 5.4: Contribution of the cryptographic operations to the overall CPU usage of ChaCha20 and AES ciphers on *LSQUIC* server and client compared. The file size of the downloaded file was changed between 8 MiB and 1 GiB. The bandwidth was limited to 100 Mbit/s for all measurements. For each data point, 20 repetitions were performed.

support. However, the deviation is small enough to consider this as a measurement inaccuracy and therefore not investigate it further.

Figure 5.4 compares the CPU utilization share for cryptographic operations dependent on the endpoint and the used cipher. We used `perf` in combination with our categorization script explained in Section 4.5 to determine the displayed values. The standard deviation for each data point is visualized with the error band. AES-NI was available on both endpoints. The file size of the downloaded file was changed between 8 MiB and 1 GiB to see if the initialization of the cryptographic context has an impact on the CPU performance. To ensure a minimal transmission duration also with small files, the bandwidth was limited to 100 Mbit/s for all measurements using the following command:

LISTING 5.2: Command used to limit bandwidth to 100 Mbit/s

```
tc qdisc add dev enp2s0f0 root tbf rate 100mbit latency 50ms burst 1540
```

It can be seen that the client needs approximately 10 % less of the overall used CPU time for cryptographic operations than the server. The ChaCha20 cipher is more CPU consuming than the AES cipher. With a file size of 256 MB, the contribution of the cryptographic operations to the overall CPU usage of the server is 10 % higher with the ChaCha20 cipher. This might be a reason for the higher goodput achieved with the

AES cipher, as the server's CPU is almost fully utilized.
The contribution of the cryptographic operations to the overall CPU usage is nearly constant for all four combinations of ciphers and endpoints over the changing filesize. We can therefore also conclude that the initialization of the cryptographic context is negligible and does not have an impact on CPU utilization.

5.4 Packet IO

The sending and receiving of UDP datagrams is a crucial part of QUIC implementations. As seen in Figure 5.1, the sending and receiving of datagrams consume a lot of CPU time. The performance of the packet IO component therefore highly influences the overall performance of QUIC implementations.

5.4.1 Batching

As there are various system calls to send UDP datagrams, we analyzed which one *LSQUIC* is using by default. The results of our `perf` captures show that *LSQUIC* uses the `sendmsg()` system call. This function call contributes to the overall CPU utilization of *LSQUIC* with 38 % on the server and 3 % on the client. The Linux kernel provides an extension called `sendmmsg()` that allows sending multiple datagrams in one system call. The usage of this extension might improve performance but is not used in *LSQUIC* by default. The command line option `-g` for the *LSQUIC* `http_server` and `http_client` should enable the usage of `sendmmsg()`. The command line option `-j` should enable the usage of `recvmmsg()`, the counterpart of `sendmmsg()` for receiving datagrams.
We planned to compare the performance of *LSQUIC* with and without `sendmsg()` and `recvmmsg()` enabled. Despite encountering issues with the command line options `-g` and `-j`, which caused *LSQUIC* to fail immediately after the handshake, we remained determined to find a solution. A deeper analysis showed that the `sendmmsg()` call returns an `EFAULT` error code. This error code indicates that a memory address passed to the function is invalid. We reported this issue on GitHub but it was not answered as of December 7, 2022. As the command line options are not mentioned in the official documentation, we assume that they are no longer supported by *LSQUIC*.
The performance influence of using `sendmmsg()` and `recvmmsg()` in quiche, Cloudflare's open-source QUIC implementation, was analyzed by Alessandro Ghedini [41]. He was able to significantly decrease the number of system calls for sending packets. The throughput increased by approximately 20 %. We expect a similar performance increase for *LSQUIC*.

5.4.2 Offloading

The Linux kernel offers several techniques for offloading tasks to either the network interface or the kernel itself. The most common offloading techniques are TCP Segmentation Offload (TSO), Large Receive Offload (LRO), Generic Segmentation Offload (GSO), and Generic Receive Offload (GRO). The first two types are only available for TCP and therefore not relevant to our measurements. The last two types are available for both TCP and UDP. [42]

By default, the QUIC implementation is responsible for constructing outgoing packets by combining one or more QUIC frames. One buffer per packet is created and then passed to the kernel. The kernel adds the headers for layers 4 and below before sending. GSO allows the QUIC implementation to pass a buffer too large to fit in one datagram to the kernel. The large buffer will be split into multiple datagrams as late as possible. This might be either in the kernel or even in the network interface, depending on the support of hardware segmentation offload. [41]

In measurements with Cloudflare's QUIC implementation *quiche*, Alessandro Ghedini was able to increase the throughput from approximately 700 Mbit/s to approximately 1300 Mbit/s by enabling GSO [41]. We performed measurements with and without GSO enabled. The gap between the measured goodput values was 15 Mbit/s and therefore within the standard deviation of 30 Mbit/s. The reason is that the application is responsible to add support for sending with GSO and *LSQUIC* does not support GSO. We tried to add support for GSO to *LSQUIC*, which was also tried in the past by Rahul Jadhav [43]. Due to the complexity of the code constructing and sending packets, we were not able to add support for GSO to *LSQUIC* in a reasonable amount of time. Rahul Jadhav compared the performance of *LSQUIC* with his GSO supporting version against the default *LSQUIC* version with and without batching [43]. He provides only perf output as a metric, showing that the best performance was achieved with batching using `sendmmsg()`.

5.4.3 Buffers

As packets can not always be processed immediately, they need to be buffered at multiple points in the network stack. If the buffers are too small, packets might be dropped. The size of the buffers can also influence the performance of QUIC implementations, as congestion control algorithms calculate different sending rates for different buffer sizes. We took a look at the buffer sizes of the UDP sockets used by *LSQUIC*. Every UDP socket has a send buffer and a receive buffer. The UDP receive buffer contains all datagrams that have been received but have not been processed by *LSQUIC* yet. Once it is full, the kernel will discard all incoming packets. The UDP send buffer contains

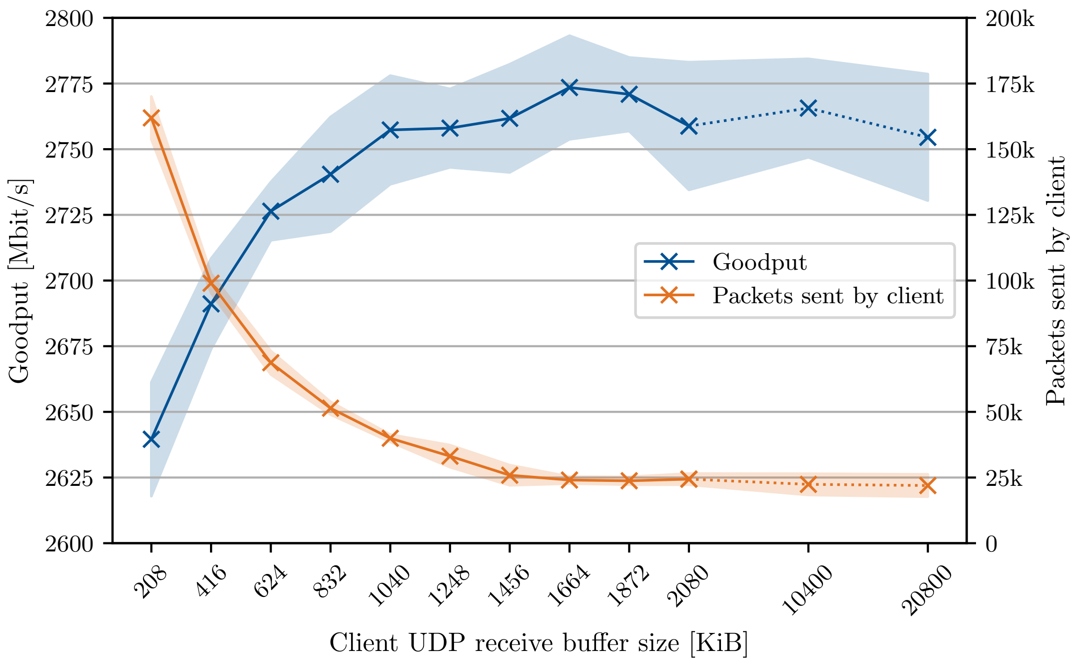

Figure 5.5: Goodput and amount of packets sent by the client with incrementing UDP receive buffer on the client. The default size of the UDP receive buffer is 208 KiB.

all datagrams that are waiting to be sent. The functional principles of the buffers are explained in more detail in Section 2.3.

Since the preparation of a datagram is usually more expensive than the sending of a datagram, the send buffers do not fill up. The receive buffers instead fill up quickly, as the sockets are faster in receiving packets than *LSQUIC* is in processing them. In our measurements, most of the data is transferred from the server to the client, which is why we expect that the client socket receive buffer is the first and only buffer that is likely to reach capacity.

We performed multiple measurements to determine the effect on the performance of the client's UDP receive buffer size. In Figure 5.5, the results of the measurements are shown. We performed 12 measurements with 20 repetitions each. On the x-axis, the size of the receive buffer in KiB is marked. The default size of the receive buffer is 208 KiB. The blue line shows the average goodput of the measurements with the standard deviation as the shaded area. The left y-axis shows a scale for this goodput line and is limited from 2600 Mbit/s to 2800 Mbit/s to increase the readability of the goodput change. The orange line shows the average number of packets sent by the client. The right y-axis provides a scale for the number of packets. We decided to include the average number of packets sent by the client into the plot, as we observed a massive change in the values for the different buffer sizes.

It can be seen that the goodput starts to increase with the also increasing UDP receive buffer size. When doubling the size of the receive buffer, the goodput already increases by more than 50 Mbit/s on average. It can also be observed that this change in the size of the receive buffer causes the client to send 30 % fewer packets. When further increasing the size of the receive buffer, the goodput increases even more. As soon as the receive buffer has reached five times the size of its initial default value, the goodput seems to stay constant. The two rightmost data points, increasing the buffer size up to factor 100 of its initial default value, show that the performance does not increase further.
A possible explanation for this behavior is that the kernel drops incoming packets if the receive buffer is full. The tool `netstat` can provide the amount of dropped packets because of insufficient UDP receive buffer size. The number of dropped packets for different UDP receive buffer sizes are shown in Figure 5.6. Every box in the plot represents 20 repetitions of a measurement with the respective UDP receive buffer size. It can be seen that with our default measurement (8 GiB single file transfer over 10 Gbit/s link), the kernel drops around 12 000 packets in average with the default UDP receive buffer size. As the server sent 6 000 000 packets on average, approximately 0.2 % of the packets are lost due to shallow UDP receive buffers. With a UDP receive buffer size of 1664 KiB, equal to eight times the default size, the kernel drops approximately 1000 packets on average, causing CUBIC to increase the sending rate and therefore the goodput.
With the presented results, the logical measure to improve performance seems to be to increase the size of the buffers. A larger buffer requires more memory, but fewer packets are dropped. However, it should also be noted that the buffer size should not be made unnecessarily large. Because of the Bufferbloat phenomenon, the performance can worsen again by too large buffers [44]. The reason for this is the high latency and jitter, which occurs when packets stay in a buffer for a long time.

As we also observed a high deviation in the number of packets sent by the client with BBR, we performed a second measurement to compare CUBIC and BBR. The results are displayed in Table 5.4. While increasing the client's UDP receive buffer when using CUBIC leads to fewer acknowledgments from the clients, fewer packet drops at the client, and a higher goodput, the results for BBR are surprisingly different. Since BBR is not loss based, the amount of packet drops is significantly smaller with the default UDP receive buffer size. With an eight times larger buffer size, the number of packet drops could be reduced to zero. However, the acknowledgment frequency got slightly higher and the goodput decreased by 17 %. As the measured goodput and

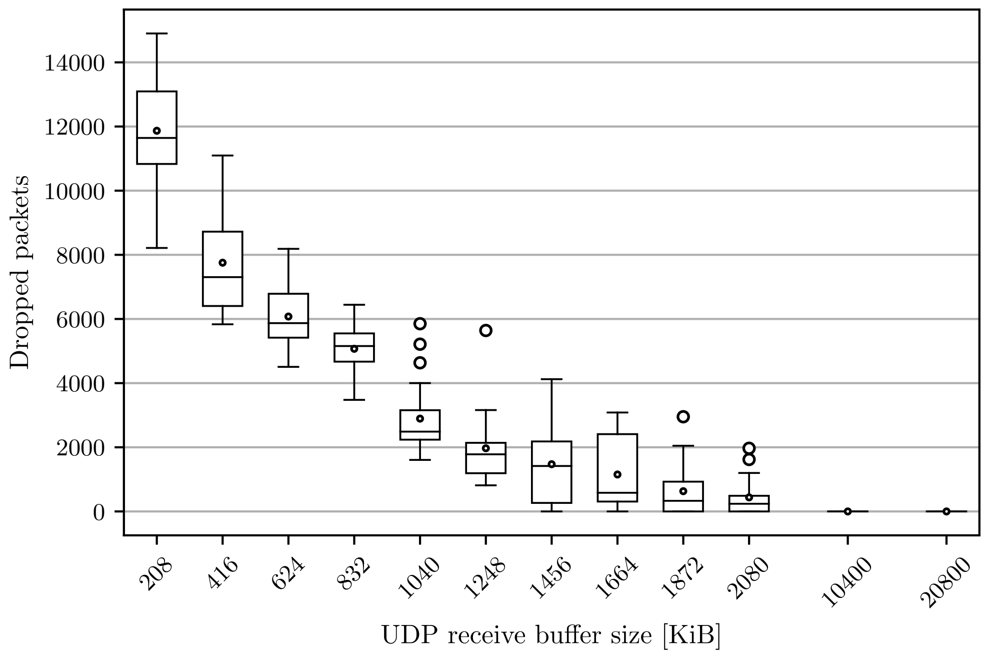

Figure 5.6: Amount of packets dropped at the client's UDP socket receive buffer.

acknowledgment frequency for BBR fluctuate a lot, these results are not very reliable. Due to a standard deviation between 20 % and 40 %, we assume that the performance of BBR is not influenced by the UDP receive buffer size in this setup.

5.5 Build Optimization

In this section, we evaluated general performance issues that are not specific to a particular component. As the CPU utilization of server and client stays close to 100 % during

Table 5.4: ACK's and packet drops for different congestion control algorithms. All metrics are average values of 20 repetitions rounded to integers with the standard deviation.

Configuration		Results		
CCA	BUF[a] [KiB]	Goodput [Mbit/s]	ACK's[b]	Drops[c]
BBR	208	1089 ± 40%	1 037 814 ± 37%	48 ± 167%
BBR	1664	904 ± 31%	1 178 723 ± 20%	0 ± 0%
CUBIC	208	2607 ± 1%	181 240 ± 7%	11 221 ± 19%
CUBIC	1664	2770 ± 1%	24 143 ± 9%	622 ± 99%

[a] Size of the UDP receive buffer, [b] Number of packets sent by the client,
[c] number of packets dropped by the client's kernel due to insufficient receive buffer size

Listing 5.3: *LSQUIC* default build type.

```
IF("${CMAKE_BUILD_TYPE}" STREQUAL "")
        SET(CMAKE_BUILD_TYPE Debug)
ENDIF()
```

Listing 5.4: *LSQUIC* default optimization flags. Commented lines removed.

```
IF(CMAKE_BUILD_TYPE STREQUAL "Debug")
        SET(MY_CMAKE_FLAGS "${MY_CMAKE_FLAGS}␣-O0␣-g3")
        [...]
ELSE()
        SET(MY_CMAKE_FLAGS "${MY_CMAKE_FLAGS}␣-O3␣-g0")
ENDIF()
```

our measurements, we identified CPU utilization as a significant factor for performance. We tried to lower CPU utilization by optimizing the build process to improve overall performance.

LSQUIC uses the *CMake* build system and the GNU Compiler Collection (GCC) for building. We use `make` version 4.3, `CMake` version 3.18.4, and `GCC` version 10.2.1 inside a Debian 11 *Docker* container to build *LSQUIC*.

GCC provides many optimization flags for performance improvement. The overall level of optimization can be set with the `-O` flag. Level 0 is used by default and does not perform any optimizations. The recommended level is 2, which should improve performance without increasing the compilation time or binary size too much. The highest optimization level is 3, which is known to increase the binary size and compilation time. The increase in the size of the output binary completely depends on the source code. In our case, the size of *LSQUIC*'s `http_server` increased from 15.2 MB to 20.5 MB when using `-O3`. The additional size of the binaries as well as the increased compilation time are not a problem for our measurements, as we only compile the binaries once and the larger filesize is still inside the usual limits. Another optimization option is `-march`, which enables optimizations for a specific CPU architecture. Using the `-march` option can cause a significant performance improvement, depending on the CPU used. Optimizing for a specific CPU architecture and instruction set causes the binary to be incompatible with other CPUs. The best performance can be achieved by using the `-march=native` option, leading the compiler to optimize for the CPU the binary is compiled on.

By default, *LSQUIC* is built with the build type `Debug`, as shown in Listing 5.3. If the build type is set to `Release` (or any value other than `Debug`), the optimization flag `-O3` is used, as shown in Listing 5.4.

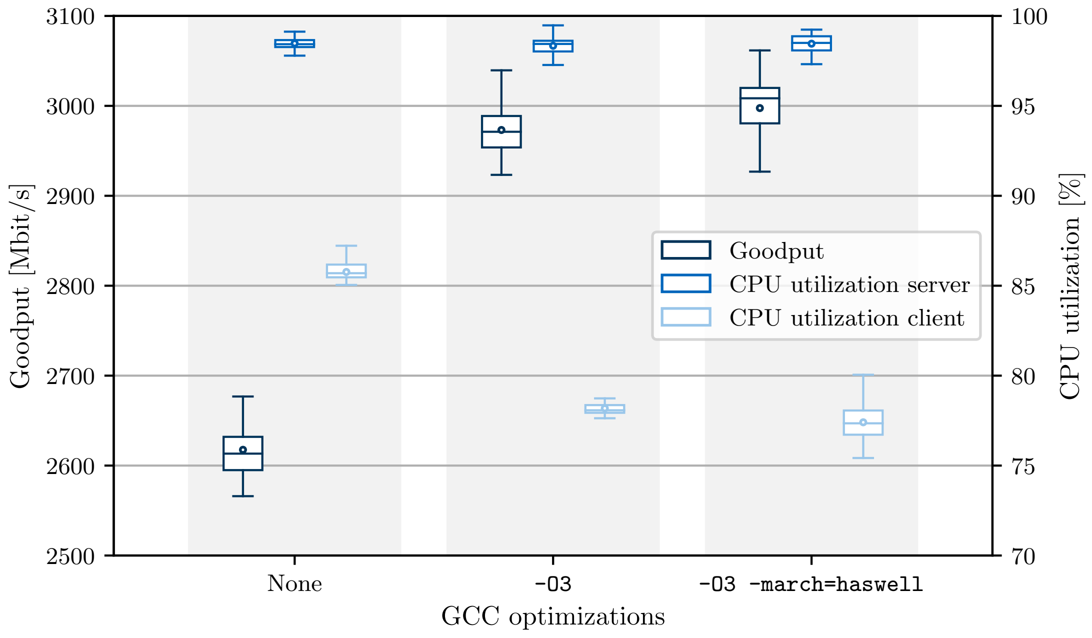

Figure 5.7: Performance of different build optimization levels compared.

We compared the default build with no optimizations to a build with the optimization flag `-O3` and another build with the `-march=native` option additionally used. Figure 5.7 shows the results of the measurements. We performed three measurements with 20 repetitions each, one measurement per build. Every area with grey background contains the results of one measurement, shown in the form of box plots. While the first box plot shows the goodput with the left y-axis, the second and third box plot shows the CPU utilization of the respective endpoint with the right y-axis. Both y-axes do not start at zero to better visualize the results.

As expected, the build with the `-O3` flag performs better than the default build without optimizations. The goodput could be increased by more than 10 %. By using the `-march=native` flag, we were able to slightly increase the goodput by roughly another percent. The CPU utilization of the server is at a constant level of approximately 98 % for all three builds. The CPU utilization of the client decreases for more optimized builds. It can be concluded that the server's CPU is the bottleneck for the performance of the QUIC connection. The optimized builds enable the server to achieve a higher goodput while still using the same amount of CPU resources. The optimizations on the client cause a decrease in CPU utilization.

Chapter 6

Conclusion

This chapter summarizes our results and contributions. Additionally, we provide an outlook for further research and possible future work.

6.1 Summary

Our results can be presented best by answering our research questions:

How do we need to design a measurement framework to analyze the performance of transport protocols?

We developed a measurement setup to analyze and compare the performance of QUIC implementations. The setup is configurable and can be used to execute automated experiments in a reproducible way. By extending the existing open source *QUIC Interop Runner*, our measurements can be easily ported to other QUIC implementations. Our modifications allow for a more detailed analysis with numerous metrics. Using newly integrated tools and the corresponding logfile parsers, measurement results can be quickly evaluated and visualized.

Which parts of QUIC implementations limit performance the most?

With our call stack categorization script, we were able to break the CPU usage of QUIC implementations down to individual components. The most CPU-demanding parts are the *packet IO* and *crypto* components. Although cryptographic operations are CPU intensive, cryptography is not a performance bottleneck in *LSQUIC*.
The *connection management* and *packet IO* components turned out to be the most performance-limiting parts in our measurements.

While congestion control implemented in user space is not as robust as the kernel's congestion control, the lack of adaptation of the acknowledgment frequency leads to additional overhead at both endpoints. The sending and receiving of packets is very CPU intensive and can be improved by using batching and offloading, which are both not supported by *LSQUIC*. Shallow buffers caused by low maximum socket buffer sizes lead to dropped packets and lower sending rates.

Which configurations influence the performance of QUIC implementations?

We identified that for *LSQUIC*, significant performance differences are caused by the choice of the congestion control algorithm. *LSQUIC*'s BBR implementation could not achieve a steady and similarly high goodput as CUBIC in our measurement setup. We discovered the client's receive buffer size and the RTT as reasons for this behavior. For *LSQUIC*'s CUBIC implementation, the CPU utilization at the server was the main bottleneck preventing a higher goodput.
By optimizing the build process of *LSQUIC* for the target CPU, we could further increase the performance of *LSQUIC* by more than 10 %. Building *LSQUIC* with default parameters causes the compiler to not apply any optimizations.
The default UDP socket receive buffer size of 208 KiB turned out to be too small for high bandwidth usage. Dropped packets due to a full receive buffer result in the sender lowering its sending rate. For our measurements, the highest performance could be achieved with a receive buffer size of approximately 2 MiB.
We have learned that the IETF QUIC standard does not strictly specify how the acknowledgment frequency should be determined for different implementations. Sending an unnecessarily large number of acknowledgments with high bandwidth leads to additional overhead on the server since the server must receive, decrypt, and process the acknowledgments. We observed a large deviation in the acknowledgment frequency between our measurements. We identified this behavior as a performance limitation, especially for high bandwidth networks.
The Linux kernel provides several mechanisms to reduce the CPU usage of packet sending. These include the acceptance of multiple outgoing packets at once (batching) or the offloading of packet segmentation to the kernel or the NIC. *LSQUIC* provides a command line flag to enable batching as it is not used by default. However, the batching implementation seems deprecated and does not work as expected. *LSQUIC* does also not support the offloading of packet segmentation, resulting in a potentially large performance benefit being missed.

6.2 Future Work

As more than 20 open-source QUIC implementations are available, other implementations could be analyzed in the future. Our framework can be easily used with other implementations. Most features and tools integrated offer immediate support without any changes needed. Related work showed that other QUIC implementations also have performance issues. Further investigation could reveal them and help to improve the performance.

The acknowledgment frequency turned out to have a considerable impact on the performance of QUIC connections. In all of our measurements, we observed a correlation between the acknowledgment frequency and the achieved goodput. The acknowledgment frequency is currently not a configurable parameter in *LSQUIC* but the delayed ACK's extension allows the endpoints to influence it. The acknowledgment frequency could be further investigated to determine how to achieve better performance in combination with the delayed ACK's extension.

Sending and receiving packets is very costly in terms of performance. Support for batching and offloading is built into every recent kernel but not used by QUIC implementations like *LSQUIC*. As explained in Section 5.4, abandoning the use of GSO or `sendmmsg()` unnecessarily costs the server a lot of CPU time. To use batching or offloading, only the source code of the QUIC implementation needs to be changed. Integrating these features into *LSQUIC* involves major changes to the crucial parts of its source code, which is beyond the scope of this thesis. Instead, we propose this as possible future work.

Another interesting topic we did not cover in this thesis is the general performance improvement by using techniques allowing faster packet processing. We collected information about combining QUIC with DPDK or eXpress Data Path (XDP). Both techniques allow for bypassing the kernel's network stack. Especially the usage of XDP seems interesting but involves major changes to the QUIC implementation. A proof of work implementation of *LSQUIC* with XDP was presented by LiteSpeedTech in 2020 [45]. They achieved a 43 % performance improvement and concluded that combining QUIC with XDP is a promising approach, especially for web servers running Linux. Microsoft's QUIC implementation *MsQuic* claims to stand out from others by optimizing for high throughput and low latency [46]. They recently published a blog post on their current progress in integrating XDP with *MsQuic* [47]. They present figures showing massive performance improvements with XDP. Microsoft also sees XDP as a promising approach for QUIC performance and stated to keep investing in *MsQuic* with XDP.

Chapter A

Appendix

A.1 List of Acronyms

AES-NI	Advanced Encryption Standard New Instructions, an instruction set for x86 CPU's to improve the speed of AES de- and encryption.
BBR	Bottleneck Bandwidth and Round-trip propagation time, a model-based congestion control algorithm.
BDP	bandwidth delay product
CPU	central processing unit
DMA	Direct Memory Access
DPDK	Data Plane Development Kit, a library to achieve faster packet processing.
GCC	GNU Compiler Collection, a compiler for various programming languages.
GRO	Generic Receive Offload
GSO	Generic Segmentation Offload
HTTP	Hypertext Transfer Protocol, an application layer protocol for data transfer, mostly used for the transmission of web pages.
IETF	Internet Engineering Task Force, an open standards organization dealing with technical development of the Internet.
IQR	inter-quartile range
LRO	Large Receive Offload
NetEm	Network Emulator, a set of features for the Linux command line tool tc for emulating network conditions.
NIC	network interface card, a computer hardware component connecting the computer to a network.

PCAP	packet capture, a file format.
PID	Process Identifier, a unique number to identify a running process.
PLT	page load time, a performance metric.
POS	Plain Orchestrating Service, a testbed management system.
pps	packets per second, a measure of throughput.
qdisc	queueing discipline, a type of queue for egress traffic in the Linux kernel.
RTT	round trip time. The time it takes for a signal to travel from sender to receiver and back.
SRTT	smoothed round trip time
tbf	token bucket filter, a queueing discipline of the Linux traffic control.
TCP	Transmission Control Protocol, a connection-oriented, reliable transport layer protocol.
TLS	Transport Layer Security, a cryptographic protocol.
TSO	TCP Segmentation Offload
UDP	User Datagram Protocol, a datagram-oriented, unreliable transport layer protocol.
WAN	wide area network
XDP	eXpress Data Path, an network packet processor using eBPF.

Bibliography

[1] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, Aug. 2022. DOI: 10.17487/RFC9293. [Online]. Available: https://www.rfc-editor.org/info/rfc9293.

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

[3] J. Roskind, *QUIC Versions*, https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34, Accessed: 2022-11-05, 2012.

[4] J. Roskind, *Experimenting with QUIC*, https://blog.chromium.org/2013/06/experimenting-with-quic.html, Accessed: 2022-11-26, 2013.

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

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

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

[8] A. Yu and T. A. Benson, „Dissecting Performance of Production QUIC“, in *Proceedings of the Web Conference 2021*, ser. WWW '21, Ljubljana, Slovenia: Association for Computing Machinery, 2021, 1157–1168, ISBN: 9781450383127. DOI: 10.1145/3442381.3450103. [Online]. Available: https://doi.org/10.1145/3442381.3450103.

[9] F. Rochet, E. Assogba, M. Piraux, K. Edeline, B. Donnet, and O. Bonaventure, „TCPLS: Modern Transport Services with TCP and TLS“, in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '21, Virtual Event, Germany: Association for Computing

Machinery, 2021, 45–59, ISBN: 9781450390989. DOI: 10.1145/3485983.3494865. [Online]. Available: https://doi.org/10.1145/3485983.3494865.

[10] A. Mishra, S. Lim, and B. Leong, „Understanding Speciation in QUIC Congestion Control“, in *Proceedings of the 22nd ACM Internet Measurement Conference*, ser. IMC ’22, Nice, France: Association for Computing Machinery, 2022, 560–566, ISBN: 9781450392594. DOI: 10.1145/3517745.3561459. [Online]. Available: https://doi.org/10.1145/3517745.3561459.

[11] D. Saif, C. Lung, and A. Matrawy, „An Early Benchmark of Quality of Experience Between HTTP/2 and HTTP/3 using Lighthouse“, *CoRR*, vol. abs/2004.01978, 2020. arXiv: 2004.01978. [Online]. Available: https://arxiv.org/abs/2004.01978.

[12] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, „It’s over 9000: Analyzing early QUIC Deployments with the Standardization on the Horizon“, in *Proceedings of the 2021 Internet Measurement Conference*, Virtual Event, USA: ACM, Nov. 2021. DOI: 10.1145/3487552.3487826.

[13] J. Iyengar and I. Swett, „QUIC Acknowledgement Frequency“, Internet Engineering Task Force, Internet-Draft draft-ietf-quic-ack-frequency-02, Jul. 2022, Work in Progress, 13 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-ack-frequency/02/.

[14] *UDP/TCP/IP Performance Overview*, https://sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/aixbman/prftungd/udptcpperfov.htm, Accessed: 2022-11-23.

[15] K. Jamshaid, B. Shihada, A. Showail, and P. Levis, „Deflating link buffers in a wireless mesh network“, *Ad Hoc Networks*, vol. 16, pp. 266–280, 2014, ISSN: 1570-8705. DOI: 10.1016/j.adhoc.2014.01.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1570870514000134.

[16] X. Yang, L. Eggert, J. Ott, S. Uhlig, Z. Sun, and G. Antichi, „Making QUIC Quicker With NIC Offload“, in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ ’20, Virtual Event, USA: Association for Computing Machinery, 2020, 21–27, ISBN: 9781450380478. DOI: 10.1145/3405796.3405827. [Online]. Available: https://doi.org/10.1145/3405796.3405827.

[17] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, *TCP Fast Open*, RFC 7413, Dec. 2014. DOI: 10.17487/RFC7413. [Online]. Available: https://www.rfc-editor.org/info/rfc7413.

[18] M. Richards, *Linux Kernel vs DPDK: HTTP Performance Showdown*, Blog Post: https://talawah.io/blog/linux-kernel-vs-dpdk-http-performance-showdown, Jul. 2022.

[19] S. Gallenmüller*, D. Scholz*, H. Stubbe, and G. Carle, „The pos Framework: A Methodology and Toolchain for Reproducible Network Experiments“, in *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21)*, Munich, Germany (Virtual Event), Dec. 2021. DOI: `10.1145/3485983.3494841`.

[20] I. Q. W. Group, *QUIC Implementations*, `https://github.com/quicwg/base-drafts/wiki/Implementations`, Accessed: 2022-10-17, 2022.

[21] T. Pauly, E. Kinnear, and D. Schinazi, *An Unreliable Datagram Extension to QUIC*, RFC 9221, Mar. 2022. DOI: `10.17487/RFC9221`. [Online]. Available: `https://www.rfc-editor.org/info/rfc9221`.

[22] M. Seemann, *QUIC Interop Runner*, `https://github.com/marten-seemann/quic-interop-runner`, Accessed: 2022-06-08, 2022.

[23] B. Gregg, *Linux perf Examples*, `https://brendangregg.com/perf.html`, Accessed: 2022-10-13, 2020. [Online]. Available: `https://brendangregg.com/perf.html`.

[24] *perf (1) — Linux manual page*, Accessed: 2022-10-13. [Online]. Available: `https://man7.org/linux/man-pages/man1/perf.1.html`.

[25] *pidstat (1) — Linux manual page*, Accessed: 2022-12-03. [Online]. Available: `https://man7.org/linux/man-pages/man1/pidstat.1.html`.

[26] *ethtool (8) — Linux manual page*, Accessed: 2022-12-08. [Online]. Available: `https://man7.org/linux/man-pages/man8/ethtool.8.html`.

[27] *netstat (8) — Linux manual page*, Accessed: 2022-12-08. [Online]. Available: `https://man7.org/linux/man-pages/man8/netstat.8.html`.

[28] *tc (8) — Linux manual page*, Accessed: 2022-11-30. [Online]. Available: `https://man7.org/linux/man-pages/man8/tc.8.html`.

[29] *tc-netem (8) — Linux manual page*, Accessed: 2022-11-30. [Online]. Available: `https://man7.org/linux/man-pages/man8/tc-netem.8.html`.

[30] *tc-tbf (8) — Linux manual page*, Accessed: 2022-11-30. [Online]. Available: `https://man7.org/linux/man-pages/man8/tc-tbf.8.html`.

[31] Brendan Gregg, *FlameGraph*, `https://github.com/brendangregg/FlameGraph`, Accessed: 2022-10-31, 2022.

[32] K. Ploch, „QUIC Performance on 10G Links“, BA thesis, Technical University of Munich, 2022.

[33] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, „Towards a Deeper Understanding of TCP BBR Congestion Control“, in *IFIP Networking 2018*, Zurich, Switzerland, May 2018. DOI: `10.23919/IFIPNetworking.2018.8696830`.

[34] J. Iyengar, *ACK generation recommendation*, https://github.com/quicwg/base-drafts/issues/3304, Accessed: 2022-11-07.

[35] G. Fairhurst, A. Custura, and T. Jones, „Changing the Default QUIC ACK Policy“, Internet Engineering Task Force, Internet-Draft draft-fairhurst-quic-ack-scaling-04, Mar. 2021, Work in Progress, 19 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-fairhurst-quic-ack-scaling/04/.

[36] A. Custura, T. Jones, and G. Fairhurst, „Rethinking ACKs at the Transport Layer“, in *IFIP Networking 2020*, 2020, pp. 731–736.

[37] E. Volodina and E. P. Rathgeb, „Impact of ACK Scaling Policies on QUIC Performance“, in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*, 2021, pp. 41–48. DOI: 10.1109/LCN52139.2021.9524947.

[38] Google, *BoringSSL*, https://github.com/google/boringssl, Accessed: 2022-10-28, 2022.

[39] Y. Nir and A. Langley, *ChaCha20 and Poly1305 for IETF Protocols*, RFC 8439, Jun. 2018. DOI: 10.17487/RFC8439. [Online]. Available: https://www.rfc-editor.org/info/rfc8439.

[40] *OPENSSL ia32cap - manual page*, Accessed: 2022-11-07. [Online]. Available: https://www.openssl.org/docs/manmaster/man3/OPENSSL_ia32cap.html.

[41] A. Ghedini, *Accelerating UDP packet transmission for QUIC*, Blog Post: https://blog.cloudflare.com/accelerating-udp-packet-transmission-for-quic/, Jan. 2020.

[42] *NIC Offloads*, https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-nic-offloads, Accessed: 2022-11-28.

[43] R. Jadhav, *UDP GSO Support*, https://github.com/litespeedtech/lsquic/pull/135, Accessed: 2022-11-27.

[44] R. Pan, P. Natarajan, C. Piglione, *et al.*, „PIE: A lightweight control scheme to address the bufferbloat problem“, in *2013 IEEE 14th International Conference on High Performance Switching and Routing (HPSR)*, 2013, pp. 148–155. DOI: 10.1109/HPSR.2013.6602305.

[45] R. Perper, *Performance Comparison of QUIC with UDP and XDP*, Blog Post: https://blog.litespeedtech.com/2020/06/01/performance-comparison-quic-udp-xdp/, Jun. 2020.

[46] Microsoft, *MsQuic*, https://github.com/microsoft/msquic, Accessed: 2022-12-01, 2022.

[47] Y. Huang, *Balance Performance in MsQuic and XDP*, Blog Post: https://techcommunity.microsoft.com/t5/networking-blog/balance-performance-in-msquic-and-xdp/ba-p/3627665, Sep. 2022.