Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

QUIC Performance on 10G Links

Kevin Ploch

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

QUIC Performance on 10G Links

QUIC Performanz auf 10G Verbindungen

Author:	Kevin Ploch
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Benedikt Jaeger, Johannes Zirngibl
Date:	October 15, 2022

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

Garching, October 15, 2022

Location, Date

Signature

Abstract

This work aims to shed light on the performance of QUIC and the CPU cost associated with its components in single-connection scenarios dictated by the congestion avoidance phase. Fitting to QUICs initial design motivation to improve web performance, many works focus on the use case of short measurement scenarios, favoring a fast connection setup. We argue that for QUIC to be usable as the general purpose transport layer protocol it was recently standardized as, we require more data on longer transfers with handshakes playing less of a role, and where a transport service user would likely turn to Transmission Control Protocol (TCP). We realized structured, reproducible measurements by extending the QUIC Interop Runner measurement framework to work for bare-metal experiments on real, physically distinct machines. We found that in long-duration scenarios, QUIC achieved only half the goodput of TCP+Transport Layer Security (TLS) when excluding Network Interface Card (NIC) offloading, and about one-third of it with offloading support for TCP. Both QUIC and TCP transfers reached 100 % CPU utilization on the server for the core they ran on. When looking at how QUIC uses its CPU time, we identified packet I/O as the most expensive component regarding CPU utilization, followed by crypto operations. Using this approach of a structured framework based on QUIC interoperability allows for easier, future hardware measurements in needed areas like flow control, where variables like ACK frequency are not yet unified by a standard document.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With QUICs official standardization in 2021, TCP acquired a strong competitor as a general-purpose protocol, which addresses strong pain points like the too long TCP+TLS 2-RTT handshake on fresh connection, HTTP/2 unforeseen head-of-line blocking on the transport layer, among other improvements. These changes, especially the decision of placing QUIC components in the userspace on top of User Datagram Protocol (UDP), triggers a need for throughout analysis to be able to decide its suitability as a replacement in many areas. This is why a lot of research has been conducted since QUICs first draft designs by the IETF started in 2016. Due to the first draft only being declared as an official internet standard recently in 2021, many works focus on QUIC implementations based on outdated drafts [1]–[4]. Many works also focus on situations that favor QUIC with its faster connection setup [3], [5]–[7]. To fullfill its application as a general-purpose transport layer protocol, we argue that it also needs to be analysed in longer transfers, dictated by the congestion avoidance phase and minimally impacted by connection setup. We implemented our measurements after the two requirements of 1) experiments in environments with no virtualization layers (process virtualization, hypervisors) and 2) the possibility for comparisons between different QUIC implementations. The second point is especially important as there are topics like the frequency of flow control updates and ACK frequency, that are so far not specified in a unifying standard, which leads to an implementation situation mirroring the wild west. With the measurement framework of the QUIC Interop Runner [8] we found a good base for these requirements. We concretely aim to measure the performance in terms of goodput and the CPU utilization of the different QUIC components in long, single-stream transfers dominated by congestion avoidance, and in different settings considering available bandwidth, flow control resources, and cipher options.

The rest of this work is structured as follows: Chapter 2 explains the background surrounding the QUIC by shedding some light on the motivation of its creation, how it differs from TCP and what the situation of current implementations is. We also explain the metric definition for our conducted measurements and tools used to gain insight into QUIC processes. In Chapter 3 we show how we conducted our measurements in an automated, reproducible way, by explaining the design of the used measurement framework. Additionally, we show the testbed topology used in our measurements. Chapter 4 reveals the results of five implemented measurements, that show different aspects of QUICs performance and component behavior in high duration, minimal connection setup impacted transfers. Additionally, we compared it directly to TCP with TLS in an identical machine and network setup. Chapter 5 shows the most tightly linked related work, and how our work differs and addressed pain points that impact the achieved results.

Chapter 2

Background

This chapter provides information on QUIC itself and other influences on transport layer transfers. We also introduce tools that are used for collecting extensive information on QUIC and processes on Linux systems in general in conducted measurements.

2.1 QUIC

QUIC is a general-purpose, connection-oriented transport layer protocol, recently standardized by the IETF in May of 2021 [9]–[11], and a direct competitor to TCP. While TCP has been improved over a long timespan since its first official draft in 1981 [12], there are certain, deeply rooted design decisions in the protocol itself that leave designers no other choice but to implement a new transport layer protocol with new fundamental properties. It is not easy to make a widespread switch to a new transport layer protocol, as seen in the example of Stream Control Transmission Protocol (SCTP), which could be attributed to middleboxes having problems with handling new protocols, the fact that middlebox upgrades are expensive and their providers do not have an incentive to change this situation. QUIC has succeeded as it is built on top of UDP, effectively being supported by every device that supports UDP [13, Section 5]. The follwing subsections aim to reveal differences between TCP and QUIC, and explain the motivations for the creation of this new protocol.

2.1.1 Placement in Userspace and on UDP

Figure 2.1 reveals a fundamental design decision of QUIC: it is based on the UDP and implements its components in userspace. Instead of developing applications traditionally using HTTP and TLS from userspace, then interfacing to TCP in the kernel

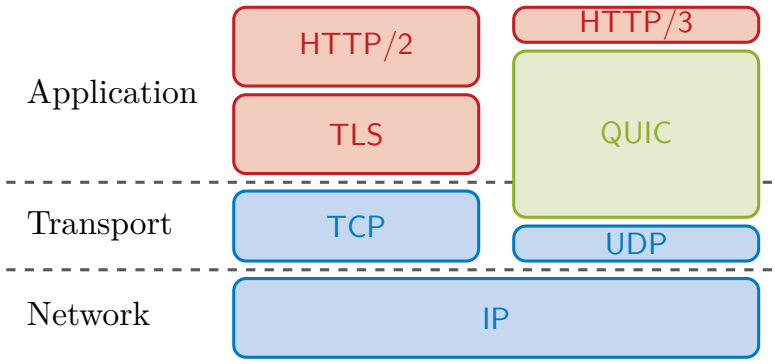


Figure 2.1: QUIC and TCP Protocol Stack, from [14].

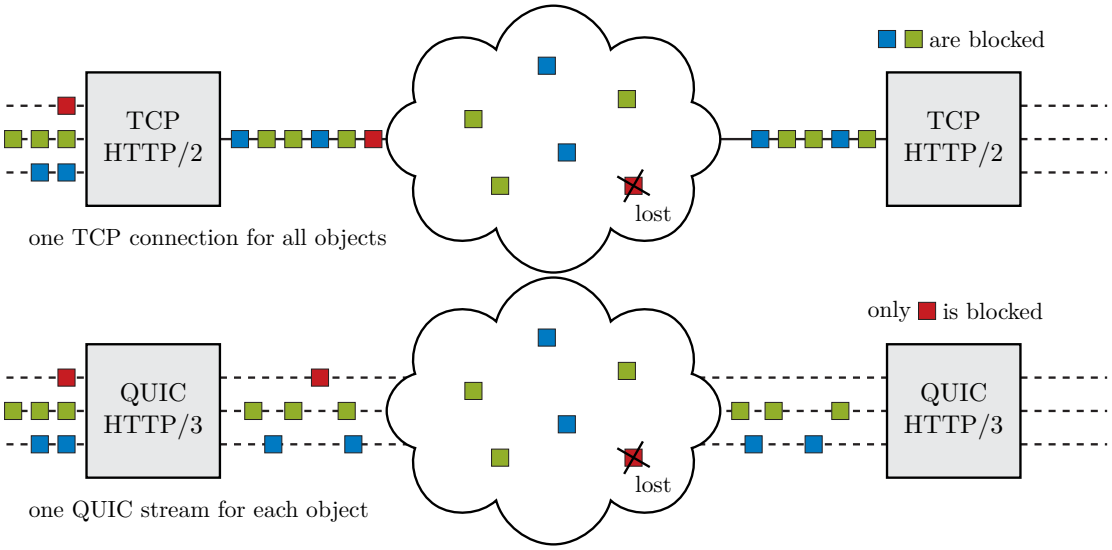


Figure 2.2: Visualization of the Head-of-Line blocking problem with TCP and QUIC.

through sockets, QUIC is provided as a userspace library, implementing transport layer components like flow control, congestion control or ACK tracking in userspace. It then interfaces to connectionless UDP sockets implemented in the kernel to send segments. It also directly incorporates TLS 1.3 cryptography, which is an advantage utilized by its handshake design, later explained in Section 2.1.3. HTTP/3 is the new HTTP version, needed to delegate stream features present in HTTP/2 to QUIC, which is explained in more detail in Section 2.1.2. The placement in userspace leads to the advantage of it being easier to update frequently – updates to TCP in the kernel are relatively slow, as it has to be included in OS update cycles. A disadvantage lies in performance – we will explore this point in detail in Section 4.3.

2.1.2 Stream Concept and TCP Head-of-Line blocking

Head-of-line blocking is the phenomenon when elements in a queue are blocked by the first element. To explain how QUIC solved head-of-line blocking at the transport layer using stream multiplexing, we start off by explaining it at the application layer.

Web traffic based on TCP uses the left protocol stack shown in Figure 2.1. A resource requested by an HTTP/2 client is wrapped first in an HTTP/2 header, then encrypted using TLS, and finally passed to TCP to be transmitted over the network. A key problem that lead to the development of HTTP/2 is the property, that HTTP/1.0 and HTTP/1.1 communication over a given TCP connection only allowed for one outstanding file request at a time [15, Section 1]. The protocol simply did not implement mechanisms to differentiate between data of different requests, and so could only serve one request after the other. This leads to large delays when serving a big file in front of a smaller one. HTTP/2 introduced *streams* by prepending each sent data piece with a header, which carries a stream identifier to let an HTTP/2 receiver know to which request this data belongs. The data and such a header together form a unit called *frame*. A bidirectional flow of frames with the same stream id then defines a single stream [15]. We thoroughly explain these HTTP/2 naming conventions at this point, as QUIC itself also uses them. These streams then allow us to multiplex different files into one TCP connection.

HTTP/2 traffic can now be visualized as in Figure 2.2, where two endpoints establish one TCP connection to transfer webobjects, each with its own stream. As we can differentiate between different webobjects through frames, HTTP/2 can now progress the transfer of multiple files at once by intertwining frames of different streams and passing them toghether to TCP. TCP splits this incoming data into different segments of the right size, and proceeds to send them to the other host. Should a segment be lost, TCP awaits a retransmission before handing any more data to HTTP, as TCP always delivers data in-order to above layers. Although this loss could only involve frames of one stream (e.g., red in the figure), it will not pass on segments after the loss (blue and green) as TCP is not aware of this multiplexing, but only of a generic byte stream received from upper layers. QUIC solves this problem by introducing streams on the transport layer, each with in-order delivery, which can be seen in the bottom image of Figure 2.2. Lost segments of one stream now do not block the progress of other streams, curating head-of-line blocking at the transport layer.

2.1.3 Combined Transport Layer and TLS Handshake

TCP starts communication off with a three-way handshake, which requires one round-trip time (RTT). Should a connection also require encryption, the first data TCP

sends kicks off the TLS handshake, which takes two additional RTT, resulting in a connection establishment requiring 3 RTT [16, Section 7.3]. QUIC combines the typical three-way and TLS handshake into one overarching handshake, which is not possible for TCP due to the layer separation of TCP and TLS. TCP is not aware of any upper-layer logic and simply receives generic bytes to transfer after its handshake. QUIC achieves a 1-RTT handshake in the general case, as well as a 0-RTT handshake for an older, cryptographically already established connection which is now being continued [9, Section 7.1].

2.1.4 Flow Control

As QUIC borrows flow control concepts from TCP, we can explain flow control mechanisms in QUIC by starting off with TCP. With the introduction of the transport layer in the layered structure of the internet protocol suite, computer networking is able to distinguish between multiple applications running on one host. This is in practice accomplished using port numbers and the port number field in transport layer protocols, where each application is assigned one or more port numbers for communication on the network.

This concept of a multi-application system introduces new problems, one being the fact that each application requires explicit memory for networking tasks like storing state information. In the case of QUIC and TCP, this takes form as buffers for segments we are planning to send (*send window*) and segments we expect to receive (*receive window*). Applications are able to place data to be sent in the send buffer, and consume received data from the according receive buffer. Because operating systems have a limited amount of memory, applications are given an upper limit on network memory usage in form of upper limits to these data structures. This protects the machine from both fast-sending, memory-hungry applications and fast, overpowering senders. Applications could possibly read receive buffers at a lower rate than compared to incoming data, of which the opposing sender needs to be notified of.

Figure 2.3 illustrates the structure of a send buffer used by TCP. The *send window* is defined as the bytes, identified by sequence numbers, that a sender is allowed to inject into the network. The left edge of the *send window* is determined by received acknowledgments (ACKs) to previously sent bytes. The right edge is determined by the sum of the position of the left edge and the window size field last signaled by the receiver. An arriving acknowledgment (ACK) and a steady (not shrinking) window size offered by the receiver lead to the window progressing ("sliding") to the right. A TCP receiver tells its peer about its *receive window* through the previously mentioned *window field*, a 16 bit field in the TCP header (possibly extended to 30 bit using the window

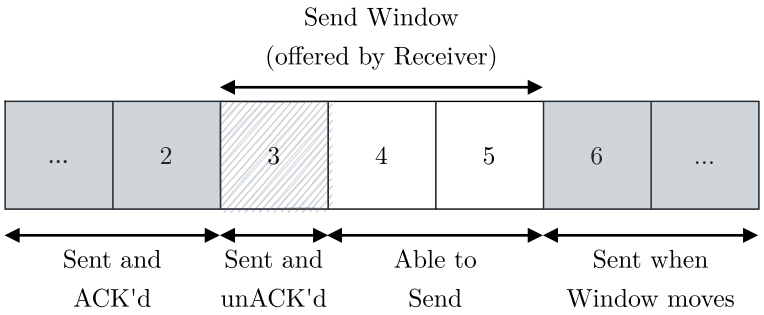


FIGURE 2.3: A simplified TCP sender sliding window structure, based on [12, Section 3.2]. It stores what sequence numbers have been sent, ACK'd or are able to be sent.

scaling TCP option [17, Section 2]). As every TCP segment contains an *acknowledgment field* and a *window field* [12, Section 3.1], we can potentially adjust the send window structure at every incoming segment.

An important metric for finding an optimal window size of a connection is the bandwidth-delay product (BDP), described in Equation (2.1). It is the product of a link's end-to-end delay called RTT, and a link's bottleneck bandwidth (BB), which is the bandwidth enforced by a path's most limited machine in form of either a host or a routing and/or switching device [18, Section 3.3.1].

$$\text{BDP (bit)} = \text{RTT (s)} \cdot \text{BB (bit/s)} \tag{2.1}$$

The BDP can be thought of as the number of bits that can be stored in the network path before the ACK of the first sent byte arrives back at the sender. A sender would start off sending bytes to the receiver, byte by byte filling up the *send window* part of bytes we can send out (Figure 2.3, white fields), until it is exhausted and flow control prevents us from delivering any more bytes to the sender. If the *send window* is equal to the BDP, exactly at this point the first ACK arrives and the *send window* starts to free up again, as the right and left edge would move on by one byte. Should the *send window* be smaller than the BDP, we would stop sending before the first ACK arrives as the left edge only then advances, and throughput (more details in Section 2.2.1) therefore is effectively limited by flow control.

As QUIC manages multiple streams of in-order data, it introduces flow control both per stream and across a whole connection, which possibly consists of multiple streams, to an endpoint. A TCP sender determines the current receive window size at the receiver by observing the *window field* in the header, where a value of 0 represents a full receive window. The *window field* then is a relative offset, as it is added to the value in the last received *acknowledgment field* (in terms of Figure 2.3: left edge+signaled window size =

Table 2.1: Mean goodput of different QUIC implementations. Measured on solana-uniswap hostpair (see Section 4.1) with 5 repetitions and a 10 GB filesize.

Implementation	Goodput (Mbit/s)	Lang.	Author
quiche	2006	Rust	Cloudflare
mvfst	891	C++	Facebook
lsquic	857	C	LightSpeed Tech
aioquic	55	Python	aiortc

right edge). QUIC instead uses an absolute byte offset, off which the highest received byte offset is subtracted to determine the current receive window. [9, Section 4.1]

$$\text{receive window size} = \text{maximum advertised byte offset} - \text{current highest byte offset} \tag{2.2}$$

This maximum byte offset is advertised in so-called `MAX(_STREAM)_DATA` frames, where stream keyword is included to indicate per-stream flow control updates and connection-wide updates otherwise. QUIC also defines the `(STREAM_)DATA_BLOCKED` frame, so a sender can let a receiver know when it would have data to send, but is limited by flow control. An important discussion is the frequency of said window updates. The standard merely mentions that implementations should decide how much and how often to advertise offsets [9, Section 4.2]. Work on the analysis of flow control behavior has been done, e.g., by Marx *et al.* [19, Section 3.1]. There ten IETF QUIC implementations have been compared, and while manual analysis does help for some time, there is no easy, automatic way of comparing flow control behavior. With the extension of the measurement framework, it is not far off to add the possibility for tests covering this need. As a side note, the problem of how frequently to send updates also applies to ACK frequency [9, Section 13.2.2], but first drafts [20] aim to address this problem for example by letting a peer advertise its desired ACK frequency.

2.1.5 Implementations

The QUIC working group maintains a list of all known QUIC implementations [21]. It currently contains 25 implementations in various programming languages for IETF QUIC, which is a direct result of QUICs placement as a userspace library, rather than as a kernel implementation like TCP. We focused on QUIC implementations with a high goodput performance, which are written in low-level languages like C or Rust. Table 2.1 shows the result of a simple goodput measurement on two directly connected bare-metal machines. It gives us a first hint that interpreted languages like Python underperform, and that *quiche*, written in Rust by Cloudflare, achieved more than double the mean goodput of the two C implementations, including one written by Facebook.

Additionally to pure performance, we also wanted to consider the current QUIC deployment situation. The QUIC internet deployment scan by Zirngibl *et al.* [14], the latest extensive scan to our knowledge, identified 23 M IPv4 and 300 k IPv6 QUIC endpoints, among other things by developing a ZMap module for QUIC detection. Considering their ZMap scan results, they could assign the largest group of IPv4 addresses to Cloudflare with more than half a million addresses. Google closely follows, in front of other entities like Akamai and Fastly. Cloudflare claims that they implement HTTP/3 support in their edge network entirely using their own implementation, *quiche* [22].

While the measurement framework we constructed does support any QUIC implementation as long as source code access is possible, we will conduct our featured measurements in this work with *quiche*, as we see its recent widespread deployment and relatively high goodput performance.

2.2 Metrics

In order to correctly interpret gathered data, it is essential to have precise metric definitions. The next sections revise the definitions of metrics used in our measurements.

2.2.1 Throughput and Goodput

In RFC5166 [23] three definitions for throughput are listed, one candidate fitting our context of a transport layer transfer using QUIC being throughput as the "link utilization or flow rate in bytes per second" [23, Section 2.1.1]. To add to the discussion, goodput is part of the throughput, but only includes useful traffic, which, for example, excludes duplicate segments that are transmitted on the wire, but do not aid in progressing a transfer, or segment, packet, and frame metadata in form of headers [23].

Another definition of throughput taken from the context of TCP is throughput as the "amount of data per unit of time that TCP transports when in the TCP Equilibrium state" from RFC6349 [18, Section 1]. The TCP Equilibrium state refers to the state TCP is in when increasing throughput in the congestion avoidance phase and reducing throughput upon a loss event, leading TCP to send at its maximum achievable throughput while effectively probing for changes in the network [18, Section 1.3]. When we talk about either throughput or goodput in the following paragraph, we refer to it as the amount of data in bits or bytes per second that QUIC achieves in its congestion control equilibrium state. We can apply the TCP Equilibrium state directly to QUIC, as QUIC implements congestion control similar to TCP but on a per-stream basis. When looking at QUIC single file transfers, as we are in this work, we effectively spectate a single QUIC stream, thus leading to a very similar congestion control situation to TCP.

$$\text{Throughput} = \frac{\text{Bits sent/received on wire (in bit)}}{\text{Time (in s)}} \tag{2.3}$$

$$\text{Goodput} = \frac{\text{Filesize (in bit)}}{\text{Transfer Duration (in s)}} \tag{2.4}$$

Equation (2.3) formulates throughput according to its definition as flow rate or link utilization. In this work, we use the unit of bits per second (bit/s). We capture it practically by gathering interface statistics on outgoing or incoming bits per second.

We calculate goodput in our experiments with the measurement framework according to Equation (2.4). "Filesize" references the size of the transmitted, randomly generated file. We capture the duration of a transfer by using timestamps before and after the client downloads the file.

`interop.py:_run_test()`:

```
1 testcase._start_time = datetime.now()
2 ... # Starting the transmission, blocking
3 testcase._end_time = datetime.now()
```

`testcases.py:check()`:

```
4 time = (self._end_time - self._start_time) / timedelta(seconds=1)
5 goodput = (8 * self.FILESIZE) / time / 10**6 # as mbps
```

Its important to note that we aim to capture the goodput of QUIC in the congestion control equilibrium state. This leads to several problems we have to keep in mind when designing measurements, as we only reach the congestion avoidance phase after the connection setup. This includes 1) the slow start phase that congestion control starts with and 2) the initial QUIC handshake and HTTP/3 request as addressed in Section 2.1.3. Therefore, it is important to make each transmission long enough to reduce the impact of the connection startup on the calculated goodput. For the measured throughput, it is enough to drop the first few throughput samples impacted by the connection startup.

2.2.2 CPU Utilization

We calculate the CPU utilization by adding up the CPU core usage (in %) of all cores the application uses with all spawned processes and threads. For a single-core application this results in a value of up to 100 %, but applications using multiple cores at once can exceed this value. For example, a multiprocessing application that fully utilizes four CPU cores has a CPU utilization value of 400 %. We capture the CPU utilization using the tool pidstat, which is further explained in Section 2.4.

2.3 Hardware Offloading

On most modern host systems, workstations and servers alike, NIC offloading techniques are utilized. NIC offloading in general describes the act of passing certain tasks to the NIC, which the host CPU would have to compute otherwise. One common theme of the following described techniques is the idea of letting the network stack process segments larger than the Path Maximum Transfer Unit (PMTU) (the maximum size in bytes of an IP packet for a path comprising of multiple links [24]) required by the network path. By doing this instead of processing multiple smaller segments, it reduces the amount of network stack operations applied to the packet as there are fewer packets to process in general and many packets of one transmission would follow similar processing anyway, e.g., given the same destination tuple in form of IP address and port number [25]. The following offloading functionalities are available in Linux systems:

Generic Segmentation Offloading (GSO) Used by both TCP and UDP. GSO is a pure software offload, meaning it can be used without offloading support in the NIC. It delays segmentation of segments as far as possible when sending, allowing TCP and UDP to build large packets. The data of these large `sk_buffs`, which represent segments in the kernel, is then split over multiple `sk_buffs` matching the PMTU. [25], [26]

Generic Receive Offloading (GRO) Used by both TCP and UDP. Being the receive-side equivalent of GSO, GRO is a software offload that should yield the reassembled, large segments split by GSO on the sender side. [26]

Large Receive Offloading (LRO) Used by TCP. LRO is an offload which aggregates packets of one TCP stream, reducing the number of packets processed by the CPU. Assembling can be done in the NIC driver or the NIC. [25], [27]

TCP Segmentation Offloading (TSO) Used by TCP. TSO is a hardware offload that lets TCP send large packets, which are then segmented into PMTU-fitting chunks and prepended with the necessary TCP, IP and link layer headers in the NIC. [25]

UDP Fragmentation Offload (UFO) A deprecated offload used by UDP, which allowed the NIC to fragment oversized UDP datagrams into multiple IPv4 fragments. [26] We did not find this offloading option on our testbed NICs, most likely due to this deprecation. Considering that QUIC makes use of underlying UDP, this would have been interesting to look at.

2.4 Pidstat

Pidstat [28] is a tool that is able to report statistics on Linux processes by using the process information pseudo-filesystem `/proc` [29]. It allows monitoring a process by PID or name over time by collecting process information in certain intervals. This also includes the activity of process threads. Useful metrics it can collect include I/O statistics, used CPU core and usage for every thread/process, the percentage of CPU time spent in userspace and kernel space, and the amount of voluntary and involuntary context switches.

To capture the CPU core usage of a process, e.g., quiche, using a string match on its name (`-G` flag), and all its threads (`-t` flag) in a one second intervall we can use the following.

```
1  $ pidstat -t -G quiche 1
2  ...
3  06:57:19 PM UID TGID  TID  %usr %system %guest %wait  %CPU CPU Command
4  06:57:20 PM   0 2095    - 51.00   47.00   0.00  0.00 98.00   9 quiche-server
5  06:57:20 PM   0    - 2095 51.00   47.00   0.00  0.00 98.00   9 |__quiche-server
6
7  06:57:20 PM UID TGID  TID  %usr %system %guest %wait  %CPU CPU Command
8  06:57:21 PM   0 2095    - 53.00   46.00   0.00  0.00 99.00   9 quiche-server
9  06:57:21 PM   0    - 2095 53.00   46.00   0.00  0.00 99.00   9 |__quiche-server
10
11 06:57:21 PM UID TGID  TID  %usr %system %guest %wait  %CPU CPU Command
12 06:57:22 PM   0 2095    - 23.00   20.00   0.00  0.00 43.00   9 quiche-server
13 06:57:22 PM   0    - 2095 23.00   20.00   0.00  0.00 43.00   9 |__quiche-server
14
15 Average:    UID TGID  TID  %usr %system %guest %wait  %CPU CPU Command
16 Average:      0 2095    - 43.40   44.62   0.00  0.01 88.02   - quiche-server
17 Average:      0    - 2095 43.40   44.62   0.00  0.01 88.02   - |__quiche-server
```

Chapter 3

Measurement Framework Design

A major tenet of this work is easy and fast reproducibility of measured results. This is achieved by a measurement framework that fixates used client and server implementations, the state of involved hosts, definitions of the measurements themselves, and all achieved results. To facilitate these requirements in a structured way, and without reinventing the wheel, the framework is based on the QUIC Interop Runner [8].

The following sections aim to explain the design of the measurement framework. This includes the question of why this particular project is a good fit as a starting point as well the question of how the framework and its components are structured. Further, it will be also be shown in what unified way QUIC implementations are added and defined in the measurement framework, and how settings in QUIC and the network environment are propagated to the test nodes.

3.1 QUIC Interop Runner as a Baseline

With the design decision of leaving QUIC's code out of the kernel, and supplying it through userspace libraries it seems to incentivize competing QUIC implementations [21]. Concrete examples (not necessarily all for the same programming language, but to illustrate that different actors get involved) include Cloudflare's *quiche* [30], LiteSpeed Tech's *lsquic* [31], Facebook's *mvfst* [32] or Microsoft's msquic [33].

The QUIC Interop Runner solves the problem of unknown interoperability by testing implementations against each other. It thus defines a basic framework for defining certain QUIC implementations, and running tests in a containerized environment. Figure 3.1

Figure 3.1: An interactive matrix with all test results of each QUIC client-server combination on the Interop Runner project website [34].

shows an example output of the result matrix that emerges when every implementation's server is observed in different test cases against each implementation's client.

Our measurement framework used this as a starting point but changed several concepts to better fit our measurement needs.

The QUIC Interop Runner executes its tests using a containerized environment using docker [35]. Instead of executing tests in process-virtualized environments, the measurements of our framework are run directly on bare hardware. This should more closely replicate a regular, real-world use case of QUIC, as process virtualization does impact application performance [36]. It also reduces the number of variables when trying to understand the behavior of QUIC implementations.
While container images also have the advantage of a packaged ecosystem regarding the building and running of applications, Section 3.4 shows how we solved the problem of building and adding implementations to the measurement framework in a structured way.

The QUIC Interop Runner executes both client and server on the same machine. Our framework adaptation is able to execute QUIC client and server implementations on physically separate machines.

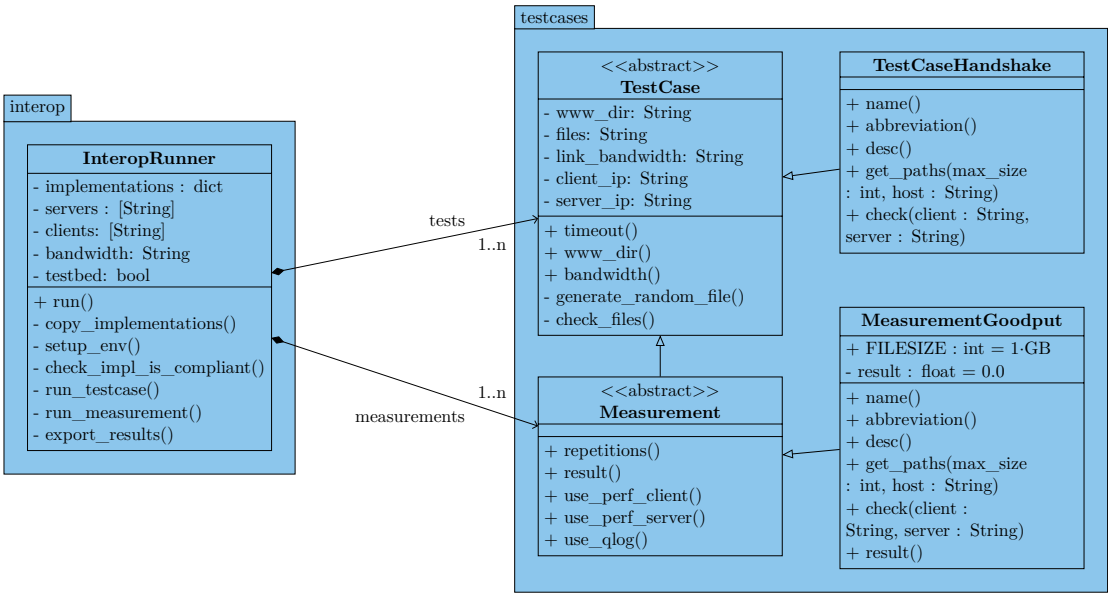


Figure 3.2: Measurement framework class structure. Repetition of similar testcases and measurement classes have been omitted.

3.2 Class Structure

As the framework is based on the QUIC Interop Runner, the class structure is very similar to the original project. Figure 3.2 shows the UML class diagram of the measurement framework.

InteropRunner is the central class in the structure as it performs the tests by iterating on all defined server and client implementations, and orchestrates the tests for each wanted test case.

The test definitions follow a hierarchical inheritance structure. First, an important distinction: a "testcase" is a test that results can either succeed or fail, e.g., did the handshake succeed or not, whereas a "measurement" is a test that can have a range of numbers as a result, e.g., achieved goodput.
TestCase is a parent class that defines attributes and methods that are useful to any testcase or measurement. Testcases, like a handshake pass/fail test, directly inherit from this class. The **Measurement** class is a parent class for all measurements and also inherits from **TestCase**. In addition to the capabilities of **TestCase**, it features measurement-specific code like the number of repetitions a test should perform or whether a measurement should be profiled using perf. All measurements inherit from this measurement parent class. A concrete example would be the **Measurement-**

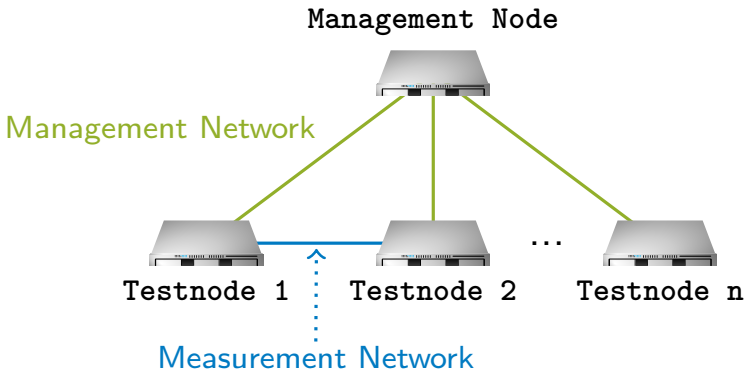


Figure 3.3: The testbed's network topology, showing one management node and a number of test nodes, the latter of which are orchestrated using Plain Orchestrating Service (POS).

Goodput class. All testcases and measurements wanted by the user are stored in the `measurements` list of the **InteropRunner** class.

3.3 Incorporation Into the Existing Testbed

As all measurements are conducted in the chair's testbed, the measurement framework naturally incorporates the testbed's orchestration system to solve essential tasks like the creation of files to transmit, the environment setup for QUIC implementation executables or changing operating system settings according to the user's needs.

The orchestration system of the testbed is Plain Orchestrating Service (POS) [37], a system which allows running experiments off of a management node, onto test nodes, which are bare metal machines and free of virtualization technologies. This is accomplished by leveraging techniques like Intelligent Platform Management Interface (IPMI) and Preboot Execution Environment (PXE) in combination with live Operating System (OS) images. POS offers a Representational State Transfer (REST) Appliation Programming Interface (API) which can be accessed using the python library *poslib* or the command line interface *pos cli* to start orchestrating test nodes from the management node.

Figure 3.3 illustrates the network topology of the testbed. One management node is connected to every test node to 1) orchestrate it using POS and 2) allow manual access to nodes using ssh. Pairs or groups of test nodes are interconnected, thus providing links where network measurements can be conducted on.

Due to the existing hierarchy in the testbed, the measurement framework should reside and be executed on the management node and orchestrate test nodes using the manage-

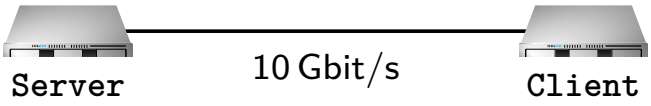


Figure 3.4: Testbed Topology used in measurements resulting from the measurement framework design, explained in Section 3.3.

ment node's POS API. As the framework is implemented in the python programming language it easily integrates POS using the python library *poslib*.

As QUIC is a transport layer protocol, it is located directly and only on end hosts involved in data transmissions over a network. Equipped with this fact, the minimal requirements for conducting measurements with QUIC are two machines with a link in-between. Figure 3.4 illustrates the network topology used in our measurements.

3.4 Standardizing Implementation Definition

As the QUIC Interop Runner makes use of process virtualization using docker containers, it has a standardized way of adding, storing and executing implementations by building implementation-specific container images by the implementation developers. In this spirit of providing a common way of storing and executing client and server implementations, we created a way to define implementation builds using pipelines and a unified server/client execution using run scripts, stored in version control.

The following components are required to add an implementation to the measurement framework:

`build.sh` Defines how the implementation is built in the CI/CD pipeline of the repository. Afterward the implementation is provided as a build artefact which the measurement framework can download and deploy on the test nodes of the testbed.

`run-client.sh` The run script the framework uses to execute the QUIC client of the implementation. It is a shell script that receives the parameters for the client in form of environment variables. The variables it has to handle are directory paths to save logs at (`SSLKEYLOGFILE`, `QLOGDIR`, `LOGS`), the kind of testcase or measurement that is being executed (`TESTCASE`) and instructions for the client execution, like the request in URL form (`REQUESTS`) and where to save the downloaded file (`DOWNLOADS`).

`run-server.sh` Analog to the client, the server receives its test settings through environment variables. Apart from the log-specific variables and the current testcase

it has to parse the server's root directory to hand files out from (`WWW`), the directory where certificates are stored (`CERTS`) and where to listen at (`IP`, `PORT`) to the server executable.

`setup-env.sh` This script fetches the requirements for an implementation. One example use case would be to install needed python libraries using pip for a python implementation.

Using this structure the measurement framework can execute server and client in a standardized way using environment variables. It does not matter what command line flags the QUIC implementation developers use in their implementation as the run scripts parse the test settings to the correct flags for the executable.

3.5 Supported QUIC and Environment Settings

As there are different settings applicable to QUIC, both on the QUIC libraries itselves, to the operating system, as well as to the surrounding network environment, we split up the placement of fine-tuning options in the following way.

The currently supported QUIC specific settings are applied directly in the client and server runscript files, introduced in Section 3.4. This includes options for the specific congestion control algorithm, or per-stream and connection-wide QUIC flow control limits. This approach requires command-line flags for each of these settings in the server and client executable, which may need to be patched in should the library not deliver a suitable example implementation. These settings can then be changed using command-line flags in the measurement framework execution, otherwise default settings will be applied. OS settings like TCP and UDP buffer sizes are applied before each measurement by accessing the test nodes directly, and should also be offered as command-line arguments. Network settings that can be applied at each host are solved in the same way. Intermediary hardware like routers and switches are currently not supported due to the structure of the current testbed, which only allows for direct connections between hosts. While this does limit network settings to solutions applicable to hosts directly, e.g., manipulating link bandwidths in software with Linux traffic control, support for network hardware would be possible if ssh is supported.

Currently implemented settings are the following.

- QUIC per-stream and connection-wide send and receive buffers
- link bandwidth using Linux traffic control
- AES hardware offloading

Implementing additional settings, especially when they cover niche settings like only enabling one single NIC offloading setting, takes effort and time to test and debug on the side of maintainers. To allow users to quickly and flexibly implement additional settings, we included the feature of flexible, generic scripts that can be given to the measurement framework. These can then be executed right before and/or right after a transmission to set and remove any settings a user might need. To allow these scripts to be flexible and produce logged results, we export variables like the current log directory on the machines, or information on the QUIC implementation like what role (client, server) it currently fulfills. This way scripts can act on specific conditions, like running a command on the QUIC server-side only and dump any output directly into the log location of the current measurement execution.

Chapter 4

Evaluation

4.1 Testbed Introduction

This section aims to give an introduction to the measurement methodology is conducted in terms of setup like the testbed and used machines, and to give a first observation how QUIC transfer performance grows or shrinks according to the hardware capabilities of different machines.

For the QUIC implementation, we settled for *quiche* due to reasons explained in Section 2.1.5.

For this measurement, we aim to measure QUIC's goodput of different machine pairs. In our case, goodput is defined as the amount of time it takes to transfer a file of a certain size, as explained in more detail in Section 2.2.1. Table 4.1 lists the different machines used and their corresponding hardware capabilities. Figure 4.1 shows the measured results. For each host pair, 20 goodput samples were captured, except for cesis-nida, where we increased the sample size to 30, due to a limited filesystem size, to counterbalance a shorter transfer duration and increase statistical significance.

Most further experiments were conducted on the solana-uniswap host pair.

4.2 Baseline

As an initial measurement, we establish a baseline, to which we can compare further results. As metrics, we chose 1) the goodput that QUIC is able to achieve and 2) the respective CPU utilization of QUIC server and client implementation. The following detailed measurement description is applicable to subsequent measurements also, unless

Table 4.1: Testbed node hardware specifications.

	Solana, Uniswap	Litecoin, Litecoincash	Cesis, Nida	Elva	Valga
CPU Model	Intel Xeon E5-1650 v3 @ 3.50 GHz	Intel Xeon D-1518 @ 2.20 GHz	Intel Xeon E5-2640 v2 @ 2.00 GHz	Intel Xeon E5-2620 v3 @ 2.40 GHz	Intel Xeon E5-2630 v4 @ 2.20 GHz
Cores per Socket	6	4	8	6	10
Threads per Core	2	2	2	2	2
RAM	64 GB	32 GB	32, 16 GB	32 GB	128 GB
AES Offloading	yes	yes	yes	yes	yes

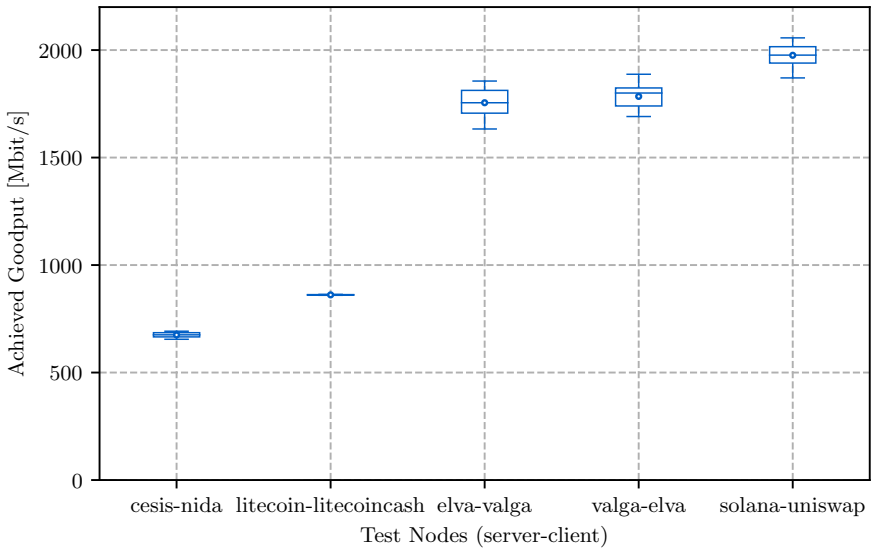


Figure 4.1: Hardware host comparison

otherwise noted. The used QUIC implementation is *quiche*, running on client- and server-side, with TCP Cubic as the congestion control algorithm. To identify how QUIC performs at different link speeds with these two metrics, we aim to consecutively increase the link bandwidth from 100 Mbit/s up to 3 Gbit/s by applying the Linux traffic control tool [38] to the server NIC. The network topology consists of two hosts connected by a single link, visualized in Figure 3.4. For each of the increasing link rate values, 20 measurement repetitions have been conducted, each lasting longer than 60 s to reduce the impact of the connection setup, as explained in Section 2.2.1. Furthermore, all network offloading has been deactivated. Each repetition resulted in one goodput value, calculated as explained in Section 2.2.1, and several CPU utilization data points, captured using pidstat [28], pinned to the *quiche* process with a time interval of one

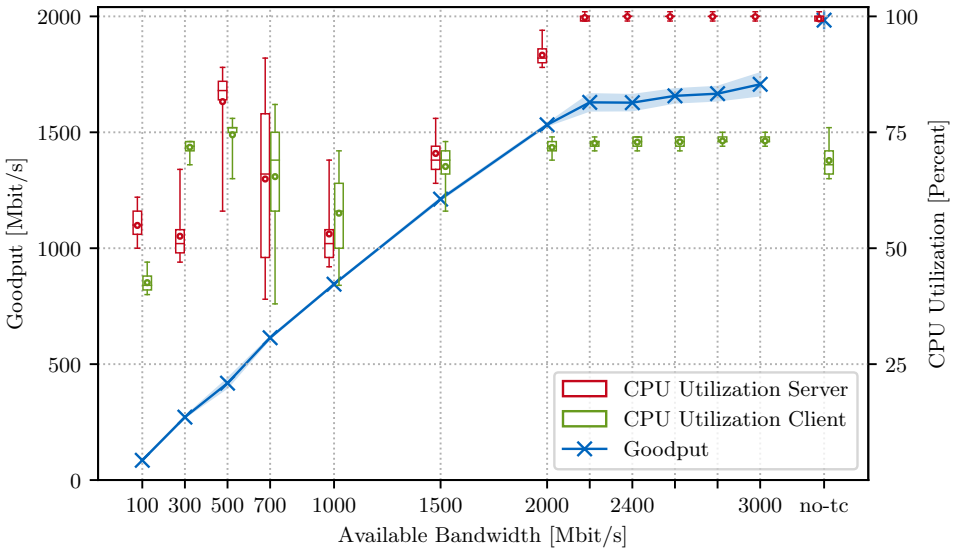


Figure 4.2: Goodput and CPU utilization of QUIC server and client implementation *quiche* on ascending link bandwidth values. Link bandwidth values have been emulated using linux tc tbf [38] on the server NIC.

second. Using this data, CPU utilization is then calculated as explained in Section 2.2.2. The plotted goodput data points are the arithmetic mean of those 20 repetitions at each set available bandwidth value, CPU utilization values of all repetitions of one set available bandwidth value have been fed into a boxplot, with each outer whisker ending at the 5th and 95th percentile, the box edges and middle representing the 25th, 75th and 50th percentile respectively, and the dots representing the arithmetic mean. The lighty shaded area around the goodput marks the range of the goodput standard deviation.

Figure 4.2 shows the results of this baseline measurement. We can observe that the achieved goodput of quiche keeps rising with link bandwidth increase, until the linear growth stops at around 2000 Mbit/s and achieved goodput settles at around 1650 Mbit/s to 1700 Mbit/s.

We also added measurements without Linux traffic control enabled, which can be seen at the “no-tc” label. As its mean goodput is higher than the limit reached with tc, this shows the cost associated with activated traffic control, in this case on the QUIC server.

To look at the linear growth up to a level of the goodput data points more closely, we take a look at the link utilization, which we define as the ratio of achieved goodput to available bandwidth. By calculating it for each available bandwidth value, we can observe the results in Table 4.2. At available bandwidths below the maximum achieved

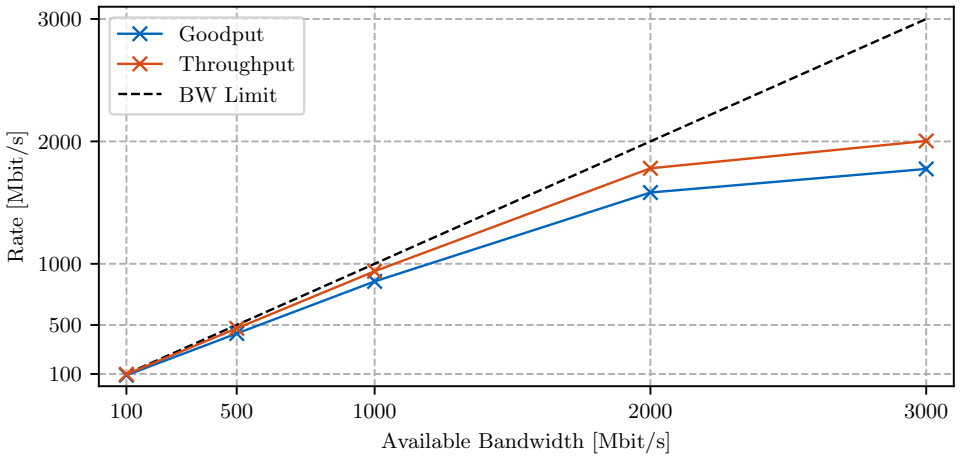


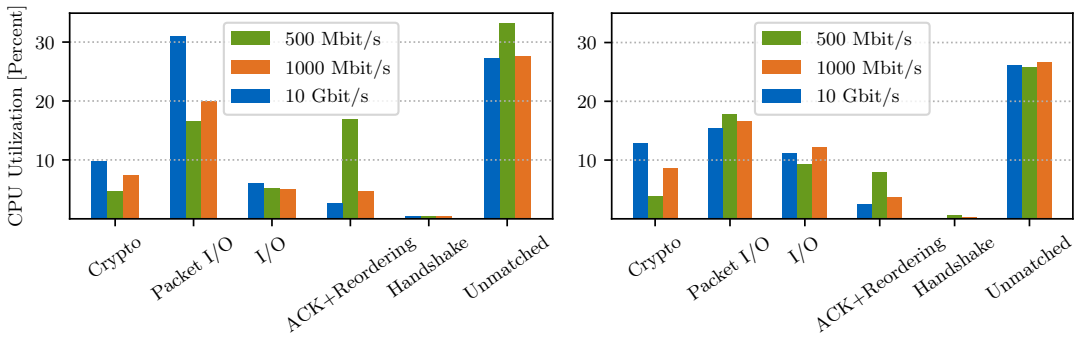
Figure 4.3: Goodput and throughput on ascending link bandwidth values. Link bandwidth values have been emulated using linux tc tbf [38] on the server NIC.

Table 4.2: Link utilization values

Link Bandwidth	Link Utilization
100 Mbit/s	85 %
500 Mbit/s	80 %
1000 Mbit/s	84 %
1500 Mbit/s	80 %
2000 Mbit/s	77 %
2200 Mbit/s	74 %
2400 Mbit/s	67 %
2600 Mbit/s	63 %

goodput, QUIC utilized the link around 80-85 %. Upon reaching the maximum, this ratio keeps dropping as the goodput does not rise along with the available bandwidth.

QUIC is able to utilize the link up to 80-85 %, even at 100 Mbit/s, due to the key distinction of throughput and goodput explained in Section 2.2.1. The underlying NIC may actually send the packets at the set available bandwidth rate, aka achieve a throughput of the link bandwidth rate, however as we effectively calculate the goodput, by looking at how long it takes to transfer a file of a certain size, which includes adding metadata to the segments, packets, and frames, as well as the time it takes to transmit the bits of our metadata, we end up with a link utilization value below 100 %. Figure 4.3 illustrates this difference. We measured the goodput as explained above, as well as the throughput by capturing the outgoing traffic of the server NIC facing the client. The measurement setup and methodology in terms of measurement repetitions and transfer duration is



(a) Server (b) Client

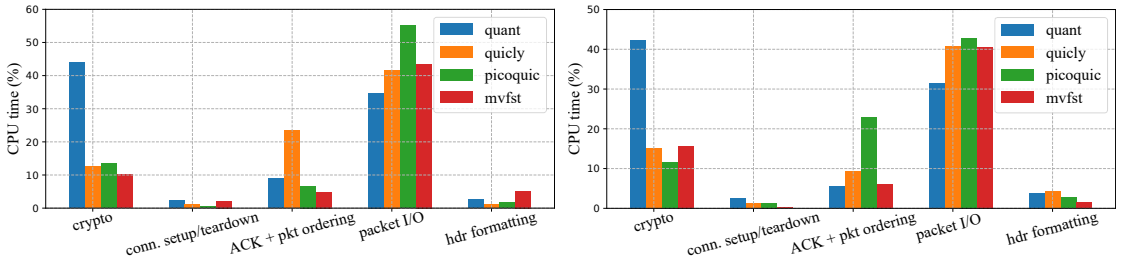
Figure 4.4: CPU utilization of different QUIC components for server and client at different link bandwidths. 10 Gbit/s represents data without activated Linux traffic control.

identical to the explanation before. It can be observed that the traffic leaving the NIC is higher than the calculated goodput, which confirms our goodput calculation in Section 2.2.1 of goodput as a subset of the throughput. This means we can conclude that QUIC was able to fully utilize lower link bandwidths, up to the ceiling when the CPU core of the server in the transmission was fully utilized.

Returning to Figure 4.2, we can also observe that the CPU utilization on the server and client rises with increasing link bandwidth rates. It is important to note that *quiche* client and server implementations run as single-cored processes, each with one thread for the transmission. For available bandwidth values up to 2000 Mbit/s, the server CPU core *quiche* ran on was not fully saturated with ascending arithmetic mean values of 50 to 70 %, until 1500 Mbit/s and around 90 % at 2000 Mbit/s. CPU utilization values above 2000 Mbit/s cap out the server CPU core at 100 %. The quiche client CPU core was maximally utilized around 75 % at bandwidth values above 2000 Mbit/s. The fully utilized server CPU core indicates that the *quiche* goodput stagnation around 1700 Mbit/s is due to a server CPU bottleneck.

Further, we can observe highly varying CPU utilization values around 500 Mbit/s, both on client and server. Due to the first indication of unusually high usage at this value, higher than the utilization at 1000 Mbit/s, where we would expect higher values than at 500 Mbit/s, we added more available bandwidth points around 500 Mbit/s. The available bandwidth values at 300 and 700 Mbit/s too indicate a pattern of increased CPU utilization on the server and client side. The following paragraph aims to add more information to this issue.

Figure 4.4 displays the CPU utilization of the different functions associated with different QUIC protocol components. We measured them with a ≈ 60 s transmission duration



(a) Server (b) Client

Figure 4.5: Yang *et al.* [5] QUIC protocol component CPU usage for *quant*, *quicly*, *picoquic* and *mvfst*, directly used from [5].

using *quiche*, while capturing system activity using perf. The resulting call graph (a chain of function names that have been executed) was matched to a regex list, and matches to a certain component were counted. The plotted percentage numbers are relative to the whole system activity to make results comparable, as the general CPU utilization of the QUIC process does change as seen in Figure 4.2. We define the different components of QUIC as the following:

Crypto Functions that surround encryption, e.g., Advanced Encryption Standard (AES), ChaCha20, Authenticated Encryption with Associated Data (AEAD) or RSA.

Packet I/O Functions that deal with sending and receiving segments. Considering QUIC, these are UDP read and write operations.

I/O Filesystem I/O and Linux file descriptor activity.

ACK & Reordering Activity of data structures that deal with ACK tracking and segment reordering.

Handshake Functions dealing with connection establishment.

Unmatched All functions that could not be matched to one of the above.

We start off with QUIC on our 10 Gbit/s link, without Linux tc bandwidth limitations. On the server, the most predominant part of the CPU time was spent on packet I/O with 30 %. After it follow crypto functions with a CPU utilization of 10 %, and regular I/O operations with 5 %. ACK and reordering protocol components used the CPU for 2.5 % of the measurement. QUIC on the client had a slightly higher portion of crypto function CPU cost, but less cost associated with packet I/O, which was 15 % compared to the 30 % on the server. The client also used more CPU time on filesystem I/O with 10 %, due to having to write the transmitted file to the disk. The ACK and reordering cost is similar on server and client.

Yang *et al.* [5], visible in Figure 5.5, also created a breakdown of CPU usage of QUIC server and client for the QUIC implementations *quant*, *quicly*, *picoquic* and Facebook's *mvfst*, all written in C or C++. *quant* is not really comparable to the other implementation's numbers, as in their work it was configured in kernel-bypass mode using netmap [39], which changes the way packets are processed by for example reducing data copy costs to the kernel by sharing buffers. In this work we focus on implementations sending packets without any kernel-bypass techniques. The in-kernel implementations were shown to reach a throughput of around 500 Mbit/s, and 300 Mbit/s with *mvfst*. These implementations too had a CPU usage of crypto functions around 10 % to 13 % on the server and slightly elevated CPU usage of crypto functions on the client, compared to *quiche* on the 10 Gbit/s link. The packet I/O CPU usage of *quiche* is significantly lower, on both client and server. Overall, the CPU usage relation of the different QUIC components, dominated firstly by packet I/O and then crypto, is identical in our measurement compared to the results of Yang *et al.*

When reminding ourselves of the baseline Figure 4.2, we noticed that *quiche*, when being limited to the area of 500 Mbit/s, had a higher server and client CPU utilization than at later stages like 1000 Mbit/s. In Figure 5.4, we additionally show the QUIC component CPU utilization at the link bandwidths of 500 Mbit/s and 1000 Mbit/s. One anomaly is the ACK and reordering CPU usage of the server at 500 Mbit/s with 15 %. The `range_search()` function of the underlying *RangeSet* data structure, responsible for ACK range tracking and based on *BTreeMap*, was triggering a high CPU utilization at this link bandwidth value of 500 Mbit/s. For higher link bandwidths like 1000 Mbit/s, which *quiche* was also able to fully utilize, or even native 10 Gbit/s, the CPU cost associated with ACK tracking at the server lowered back down to 5 %. Without debugging *quiche* in detail at this link speed, we cannot state the definitive reason for this behavior at this point.

4.3 Comparison to TCP with TLS

As QUIC provides a secure, connection-oriented transport layer service, it is a direct competitor to the current, widely adopted TCP+TLS stack. The early development of QUIC was started by Google to improve user experience, particularly in terms of the metric page load time [40]. While this web metric is important, and research has been done on it specifically [3], [4], [7], [41], QUIC is also designed as a general-purpose transport layer protocol. It thus has to compete against TCP not only on the web, but also in situations with longer file transfers, which make for example the shorter

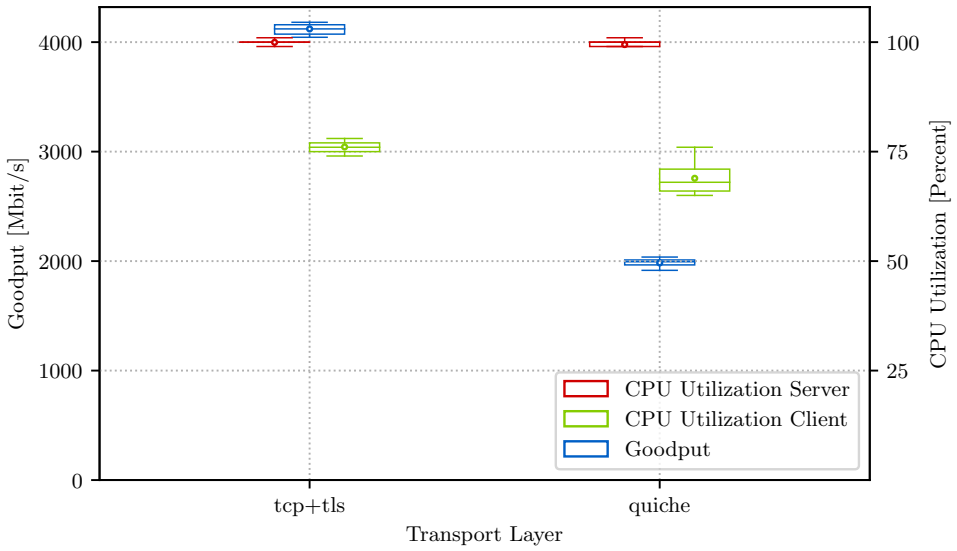


Figure 4.6: Goodput and CPU utilization of TCP with TLS and QUIC.

handshake a very small part of the overall transfer and puts more emphasis on its performance in the congestion control avoidance phase.

We aim to measure the goodput performance, as well as the CPU utilization, of QUIC compared to TCP with TLS on a 60 s+ transfer. We chose *nginx (v1.18.0, built with OpenSSL v1.1.1n)* and *wget (v1.21)* as the TCP webserver and client of choice. They negotiated an AES256 cipher using TLS 1.3, as did *quiche*. It can be noted that regarding the transport layer, it is less critical what TCP application is chosen as the socket interface and the underlying TCP implementation in the kernel are going to be identical for all applications. QUIC on the other hand is provided as a userspace library, on top of in-kernel UDP. TCP Cubic was the activated congestion control algorithm. Any kind of transport layer NIC offloading is once again disabled.

Figure 4.6 shows the result of this measurement. With values closely above 4000 Mbit/s TCP achieved more than double the goodput of *quiche*, which achieved about 2000 Mbit/s. It was effectively able to transfer the same file as *quiche* in half the time. Identical to *quiche*, the server CPU core that *nginx* ran on bottlenecks the transfer.

The fact that TCP with TLS uses the CPU more efficiently is not groundbreaking news. The google paper of Langley *et al.* [42, Section 6.7] observed for Google QUIC, a pre-IETF-standard version, a 3.5 times higher server CPU utilization at one point, and reduced it to be 2 times higher after optimizations. They found a high cost of packet I/O, and so did we. A reason for this cost based on the architecture of QUIC can be made

Table 4.3: Supplementary information on mean spent time in kernel- and userspace, and on context switches for QUIC and TCP with TLS. Client numbers are not true averages but handpicked near-median samples due to missing tool output. Captured using pidstat (see Section 2.4).

	Kernelspace Time of Total (in %)	Userspace Time of Total (in %)
TCP+TLS server	71	29
QUIC server	52	48
TCP+TLS client	51	49
QUIC client	42	58

by recognizing QUICs position in the userspace, repeatedly triggering write and read system calls to the UDP socket to send and receive small enough, MTU sized segments. Each system call leads to a switch from user mode to kernel mode, giving control to the system call handler [43, Section 1.6 "System Call"], which proceeds to copy the data from the userspace buffer to kernelspace [44] before handing it off to its journey down the protocol stack, ending at a Direct Memory Access (DMA) transfer to the NIC. Each process switch, or context switch, to the system call handler leads to a swap of register content, of the memory map in the MMU and to memory cache invalidations, leading it to be reloaded when entering and exiting the kernel [43, Section 2.4 "Scheduling"]. A lot of context switches per second can thus require additional CPU time. TCP should require less such system calls, as it receives a longer bytestream, which it slices into valid segments itself, in kernelspace. Another argument consists in the fact, that ACKs in QUIC are encrypted and have to be decrypted to be read, while TCP employs cleartext acknowledgments [45, Section 7]. Table 4.3 provides supplementary information on the activity of the TCP and QUIC servers, where we see that the QUIC server spent more of its time in the userspace compared to TCP.

4.4 Packet Offloading

While the goal of this work is not to extensively analyze the reduction of CPU load using techniques like custom NIC offloading for QUIC, for example done by Yang *et al.* [5], we nevertheless want to show the impact of already existing NIC settings that suggest a CPU load reduction for host systems. Section 2.3 introduced all network offloading options seen in this section, options applicable to both TCP and/or UDP. As QUIC utilizes UDP to encapsulate and send its own QUIC segments, as explained in Section 2.1.1, we aim to show the impact of offloading techniques applicable to UDP and therefore, to QUIC.

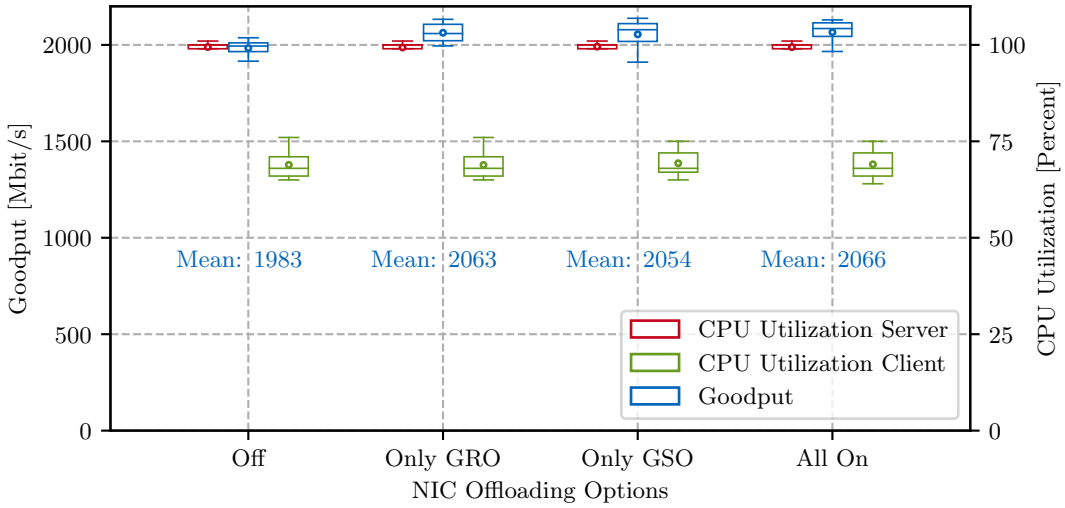


Figure 4.7: Goodput and CPU utilization of QUIC when using different, existing offloading options.

Following previous measurement designs, we capture the performance metric goodput (see Section 2.2.1) under different activated offloading options. We additionally track the CPU utilization of the core that *quiche* ran on for both server and client.

Figure 4.7 shows the resulting impact of different UDP offloading options on QUIC. Goodput mean values have been explicitly noted on the plot itself. When all offloading options are deactivated, we can see *quiche* achieving a mean goodput of 1983 Mbit/s. As seen in previous measurements, the server CPU core *quiche* was executed on reaches a utilization of 100 %, while the *quiche* client CPU core is utilized for around 70 %.

Then we activate only Generic Receive Offloading (GRO), a receive-side software offload on both server and client at once. We can observe a slight increase in the mean goodput by 80 Mbit/s, of our 20 goodput measurements in total for each offloading option. The CPU utilization on server and client side stays constant while observing this slight jump in mean goodput. It seems unusual to see a higher goodput when activating an offload supporting the processing of arriving segments, as we know that the bottleneck resides in sending data segments at the server, indicated by a fully utililized CPU core of the server in the "Off" setting, and from the baseline plot in Section 4.2. One hypothesis could be that GRO reduces the strain on the server CPUs processing of the arriving UDP QUIC ACK segments.

Next, we solely activated GSO, a software offload impacting segmentation when sending segments. Similar to GRO, GSO increased the mean goodput slightly to 2054 Mbit/s. It seems plausible that a sender-limited transfer, indicated by a once again fully saturated

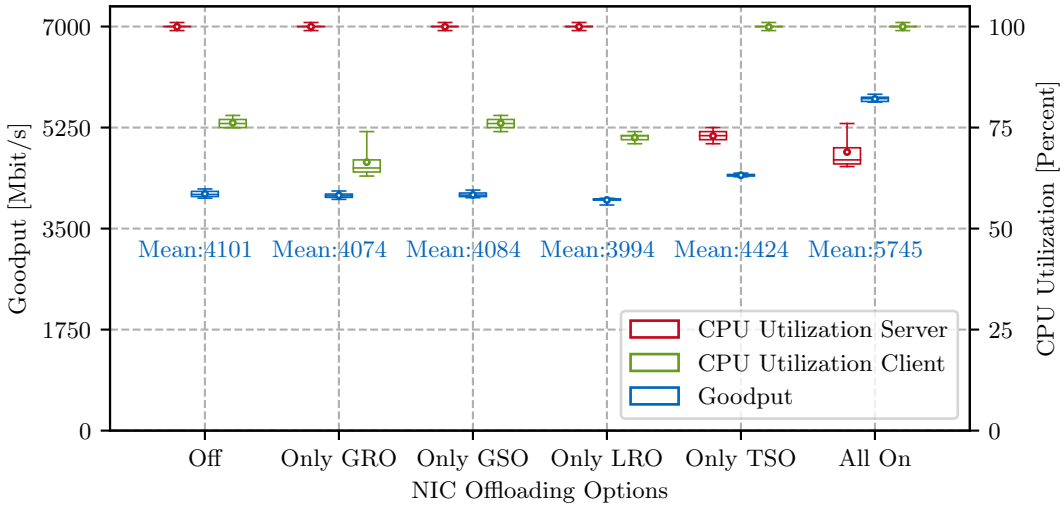


Figure 4.8: Goodput and CPU utilization of TCP with TLS when using different, existing NIC offloading options.

CPU core on the QUIC server, is able to increase goodput by reducing the cost associated with sending segments. Upon activating both GRO and GSO, we see no further rise in goodput, compared to the previous, solo activation of each offloading option. It merely achieved a mean goodput of 2066 Mbit/s, an increase of similar order of significance, although both solo options increased the mean goodput, but a higher first quartile in the goodput samples indicates a goodput increase in more samples than when using GRO or GSO on their own. Nevertheless, the total increase is relatively low and seems to lie in the fact that both GRO and GSO are software offloads, which do reduce the burden on the CPU but do not relieve the CPU of the processing altogether.

To be able to better classify the impact of existing offloading techniques, we compare the previous results to the impact on TCP with TLS transmissions. TLS is added to have a better comparison, as QUIC enforces the usage of encryption with TLS 1.3. Figure 4.8 shows the gathered results. We already saw the result of the "Off" setting in the previous chapter – the server CPU core *nginx* ran on is fully saturated, and the mean goodput settles around 4000 Mbit/s. GSO, the software segmentation offload, seems to have little to no impact on TCP. On the other hand its receive-side equivalent, GRO, reduced the mean client CPU core utilization by a significant amount by roughly 10 %. Also, the majority of client CPU utilization samples is grouped around the new mean CPU utilization value, indicated by the first and third quartile borders of the boxplot. This is a major reduction compared to the impact GRO had on QUIC.

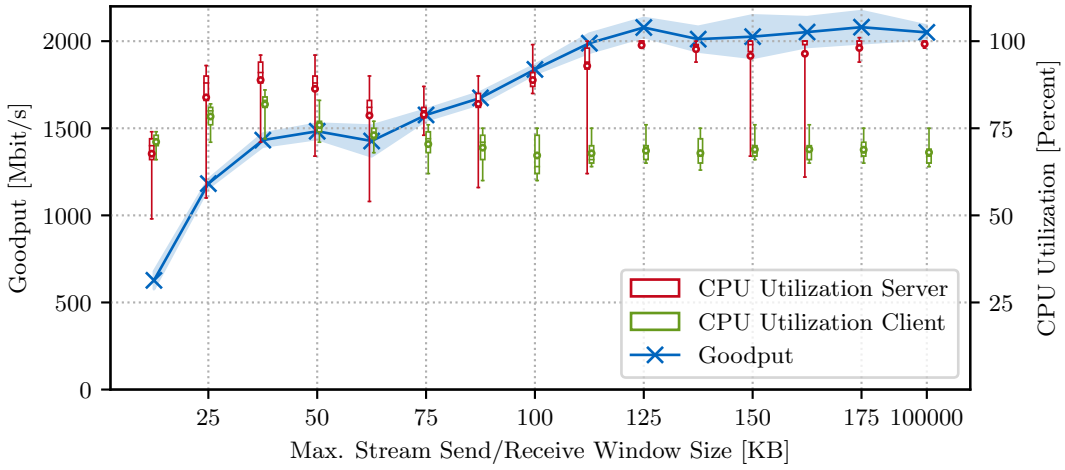


Figure 4.9: Goodput and CPU utiliation of QUIC with incrementing, per-stream send and receive window sizes.

For complete coverage of default network offloading possibilities, we also included the TCP-specific techniques LRO and TSO, first introduced in Section 2.3. The usage of LRO exclusively, mysteriously reduced the achieved mean goodput slightly, but also reduced the client CPU utilization, indicated by the mean and range of the first and third quartile box edges. TSO on the contrary is the first offloading option to increase the goodput of TCP by a noticeable amount of roughly 11 %. Additionally, the relationship of client and server CPU utilization shifted: the client CPU core is now fully utilized, and the mean server core utilization settles at around 72 %, leading to the client core bottlenecking the transfer now. This could be accounted to the fact that TSO is a hardware offload, reducing the CPU burden of sender-side segmentation, which is now handled by the NIC. The combination of all offloading techniques accessible to TCP yields an increased mean goodput of 40 %, at 5745 Mbit/s compared to deactivating all options ("Off"). This additional jump from TSO only to the activation of all options could be explained by a reduced CPU utilization on the receiving client-side through GRO and LRO.

4.5 Flow Control

Although QUIC's flow control is in principle similar to TCP's, it does employ changes, like in the protocol definition itself with a different protocol header structure compared to TCP, or in the way QUIC is delivered – while the IETF drafts define the base structure and functionality of QUIC, implementations do have room for adjustments in e.g., flow control receive window update frequency, as explained in Section 2.1.4. This

is quite a new situation, triggered by the design decision of QUIC as a userspace library, leading to a landscape of QUIC implementations, each with potentially different minor tuning in topics like flow control. This issue did not arise with TCP, supported by the fact that each operating system only has one implementation in the kernel.

We aim to observe the impact of incremented per-stream maximum send and receive windows on 1) the goodput and 2) the server and client CPU utilization. Due to our focus on single-stream transfers, we exclude the interplay of connection-wide flow control limits and per-stream flow control limits. Send and receive windows are set to a specific value both at once. For each window size value, 10 measurement repetitions were conducted, each resulting in one goodput value and several CPU utilization samples. All samples of goodput and CPU utilization belonging to one window size value then are aggregated and, each, fed into boxplots with the same edge definitions as in Section 4.2.

Figure 4.9 shows the changing goodput and CPU utilization on incremented window size, in form of both send and receive window. Starting off with a situation that is definitely not limited by flow control, by looking at a window size of 100 MB, or 100 000 kB on the far right on this x scale, we can see a goodput closely above 2000 Mbit/s as already known from our baseline measurement in Section 4.2. At 125 kB we can see it reaching our empirically determined goodput cap of 2000 Mbit/s for the first time. At window sizes below 125 kB, we see a reduced goodput, which rises from the lowest point, at 12.5 kB with a mean goodput of 600 Mbit/s, with each incremental window size increase. While the client CPU utilization stays steady at 70 %, the server CPU utilization is lower with reduced window sizes, e.g., with 70 % at a window size of 25 kB, and rises up to the usual 100 % starting at a window size of 125 kB.

At window sizes in the range of 25 kB to 62.5 kB the mean CPU utilization of client and server sees a slight increase, with a local peak at 37.5 kB. We suspect a data structure CPU usage increase like in Section 4.2, but cannot confirm it at this moment.

As the measurement framework implements the custom setting of window sizes, it is a step towards flow control comparisons, which, as stated in Section 2.1.4, can be different from QUIC implementation to QUIC implementation, and not regulated by a unifying RFC. Observing the component cost of ACK tracking and flow control under different window sizes could also form an additional, future experiment over different QUIC implementations.

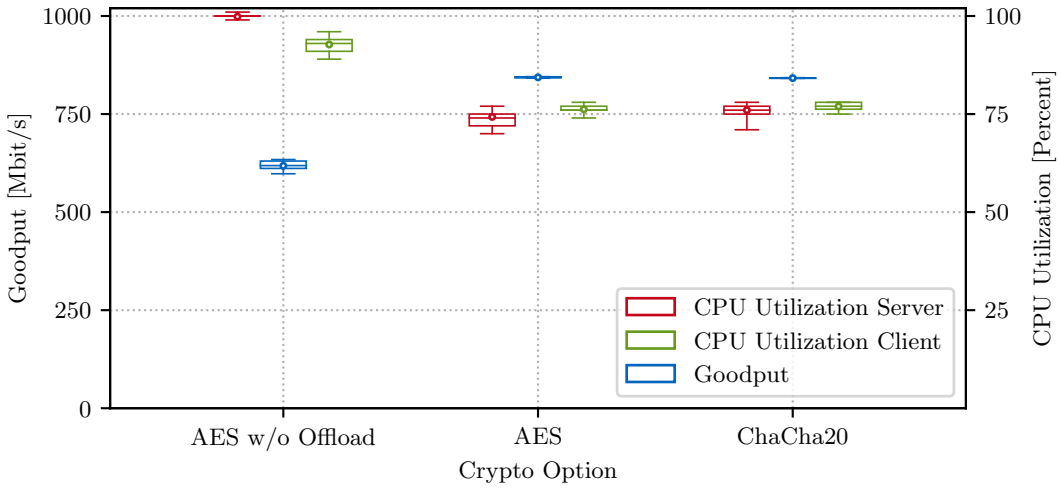


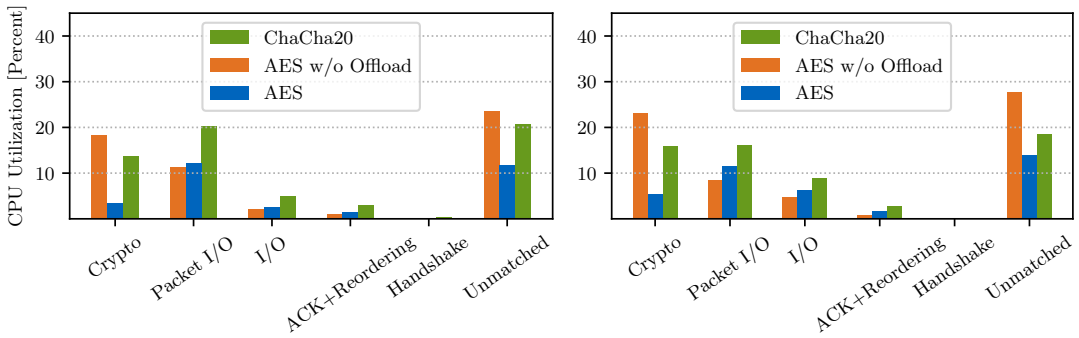
Figure 4.10: Goodput and CPU utilization of AES256 and ChaCha20 in QUIC.

4.6 Crypto

The TLS 1.3 specification RFC8446 [46] defines the list of required ciphers (`TLS_AES_128_GCM_SHA256`) and recommended ciphers (`TLS_AES_256_GCM_SHA384`, now abbreviated with AES256; `TLS_CHACHA20_POLY1305_SHA256`, now abbreviated with ChaCha20) ciphers used in QUIC.

We aim to measure the metrics 1) goodput and 2) CPU utilization of client and server of the ciphers AES256 and ChaCha20 available in *quiche*. Out of interest we additionally added the option of AES without hardware support. Instead of the usual host pair, we measured these results on the litecoin-litecoincash host pair described in Section 4.1. They have lower CPU frequencies, which should reduce the achieved goodput in the following results. For each cipher option, 10 measurement repetitions were conducted, each resulting in one goodput value (defined in Section 2.2.1) and several CPU utilization samples (described in Section 2.2.2). All samples of goodput and CPU utilization belonging to one cipher then are aggregated and, each, is fed into boxplots with the same edge definitions as in Section 4.2.

Figure 4.10 shows the result of this measurement. We can see similar mean goodput values for both AES256 and ChaCha20 at 850 Mbit/s, while AES256 without hardware support lags behind at 630 Mbit/s. The CPU utilization of server and client of AES256 and ChaCha20 reside at a similar level of 75 %. Considering the mean goodput, AES256 with hardware support and ChaCha20 achieve the same result.



(a) Server (b) Client

Figure 4.11: CPU utilization of QUIC components when using AES256 or ChaCha20.

We saw that the general CPU utilization of AES256 and ChaCha20 for server and client looked very similar, but this does not say anything about how each cipher option influences the time spent on cryptographic functions in QUIC. For each cipher option a single 60 s transmission using *quiche* has been conducted while capturing system activity using perf. The definition of each QUIC component has been explained in Section 4.2. A possible improvement would be to calculate a mean CPU utilization of each component by repeating this measurement multiple times and averaging the results, for even more reproducible results, nevertheless this also shows the relations of the different QUIC components.

Figure 4.11 shows the CPU utilization of each QUIC component for AES256, ChaCha20 and AES256 without hardware support. We can observe ChaCha20 and AES256 without hardware support spending a relatively large part of the CPU time on crypto functions, where AES256 without hardware support spends 5 %, on the server, and 8 %, on the client, more time on functions associated with cryptography than ChaCha20. ChaCha20 does spend 10 % more CPU time for packet I/O on the server compared to both AES256 options. AES256 **with** hardware support on the other hand only spends a minor fraction of its CPU time on crypto, with about 5 % on server and client. The reason for this is the hardware support embedded into modern CPUs, which allows the CPU to offload computation onto special-purpose hardware.

Chapter 5

Related Work

Measurement Framework. We based our measurement framework for reproducible QUIC measurements directly on the QUIC Interop Runner [8]. It is a framework in which different QUIC client and server implementations communicate with each other in various tests. Implementations are tested inside several layers of virtualization, as implementations are provided inside docker containers, with a network environment simulated using ns-3, all on the same machine. We changed the codebase to suit our need for bare metal performance, with it now being capable to orchestrate tests between machines in our testbed. We also added more options for fine-grained behavior of QUIC implementations, for example by letting the user set flow control variables. Lastly, we also implemented TCP with TLS into the framework.

QUIC Measurements. Yang *et al.* [5] compared the QUIC component cost of four different C/C++ implementations conforming to IETF draft version 27 in a single connection scenario on a 10 Gbit/s link. They found that the most CPU time was spent on packet I/O (35-55 %), followed by crypto operations (10-13 %). They also configured one implementation in kernel-bypass mode using netmap, which then reached a 10x higher average throughput compared to the rest. We noticed a very small filesize definition of 50 MB for their tests, which would lead to a transfer duration in the area of one second. This leads to the connection setup having a relatively large impact on the throughput result. While they did not perform direct comparisons to TCP, such a short transfer would seem to favor QUIC and its shorter handshake. It may be true that such filesizes are realistic on the web, but we argue that for a complete picture of a general-purpose transport layer protocol, QUIC also has to undergo comparisons in transfers that are more dictated by the congestion avoidance phase. In their conclusion, they mention that the performance impact of UDP Generic Segmentation Offloading

(GSO) was not able to be investigated, a topic which we could shed some light on. Measurements were performed using simple bash scripts. We implemented a more overarching measurement framework, which is able to conduct a large number of flexible, reproducible measurements on all QUIC implementations on bare metal. Additionally, they did mention that they wanted to focus on implementations written in a low-level language, but did not include the low-level implementation *quiche* by Cloudflare written in Rust, which according to internet scan results by Zirngibl *et al.* may have been the most widespread QUIC implementation at the creation of this work. We aim to base our featured measurements on it, also because it achieved the highest goodput in our tests.

Shreedhar *et al.* tested QUIC in several web use cases like page visits, cloud storage transfers, and video workloads on the public internet. They found out that QUIC achieved a higher mean throughput than TCP+TLS for smaller filesizes in Google cloud storage transfers ($< 20\,\text{MB}$) due to the shorter handshake highly impacting the transfer. For larger filesizes, TCP performed better, and in general, QUIC showed a higher CPU usage. Their used QUIC implementation was *lsquic*, which, while being written in C, in our tests did lag behind the goodput performance of *quiche* by Cloudflare. Measurements on the public internet are inherently not very reproducible, as the intermediary paths are not predictable, and also depend on the actual location. While QUIC's behavior in this black box may also have to be tested, we focus our efforts exclusively on measurements in testbeds, with clear variables. They also did show CPU utilization in form of flamegraphs, we took it one step further and present the CPU utilization of each component after assigning functions to specific QUIC functionality. This approach allows for easier, visual comparisons.

Chapter 6

Conclusion

In Chapter 1 we introduced the problem of little information on the performance of QUIC in settings where a user would likely turn to TCP, like longer file transfers. These are situations QUIC also has to face as a general-purpose transport layer protocol, although it was first introduced to improve user experience specifically on the web [40].

We saw that QUIC was able to fill lower bandwidths well, although it did show unexpectedly high CPU utilization at certain, low link bandwidth values. The limiting peer was the QUIC sender, while TCP transfers with NIC offloads proved the client to be the limiting factor. We learned, that TCP was more than twice as fast as QUIC in transfers in the one-minute range in terms of goodput, and three times as fast with default NIC offloading options for both QUIC and TCP. We identified packet I/O (30 %) to be the biggest factor in the server CPU usage of QUIC, most likely due to the context switching costs associated with frequent system calls, and the required memory-to-memory data copy from user to kernelspace. This was followed by the cost of cryptography operations (10 %) on the server. We argue that a transport layer user aiming to achieve high throughput on big transfers in terms of filesize is at the moment better advised to make use of TCP, until packet I/O cost reduction in form of segmentation offloads for QUIC are available. Work on segmentation or crypto offload is important and has been started, featuring the works of Yang *et al.* [5] for crypto offload and Hay *et al.* [1] for crypto and segmentation offload.

All these results with varying QUIC and OS variables were obtained in an automated and reproducible way, by extending the QUIC Interop Runner for use in measurements without virtualization, on physically distinct machines. As it also allows for direct comparisons between different implementations, future work could make use of its capabil-

ities to research the differences in implementations, that are not covered by a unifying draft, like flow control update frequency or the delay of acknowledgments. For work on these implementation differences we want to mention [19], [47], [48] for flow control and ACK frequency algorithm categorization and performance research.

Chapter A

Appendix

A.1 List of acronyms

TCP Transmission Control Protocol
UDP User Datagram Protocol
POS Plain Orchestrating Service
IPMI Intelligent Platform Management Interface
PXE Preboot Execution Environment
OS Operating System
REST Representational State Transfer
NIC Network Interface Card
API Appliation Programming Interface
ACK acknowledgment
BDP bandwidth-delay product
RTT round-trip time
BB bottleneck bandwidth
TLS Transport Layer Security
GRO Generic Receive Offloading
GSO Generic Segmentation Offloading
LRO Large Receive Offloading
TSO TCP Segmentation Offloading
UFO UDP Fragmentation Offload
PMTU Path Maximum Transfer Unit
AEAD Authenticated Encryption with Associated Data
AES Advanced Encryption Standard

TLS Transport Layer Security
DMA Direct Memory Access
SCTP Stream Control Transmission Protocol

Bibliography

[1] J. Hay, M. Machnikowski, G. Bowers, N. Wochtman, J. Muniak, and M. Deval, "Accelerating QUIC via Hardware Offloads through a Socket Interface", in *The Technical Conference on Linux Networking (Netdev)*, 2019.

[2] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC", *CoRR*, vol. abs/1906.07415, 2019. arXiv: 1906.07415. [Online]. Available: http://arxiv.org/abs/1906.07415.

[3] P. Megyesi, Z. Krämer, and S. Molnár, "How Quick is QUIC?", in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6. DOI: 10.1109/ICC.2016.7510788.

[4] S. Cook, B. Mathieu, P. Truong, and I. Hamchaoui, "QUIC: Better for What and for Whom?", in *2017 IEEE International Conference on Communications (ICC)*, 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7997281.

[5] 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.

[6] 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.

[7] 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.

[8] M. Seemann and J. Iyengar, "Automating quic interoperability testing", in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20, Virtual Event, USA: Association for Computing Machinery, 2020, 8–13, ISBN: 9781450380478. DOI: 10.1145/3405796.3405826. [Online]. Available: https://doi.org/10.1145/3405796.3405826.

[9] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC Editor, RFC 9000, 2021.

[10] M. Thompson 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.

[11] 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.

[12] J. Postel, "Transmission Control Protocol", RFC Editor, RFC 793, 1981. DOI: 10.17487/RFC0793.

[13] A. Joseph, T. Li, Z. He, Y. Cui, and L. Zhang, "A Comparison between SCTP and QUIC", Internet Engineering Task Force, Internet-Draft draft-joseph-quic-comparison-quic-sctp-00, Mar. 2018, Work in Progress, 24 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-joseph-quic-comparison-quic-sctp/00/.

[14] 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 21st ACM Internet Measurement Conference*, ser. IMC '21, Virtual Event: Association for Computing Machinery, 2021, 261–275, ISBN: 9781450391290. DOI: 10.1145/3487552.3487826. [Online]. Available: https://doi-org.eaccess.ub.tum.de/10.1145/3487552.3487826.

[15] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC Editor, RFC 7540, 2015. DOI: 10.17487/RFC7540.

[16] T. Dierks and E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, Aug. 2008. DOI: 10.17487/RFC5246. [Online]. Available: https://www.rfc-editor.org/info/rfc5246.

[17] V. Jacobson, R. Braden, and D. Borman, *TCP Extensions for High Performance*, RFC 1323, May 1992. DOI: 10.17487/RFC1323. [Online]. Available: https://www.rfc-editor.org/rfc/rfc1323.html.

[18] R. Schrage, G. Forget, R. Geib, and B. Constantine, *Framework for TCP Throughput Testing*, RFC 6349, Aug. 2011. DOI: 10.17487/RFC6349. [Online]. Available: https://www.rfc-editor.org/info/rfc6349.

[19] R. Marx, J. Herbots, W. Lamotte, and P. Quax, "Same standards, different decisions: A study of quic and http/3 implementation diversity", in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, ser. EPIQ '20, Virtual Event, USA: Association for Computing Machinery, 2020, 14–20, ISBN: 9781450380478. DOI: 10.1145/3405796.3405828. [Online]. Available: https://doi-org.eaccess.ub.tum.de/10.1145/3405796.3405828.

[20] 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/.

[21] *QUIC Working Group Implementations Overview*, https://github.com/quicwg/base-drafts/wiki/Implementations.

[22] A. Ghedini and R. Lalkaka. "HTTP/3: the past, the present, and the future". Accessed: 2022-10-14. (), [Online]. Available: https://blog.cloudflare.com/http3-the-past-present-and-future/.

[23] S. Floyd, *Metrics for the Evaluation of Congestion Control Mechanisms*, RFC 5166, Mar. 2008. DOI: 10.17487/RFC5166. [Online]. Available: https://www.rfc-editor.org/info/rfc5166.

[24] G. Fairhurst, T. Jones, M. Tüxen, I. Rüngeler, and T. Völker, *Packetization Layer Path MTU Discovery for Datagram Transports*, RFC 8899, Sep. 2020. DOI: 10.17487/RFC8899. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8899.

[25] R. Hat. "NIC Offloads". Accessed: 2022-10-09. (), [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-nic-offloads.

[26] kernel.org. "Segmentation Offloads". Accessed: 2022-10-09. (), [Online]. Available: https://www.kernel.org/doc/html/latest/networking/segmentation-offloads.html.

[27] J. Corbet. "Large Receive Offload". Accessed: 2022-10-09. (2007), [Online]. Available: https://lwn.net/Articles/243949/.

[28] S. Godard, *Pidstat(1) linux user's manual*, Jul. 2020.

[29] T. L. D. Project. "/proc". Accessed: 2022-09-06. (), [Online]. Available: https://tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html.

[30] *Quiche*, https://github.com/cloudflare/quiche.

[31] *LiteSpeed QUIC*, https://github.com/litespeedtech/lsquic.

[32] *mvfst*, https://github.com/facebookincubator/mvfst.

[33] *MsQuic*, https://github.com/microsoft/msquic.

[34] *QUIC Interop Runner*, https://github.com/marten-seemann/quic-interop-runner.

[35] "Docker". Accessed: 2022-10-14. (), [Online]. Available: https://www.docker.com/.

[36] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using Docker in high performance computing applications", in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016, pp. 52–57. DOI: 10.1109/CCE.2016.7562612.

[37] 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.

[38] B. Huber, *Tc(8) linux user's manual*, Dec. 2001.

[39] "netmap". Accessed: 2022-10-14. (), [Online]. Available: https://github.com/luigirizzo/netmap.

[40] Google. "QUIC, a multiplexed transport over UDP". Accessed: 2022-10-14. (), [Online]. Available: https://www.chromium.org/quic/.

[41] K. Nepomuceno, I. N. d. Oliveira, R. R. Aschoff, *et al.*, "QUIC and TCP: A Performance Evaluation", in *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, pp. 00 045–00 051. DOI: 10.1109/ISCC.2018.8538687.

[42] A. Langley, A. Riddoch, A. Wilk, *et al.*, "The QUIC Transport Protocol: Design and Internet-Scale Deployment", in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, 183–196, ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. [Online]. Available: https://doi-org.eaccess.ub.tum.de/10.1145/3098822.3098842.

[43] A. S. Tanenbaum, *Modern Operating Systems*, Third. Pearson Education Inc., 2009.

[44] IBM. "Know your TCP system call sequences". Accessed: 2022-10-14. (2017), [Online]. Available: https://developer.ibm.com/articles/au-tcpsystemcalls/.

[45] T. Shreedhar, R. Panda, S. Podanev, and V. Bajpai, "Evaluating QUIC Performance over Web, Cloud Storage and Video Workloads", *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021. DOI: 10.1109/TNSM.2021.3134562. [Online]. Available: https://doi.org/10.1109/TNSM.2021.3134562.

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

[47] E. Volodina and E. P. Rathgeb, “Flow control in the context of the multiplexed transport protocol quic”, in *2020 IEEE 45th Conference on Local Computer Networks (LCN)*, 2020, pp. 473–478. DOI: 10.1109/LCN48667.2020.9314796.

[48] 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.