Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Evaluation of Scalability and Limitations of HTTP/3

Michael Kutter

Technical University of Munich

Department of Informatics

Bachelor's Thesis in Informatics

Evaluation of Scalability and Limitations of HTTP/3

Evaluation der Skalierbarkeit und der Grenzen von HTTP/3

Author:	Michael Kutter
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

In June 2022, the new version of the Hypertext Transfer Protocol has been finalized as HTTP/3, which uses QUIC as a transport layer protocol. Previous works have already showed that the network performance of HTTP/3 is similar compared to HTTP/2, except for small file sizes, where the transmission time of HTTP/3 was better. It was also shown that HTTP/3 has a worse computational efficiency than HTTP/2. However, none on the previous works have analyzed high load scenarios and its effects of HTTP/3. Therefore, we create a measurement setup in order to evaluate performance differences between HTTP/2 and HTTP/3 under high load. We especially focus on metrics like response time, number of requests, CPU utilization and memory consumption to describe the server performance. Additionally, we create different measurement scenarios, where we can focus on different parameters like number of concurrent connections and file size. We show that HTTP/3 has always around 4.77 times higher memory consumption than HTTP/2. For a small file size, we show that HTTP/3 is able to always handle more connections per second than HTTP/2, which is due to the improved handshake of HTTP/3. However, for bigger files, we show that HTTP/2 performs better under high load, due to the better computational efficiency.

Contents

List of Figures

Chapter 1

Introduction

The new major version of the Hypertext Transfer Protocol (HTTP) has been finalized in June 2022 as HTTP/3 [2]. Even though it is a relatively new protocol, it already takes up to 30% of all HTTP traffic, as seen in Figure 1.1. Compared to previous versions, which are using TCP as a transport layer protocol, HTTP/3 is built on top of QUIC [3]. The main advantages of HTTP/3 comes from enabling the features provided by QUIC.

The QUIC protocol specifications were standardized in May 2021 after nearly five years of development. It uses UDP on the transport layer, in order to keep support for middleboxes. The goal of this protocol is to improve HTTPS performance and to achieve high security. Therefore, QUIC exchanges cryptographic information during the connection establishment. This reduces the number of needed packets before transferring encrypted data between endpoints. It also saves previous session keys, which are used when reconnecting to a server. This allows to immediately transfer data during the handshake.

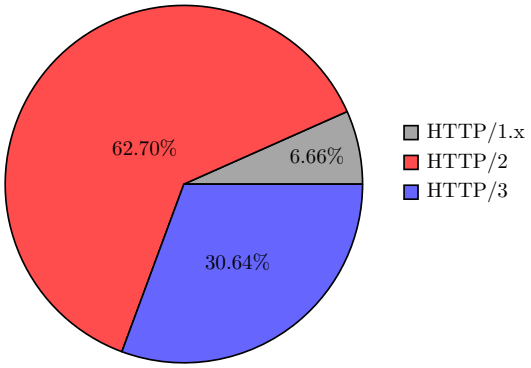


Figure 1.1: HTTP traffic distribution [1]

QUIC also introduces stream multiplexing, in order to solve the head-of-line blocking problem of TCP. Another benefit of QUIC is that it allows to change the IP address mid-connection. This is achieved by using connection IDs to identify the connection. Compared to TCP which is implemented in the Linux kernel, QUIC is implemented in the userspace. This allows for faster deployment cycles, as this can be achieved by a simple software updated. However, it reduces the performance due to the restricted memory and hardware access [4].

Previous studies have already analyzed the network performance of HTTP/3 and compared them to the currently most used version HTTP/2 [5]. It was shown that the performance between HTTP/2 and HTTP/3 was similar. However, HTTP/3 exceeds for smaller files, due to the improved handshake of QUIC. In this thesis, we want to focus on the server side and analyze how HTTP/3 behaves under high load scenarios compared to HTTP/2. We also want to verify if the increased complexity of the QUIC protocol has any impact on the performance.

We will start this thesis by giving some background about HTTP/2 and HTTP/3 in Chapter 2. Here, we give details about how those protocols work in general and try to outline the differences between them. In Chapter 3, we present related work and outline differences to our paper. Afterwards, in Chapter 4, we introduce our measurement setup. Here, we list details about our used hardware, software and tools in order to measure the performance. Also, we try to outline how we measure and calculate our used metrics. In Chapter 5, we then evaluate our measurements and compare the differences or similarities between HTTP/2 and HTTP/3 performance. The last Chapter 6 then concludes the thesis.

Chapter 2

Background: HTTP

The Hypertext Transfer Protocol (HTTP) builds the foundation for accessing websites on the Internet. It is an application layer protocol to transfer data between a client and a server. The client, typically a web browser, establishes a connection to the webserver. Afterwards, it sends an HTTP requests containing a path to the requested resource. The server then answers with a response containing the requested data. This typically happens multiple times, as a website normally consists of multiple resources. HTTP itself does not track lost packets. Therefore, it is dependent on a reliable transport layer protocol, like the Transmission Control Protocol (TCP) or QUIC.

The Internet Engineering Task Force (IETF) is mainly responsible for coordination of the development of HTTP through Requests for Comments (RFCs), with the latest version being HTTP/3 (RFC9114 [2]).

HTTP/1.x: The first major HTTP version HTTP/1.0 was published by the IETF in May 1996 as RFC1945 [6]. It uses TCP on the transport layer to reliable transfer data. This version suffered from poor network performance, as it needs to establish a TCP connection for each request. This was solved by HTTP/1.1, which was published in June 1999 as RFC2616 [7]. Here, multiple requests can be transferred over a single TCP connection. Nowadays however, these versions do not play an important part on the HTTP traffic as seen in Figure 1.1. This is due to newer HTTP versions offering better performance.

HTTP/2: HTTP/2 is the second major version of the HTTP network protocol. Currently, it is by far the most used version, making up to 60% of the HTTP traffic, as seen in Figure 1.1. Google originally developed an experimental successor for HTTP/1.1 called SPDY, which was later used as the basis for HTTP/2. The IETF then finalized

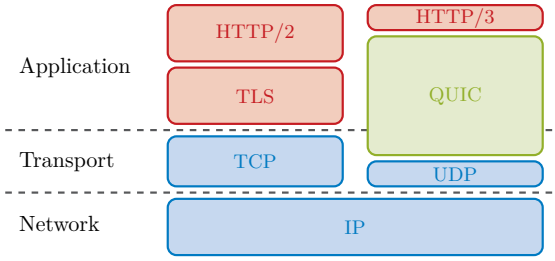
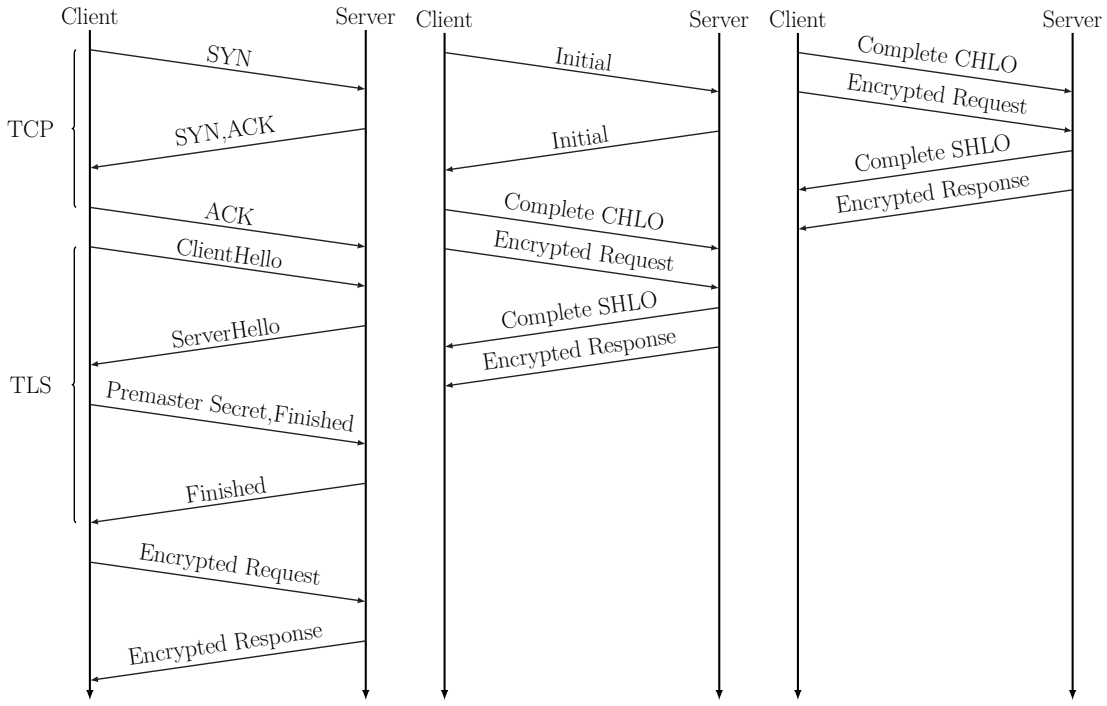


Figure 2.1: HTTP/2 and HTTP/3 network stacks

it in May 2015 as RFC7540 [8]. The main goal of this version was to reduce the web page load latency. This was achieved by introducing features such as stream multiplexing (Section 2.3) and header compression (Section 2.2). Similar to the previous version HTTP/1.1, HTTP/2 also utilizes TCP as a reliable transport layer protocol. One issue of HTTP is that all the data is being transferred in plain text. This means that on every intermediate node of the connection (e.g. router), the data could be read out easily. This can be prevented by encrypting data with e.g. Transport Layer Security (TLS). Transmitting encrypted HTTP data is called Hypertext Transfer Protocol Secure (HTTPS). The HTTP/2 standard itself can operate with and without encryption. Nowadays however, most browsers and webservers do not even support HTTP/2 without TLS. Therefore, when we talk about HTTP/2 in this thesis, we will only focus on HTTP/2 with TLS. Figure 2.1 shows the full HTTP/2 network stack with TLS. In June 2022, HTTP/2 received with RFC9113 its latest update [9].

HTTP/3: HTTP/3 is the latest major version of the HTTP network protocol. It was finalized by the IETF in June 2022 as RFC9114 [2]. Compared to previous HTTP versions, HTTP/3 uses QUIC as a transport layer protocol. Therefore, its main features and differences comes from utilizing QUIC, which was standardized in May 2021 as RFC9000 [3]. QUIC was originally developed by Google as an alternative for the TCP/TLS stack. It is built on top of User Datagram Protocol (UDP), in order to keep support for all middleboxes. Figure 2.1 shows the full HTTP/3 stack. The goal of this new protocol was to further improve performance (e.g. reducing page load time) of HTTPS connections, while also achieving high security. This was realized by improving the connection establishment (Section 2.1), stream multiplexing (Section 2.3) and by introducing connection migration (Section 2.5). Addionally, QUIC always encrypts traffic using TLS 1.3.



(a) HTTP/2 (b) HTTP/3 1-RTT (c) HTTP/3 0-RTT

Figure 2.2: HTTP/2 and HTTP/3 handshakes

2.1 Handshake

While HTTP/2 is built on top of TCP/TLS, HTTP/3 uses QUIC. Therefore, the connection establishment process differs between both versions.

HTTP/2 starts this process by performing the typical 3-way handshake of TCP [10]. Here, the client sends a SYN flag to the server, indicating that he wants to start a connection. The server then answers with a SYN+ACK flag to indicate that he accepts this connection. In order to finalize this connection establishment, the client then respond to the server with an ACK flag. Now the client and the server have successfully established the TCP connection. However, the connection is not encrypted yet. Therefore, the client performs a TLS handshake. This starts by the client sending a `ClientHello` message to the server. It typically contains supported TLS and cipher suites versions as well as the `Client Random`, which is a random sequence of bytes. The server then answers the client with a `ServerHello`. This message contains the server certificate, the used cipher suite and the `Server Random`. The client then authenticates the received certificates and answers with a premaster secret, which is also a random sequence of bytes encrypted with the public key. This key can be obtained with the certificate of

the server. Both client and server then generate the session key by using the previously shared `Client Random`, `Server Random` and premaster secret. The session keys should be the same for client and server. This is verified by exchanging a `Finished` message, which is encrypted with the created session keys. The total process takes 3 round-trip times (RTTs) before the encrypted request can be send. Figure 2.2a illustrates the complete process. With TLS 1.3 the process can be reduced to 2 RTTs. Here, during the exchange of the `Finished` message, encrypted data can already be exchanged.

HTTP/3 reduces the delay of the handshake by utilizing the 1-RTT handshake of QUIC. Here, the connection establishment and the handshake for encryption is combined. This means that during the connection establishment, QUIC also exchanges cryptographic information. It starts with the client sending an `Initial` packet to the server. This packet already contains the cryptographic information of the `ClientHello` message. The server then also answers with an `Initial` packet, which acknowledges the packet from the client. It also contains cryptographic information of the server. Afterwards, the client then responds by completing the `ClientHello` message. With this response, the client then can already transmit the encrypted request to the server. The server answers with finalizing the cryptographic exchange and already transmitting the response to the requested data. Figure 2.2b shows an overview of this process.

QUIC can further reduce the handshake delay when reconnecting to a server. Here, it can directly send the encrypted request to the server before receiving any response from the server. This handshake is called 0-RTT handshake. This is achieved by reusing the preshared encryption keys of the session ticket, which were negotiated in the first connection. Figure 2.2c demonstrates this in detail.

2.2 Header Compression

In order to further improve network performance, HTTP/2 and HTTP/3 are using header compression, which results in a smaller packet size and therefore in faster transmission times. The algorithm is called HPACK for HTTP/2 and QPACK for HTTP/3 [11][12]. They both work in a similar way.

The compression algorithm consists of a static and dynamic dictionary. The static dictionary is a look up table for the most commonly used header fields, while the dynamic dictionary only contains headers which were encountered during the connection. The use of these dictionaries can reduce the header size drastically, because if an entry already exists in the dictionary, it then only needs to reference this entry and not send the full header again. This saves data especially for repetitive header fields. In case if

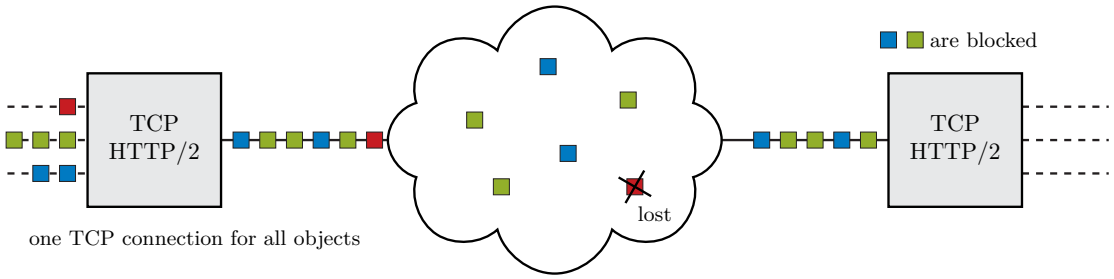


Figure 2.3: TCP head-of-line blocking

the entry was not found, the header is compressed by using Huffman encoding, which further reduces the size of the header compared to no compression.

The main difference between HPACK and QPACK is that QUIC does not ensure in-order delivery compared to TCP. Therefore, QPACK tracks states of entries in the dictionary and synchronizes them between the encoder and decoder. This ensures that the compression works even if packets are not send in order.

2.3 Stream Multiplexing

Early HTTP versions were loading resources in succession. However, this approach has a major flaw. If a resource cannot be loaded, then all following resources are blocked. This is called head-of-line blocking problem. In order to avoid this issue, HTTP/2 and HTTP/3 are introducing stream multiplexing. This ensures that multiple independent data streams can be loaded at the same time over a single connection. However, stream multiplexing works differently between HTTP/2 and HTTP/3.

HTTP/2 creates an independent data stream for each resource. All streams are then multiplexed and sent through a single TCP connection. This means that requests and responses can be sent in parallel. Now when a single resource can not be loaded, it does not block the other streams as they are sent in parallel and thus solving the head-of-line blocking problem. However, HTTP/2 still suffers from the head-of-line blocking of TCP. This occurs when a packet is lost during transmission. Then all other streams need to wait until the successful retransmission of the lost packet. Figure 2.3 illustrates the head-of-line blocking of TCP.

In contrast, HTTP/3 utilizes the stream multiplexing feature of QUIC. Here, QUIC creates independent data streams on the transport layer. This ensures that lost packets only block the streams, which were sent within this packet until retransmission. A typical strategy is to only send data of a single stream inside one QUIC packet. Having multiple independent data streams on the transport layer solves the head-of-line blocking

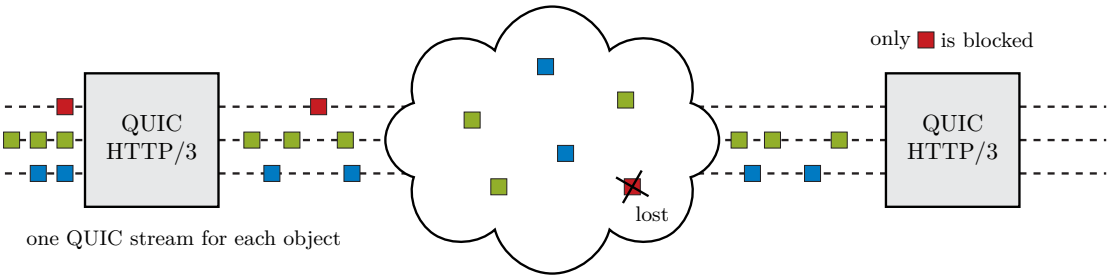


Figure 2.4: QUIC stream multiplexing

of TCP which is shown in Figure 2.4. This especially improves performance in networks with a high loss rate.

2.4 Prioritization

Due to the asynchronous behavior of stream multiplexing, important data may be received last. This can cause more loading time, as the client may not be able to start processing the data because it is still waiting for the important data to be received. E.g., the web browser cannot start rendering the page until the web page layout was received. Therefore, HTTP/2 and HTTP/3 are introducing prioritization. This allows the client to indicate which resources the server should send first. With this, it is possible to reduce loading times, as the client can start processing data earlier. It especially improves performance, when resources are dependent on each other. However, there is one major difference between HTTP/2 and HTTP/3. While HTTP/2 uses its own priority frames to indicate prioritization, HTTP/3 uses the prioritization feature of the QUIC protocol by indicating the relative priority of QUIC streams [3, Section 2.3].

2.5 Connection Migration

A new feature introduced in the QUIC protocol is connection migration [3]. This allows endpoints to change the IP address during data exchange. QUIC realizes this by using connection IDs to identify the connection after the IP address changed. These connection IDs are negotiated during the QUIC handshake. For the connection migration we need to differentiate between server and client. While the client can migrate mid-connection, the server is only allowed to migrate it after the handshake. Here, the server can transfer the connection to a preferred address [3, Section 9.6]. This is especially useful, when an IP address is shared by multiple servers. After an endpoint migrated the connection, the other endpoint initiates path validation, in order to verify and authenticate the new address.

The main advantage of connection migration is that the connection does not have to be fully reinitiated, as the state before the migration occurred can be reused.

Chapter 3

Related Work

The network performance of HTTP/2 and HTTP/3 is similar. This was observed by A. Yu and T. A. Benson [5]. The authors compared different HTTP/2 and HTTP/3 implementations on publicly available endpoints from Google, Facebook and Cloudflare. For small files the authors could observe that the HTTP/3 implementations transferred the files faster compared to HTTP/2. This was a result of the improved handshake of QUIC compared to the TCP/TLS handshake. For larger files the performance was similar, where the effect of the improved handshake of QUIC reduces. The authors also analyzed the effect of different network conditions like packet loss or network delay. The results were similar to optimal conditions, except for a few implementations. But this could be traced back to different congestion control algorithms or different server configurations and not to the protocol specifications itself. However, this paper only focuses on network performance and does not cover any server limitations or effects of high load scenarios.

The computational efficiency was analyzed by the blog post by K. Oku and J. Iyengar [13]. Here, the authors compared their QUIC implementation *quicly* with TCP and with TCP + TLS 1.3. Their goal was to identify computational differences between those implementations. Therefore, they reduced the CPU frequency, in order to bottleneck the systems processing power. They then measured the throughput achieved by the implementations at 100% CPU utilization. Plain TCP achieved close to double the throughput compared to TCP + TLS 1.3, which was twice as fast as QUIC. However, through some kernel and packet size optimization, QUIC was able to achieve similar throughput compared to TCP + TLS 1.3. These measurements however do not cover multiple client requests or multiple client connections.

The paper by K. Jacksi et al. analyzed the impact of different Distributed-Denial-of-Service (DDoS) attacks [14]. The authors created a test setup with multiple hosts in order to attack the server. They analyzed 3 scenarios, one scenario without any attacks one with SYN attacks and one with HTTP attacks. In order to measure the impact of these DDoS attacks, they used CPU utilization and response time as performance indicators. However, this paper did not cover the new HTTP/3 protocol, but it provides a good baseline for our measurement setup in order to generate high load.

In order to create high load scenarios, we need to consider bandwidth utilization, which was analyzed by G. Lui et al. [15]. Here, the authors created a measurement setup consisting of multiple client hosts connected to a single webserver. They analyzed the bandwidth of the webserver for different file sizes and different number of concurrent requests. The result was that for small files the bandwidth utilization was very low, regardless of the number of concurrent connections.

The impact of hardware scalability was analyzed by A.-P. Barzu et al. [16]. The authors evaluated the effects of number of CPU cores and amount of RAM on a web server. They observed the following behavior. The increased number of CPU cores improved the response and processing time, while the increased amount of memory reduced the number of failed requests. As our measurements are also heavily dependent on the hardware, this paper provides a baseline on what needs to be considered when creating our measurement setup.

Chapter 4

Measurement Setup

To run our experiments we created the measurement setup shown in Figure 4.1. The setup consists of two hosts, where one host acts as the webserver and the other host acts as the clients. The webserver is responsible for handling incoming client requests and monitor metrics like CPU and memory utilization. The client generates multiple HTTP requests to keep the server on high load. Both hosts are directly connected to each other without any intermediate nodes. With this we want to ensure optimal network conditions, as we only want to evaluate server performance and not network performance.

However, this setup can have one major bottleneck. We are limited by a single host on the client side. This means that this host may not have enough processing power in order to keep the server at a high load. We can compensate this issue by choosing appropriate hardware with enough processing power.

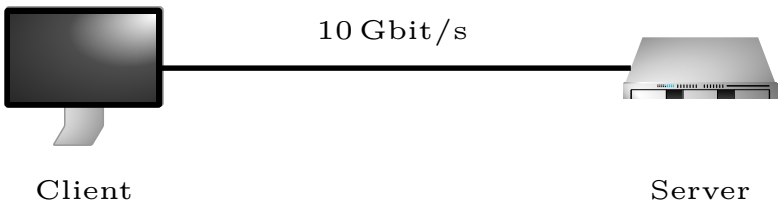


Figure 4.1: Measurement setup

CPU	Intel Xeon E5-2640 v2 @2.00 GHz
Cores	8 cores (16 threads)
RAM	32 GB DDR3 @1333 MHz
NIC	10 Gbit/s Intel X540-AT2
OS	Debian 11.0 (Bullseye)
Kernel	Linux Kernel 5.10

Table 4.1: Hardware

4.1 Hardware

We use the same hardware for the client and server. A detailed list is shown in Table 4.1. For the CPU we use an 8 core (16 threads) Intel Xeon CPU. This enables us to evaluate how the server distributes load on multiple threads. However, to not run into the previously mentioned client bottleneck, we limited the webserver application to only use a maximum of 8 threads. This also allows us to run the monitoring tasks on the 8 remaining threads. Therefore, the measurement tools do not impact the webserver performance. However, on the client side, we fully utilize all 16 threads to generate client requests. To not run into memory issues, we use 32 GB of RAM. We also use a 10 Gbit/s ethernet connection. This allows us to keep the server under high load and not run into the bandwidth limit throughout our tests. On the operating system (OS) side, we use Debian Bullseye.

4.2 Server

The server is responsible for handling all the incoming HTTP requests and monitoring the CPU and memory utilization. Therefore, we use following tools for these tasks.

4.2.1 Proxygen

For our webserver implementation, we use Proxygen [17]. Proxygen is a high-performance HTTP library written in C++. It was originally developed by Meta as a software library for proxies [18]. Nowadays however, it evolved to a framework for building client, server and proxies for all HTTP versions, beginning from HTTP/1.1 up to the new version HTTP/3. For HTTP/3, they use their own inhouse QUIC implementation called mvfst [19]. One of the reasons why we choose Proxygen was that it is already in commercial use by Meta [20]. This means that it is not just some concept implementation for the new HTTP/3 version with no optimizations. Another reason was that it also provides sample implementations for webservers for HTTP/2 and HTTP/3, which we are using for our measurements. We set up the server in a way that it only allows file

transfer, as we do not want to have any side effects of a full webservice on the CPU and memory utilization.

4.2.2 perf

To measure CPU usage of the webserver, we use the tool perf. Perf is a powerful performance analyzer with various possibilities directly implemented in the Linux kernel [21]. It allows for profiling general performance information like the CPU time of a process up to detailed hardware events such as cache-misses. However, perf can cause high CPU loads itself when recording events in detail. For our setup, this is not an issue, as we only use 8 threads for our webserver, which means that we have 8 unused threads for tools like perf, and we only count the CPU time of the webserver process.

4.2.3 pmap

For analyzing the memory usage of the webserver, we use the tool pmap. Pmap does not only analyze the memory utilization of the given process, it also analyzes the memory consumption of each individually used component of this process [22]. We use the resident set size (RSS) field as our memory utilization value.

4.3 Client

The client is responsible for keeping the webserver under constant load by establishing connections and sending HTTP requests to the webserver. Additionally, we also use the client for analyzing the server performance, by monitoring metrics like response times, number of requests, etc. Therefore, we created our own client implementation by using the lsquic and nghttp2 libraries.

4.3.1 lsquic

For our HTTP/3 client, we use the lsquic library. Lsquic was developed by LiteSpeed Technologies and provides a feature rich HTTP/3 and QUIC library written in C [23]. During the draft phase of the QUIC protocol by the IETF, they were one of the first implementation to update their code once a new draft was released. Similar to proxygen, lsquic is also already in commercial use. Our code is based on the tutorial implementation by Dmitri Tikhonov[1]. However, we have modified it in order to support multiple parallel connections and to keep the server under constant load.

[1] `https://github.com/dtikhonov/lsquic-tutorial`

4.3.2 nghttp2

For our HTTP/2 client, we use the nghttp2 library. Nghttp2 was mainly developed by Tatsuhiro Tsujikawa and provides a HTTP/2 library written in C [24]. E.g. the curl client is based on this library for their HTTP/2 connections [25]. Our code is based on the sample implementation provided by the library, which we have modified in order to support multiple parallel connections and to keep the server under constant load.

4.4 Key Performance Indicators

A single measurement runs for 60 s with fixed parameters, which are dependent on the test scenario. The 60 s starts after we initialized our client. Then, we create a fixed amount of concurrent connections, which were previously specified by the test scenario. These connections then send an HTTP request to the webserver. Once the requested data was transmitted, we then either send a new HTTP request or we close the connection and create a new one, which then sends the HTTP request again, depending on the test scenario. With this we can ensure that always the same number of concurrent connections are active. Once the 60 s have passed, we stop our measurement. Afterwards, we start a cooldown phase, where we wait until all the active connections were successfully finished. However, we do not include these connections in our measurement. During the measurement, we monitor or calculate different metrics, which then indicates performance of the server. We use following parameter as your key performance indicators.

Number of Requests: To analyze the capacity of the webserver, we count the number of successful requests. A request is considered successful once all of the requested data is transmitted. Based on this, we can estimate how many requests per second the server can handle, by dividing the number of requests through the measurement time. Here, the measurement time is 60 s. In order to ensure that a timeout does not block a connection or interfer with our measurements, we set the timeouts to 2 s.

Response Time: The response time is calculated for each connection. Here, it is considered as the difference from the start of the connection until the end. The start of the connection begins with sending the first packet of the handshake. The connection ends when sending the connection closed packet. We add up all the calculated response times for each successful connection and divide it by the number of requests in order to calculate an average response time for all connections.

Goodput: We calculate the goodput of the webserver by counting all the successful transferred bytes of the requested file. Afterwards, we divide this number by the measurement time in order to get the effective goodput per second.

CPU Utilization: As described in Section 4.2.2, we use perf to measure CPU usage. The measurement starts with a 20 s delay after the client started sending the requests. It then records the CPU time of the proxygen process for 20 s. This ensures that the CPU time is tracked when the server is under constant load.

Memory Utilization: To monitor memory consumption of the webserver, we use the tool pmap as described in Section 4.2.3. We start measuring once before we start sending client requests to the server, in order to receive a baseline memory consumption. Afterwards, we measure the used memory each second. With this we can monitor how the memory utilization changes during run time. After the client finished sending requests, we once again measure the memory.

Chapter 5

Evaluation

In order to analyze the server performance of HTTP/2 and HTTP/3, we created five different scenarios. Each scenario consists of multiple measurements with changing parameters (e.g., number of concurrent connections, file size) depending on the scenario.

5.1 Scenario: File Descriptor

With this scenario, we evaluate the number of open file descriptors the server process opens during run-time. File descriptors are used in order to uniquely identify an open file [26]. A process needs to open a file when it wants to handle input/output resources, such as kernel handles (e.g. network sockets) or regular files. When a process starts, the first three file descriptors, which are opened, are stdin, stdout and stderr. They are mainly used for input/output of a terminal.

We made multiple different measurements to receive the number of open file descriptors. With each measurement we increase the number of concurrent connections. In order to ensure that the connections were active and processed in parallel, we constantly requested the same file before closing the connection. In detail, we established a fixed number of connections. Then each connection send an HTTP request to the server. Once this request was successfully handled by the server, we immediately re-requested the same file. After the number of open file descriptors where measured, we then closed the connections again. The size of the requested file was 5 kB.

Figure 5.1 shows the number of open files descriptors of the proxygen server for HTTP/2 and HTTP/3. We can see that without any active connections the proxygen process has already 127 open files by default. Most of these open files come from accessing

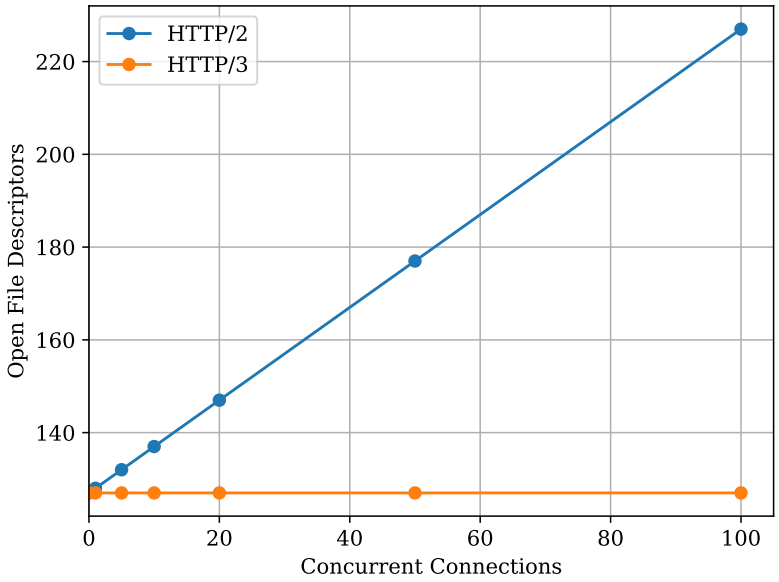


Figure 5.1: Open File Descriptors

shared libraries, events, and pipes. More interesting are the file descriptors which are used for the HTTP connections. Here, the server uses one file descriptor in order to listen for incoming TCP connections for our HTTP/2 requests. For our HTTP/3 requests the server uses eight UDP file descriptors (one file descriptor for each thread). When we increased the number of concurrent connections, we could observe different behaviors for HTTP/2 and HTTP/3. For HTTP/3, the number of file descriptor were always constant, regardless of the number of concurrent connections. For HTTP/2, the number of file descriptor increased by one for each HTTP/2 connection. The difference between HTTP/2 and HTTP/3 come from the different transport layer protocols. While HTTP/3 uses QUIC, which is based on UDP, HTTP/2 uses TCP. The UDP sockets were created at startup of the proxygen server, which handles all the QUIC traffic. TCP on the other hand creates a new socket for each accepted connection in order to handle the traffic.

Another issue which should be kept in mind when using a webserver is that Linux limits the number of file descriptors to 1024 for each process [27]. Therefore, for a high number of concurrent connections, this limit should be increased, especially when using HTTP/2. Otherwise, the server program will crash. Therefore, for the following scenarios we increased this limit to 20 000

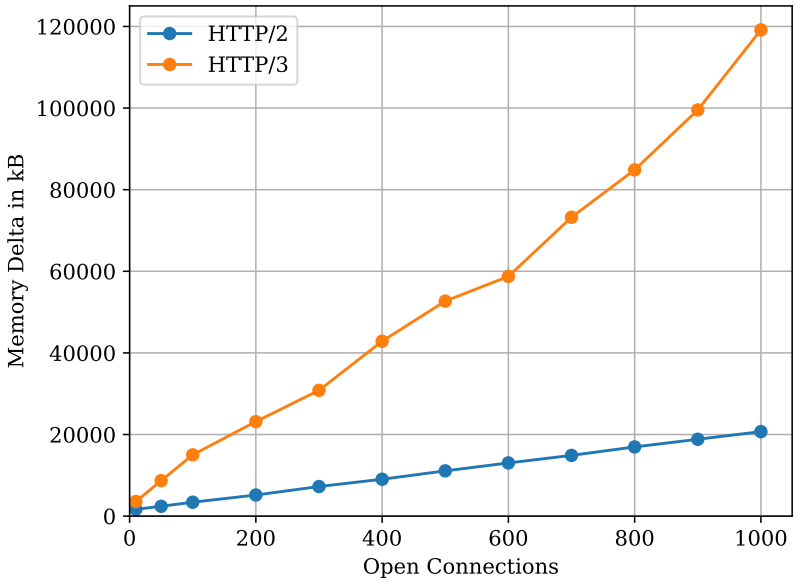


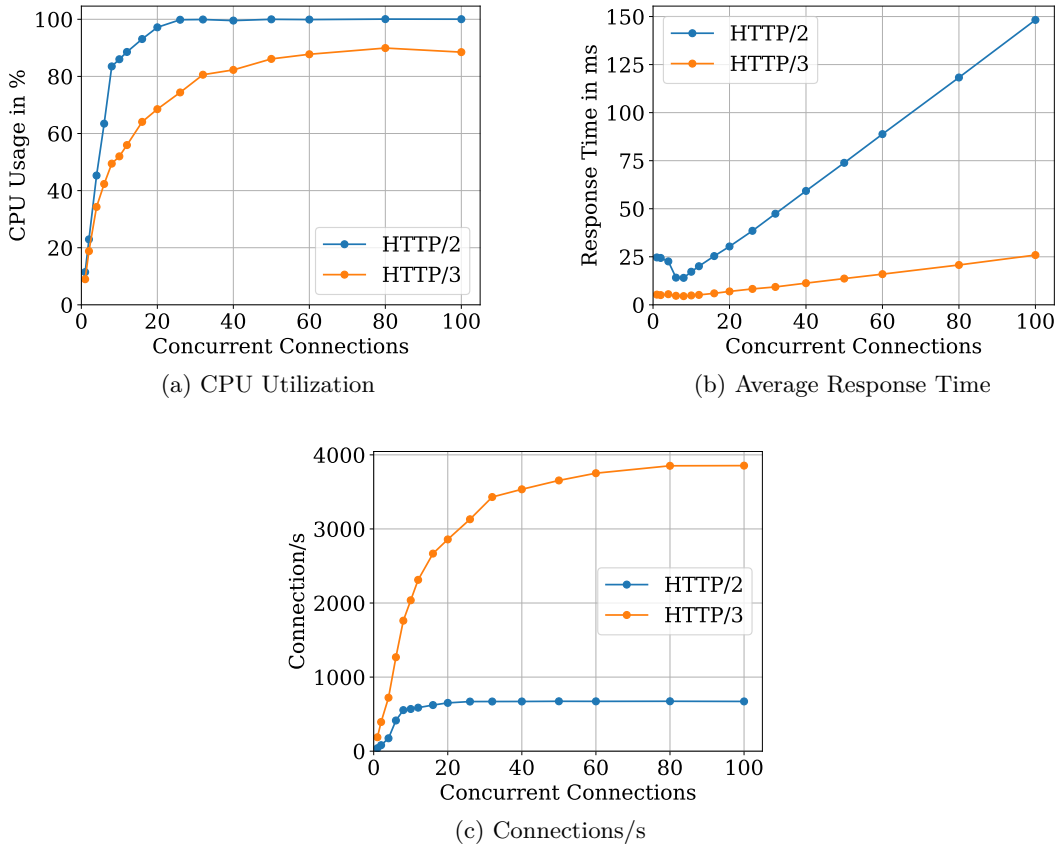
Figure 5.2: Keep Alive Memory Consumption

5.2 Scenario: Keep Alive

To improve loading times, it is faster to keep a connection open. With this, we can save the connection establishment process and directly request data. However, this causes additional memory consumption for the server, as the state of the connection needs to be saved. With this scenario, we want to analyze memory consumption for a different number of open connections between HTTP/2 and HTTP/3. Therefore, we set up our client to only establish a specified number of connections and keeping it alive for 10 s. Then, we measure the memory consumption of the proxygen process every second. The measurement is started before we establish the connections and is stopped after all connections are closed. The proxygen process was restarted between measurements in order to ensure that previous measurements do not have any effects on the current measurements.

5.2.1 Memory Consumption

The initial memory consumption before a measurement of the proxygen process always differs. Therefore, we calculate a memory delta for better comparison. The memory delta is the difference between the highest measured point and the initial memory consumption before the first connection establishment. The highest measured memory point was directly reached after all connections were established. Afterwards the mem-



(a) CPU Utilization

(b) Average Response Time

(c) Connections/s

Figure 5.3: Concurrent Connections

ory consumption stayed at this point for the duration of the measurement. Figure 5.2 shows this delta of memory for an increasing number of open connections. It shows that for both HTTP/2 and HTTP/3, the memory consumption increases nearly linear to the number of open connections. E.g., for 500 open connections, the memory delta is 11 088 kB for HTTP/2 and 52 688 kB for HTTP/3. Based on these numbers, a single connection needs about 22 kB for HTTP/2 and 105 kB for HTTP/3. This shows that HTTP/3 needs about 4.77 times more memory than HTTP/2.

5.3 Scenario: Concurrent Connections

A webserver needs to be able to handle different connections in parallel, as requests can be received at the same time from different clients. Therefore, we created a scenario, which analyzes the impact of concurrent connections on the webserver. Here, we create a fixed amount of concurrent connections where each of the connection sends an HTTP

request to the webserver in parallel. The requested file is 5 kB in size in order to ensure that the bandwidth does not bottleneck our measurement. Once a request was served by the server, the client closes the connection and directly creates a new connection which then send the HTTP request again. With this, we can ensure that always the same number of connections are active at the same time. Each measurement runs for 60 s. Additionally, the proxygen process is being restarted between each measurement to ensure that previous measurements do not have any effects on the current measurements.

5.3.1 UDP Buffer Size Issue

With our first measurements, the HTTP/3 server reported multiple Probe Timeouts (PTOs). These are used in order to recover from loss of tail packets or acknowledgments [28]. As the PTO does not directly indicate packet loss [28, Section 6.2], we first thought that this issue was linked to our client implemenation. However, with further analysis, we found out that the HTTP/3 server suffered from packet loss. This was caused by the UDP buffer, which could not handle all incoming data of the client due to the buffer size being too small. Therefore, we increase the buffer size from 208 KiB to 25 MiB. This ensures that the server has enough buffer for all incoming data. We apply this change of the receive buffer for all the following measurements and scenarios.

5.3.2 CPU Utilization

The CPU is one of the main factors, which can limit the server performance. A faster CPU or more cores can lead to faster response and processing times [16]. Figure 5.3a shows the CPU utilization for an increasing number of concurrent connections. Here, 100% means that all 8 cores of the proxygen process are being used. For a single active connection, both HTTP/2 and HTTP/3 only utilize around 11% of the available CPU resources. However, this increases for an increasing number of concurrent connections. For a high number of concurrent connections, it slowly reaches a limit. Here, HTTP/2 is able to utilize all 8 available threads, while HTTP/3 only achieves 90% of CPU load. A per thread analysis showed for HTTP/3, that one threads always run at 100% utilization, while for the other threads the utilization is constantly fluctuating. This suggests that this one thread is responsible for distributing new incoming connections to the other threads. However, we could not fully verify this behavior. But it shows the difficulty of managing the transport layer protocol (QUIC) in the userspace due to the restricted memory and hardware access [4]. For HTTP/2, the transport layer protocol (TCP) is directly implemented in the Linux kernel. Therefore, the kernel manages all incoming and outgoing packets thus resulting in a more efficient use of memory and hardware resources.

5.3.3 Response Time

Figure 5.3b shows the average response time of all sent requests. It shows that the average response time increases linearly in relation to the number of concurrent connections. The only exception is for a small number of concurrent connections (≤ 4), where there is a spike visible. Here, the average response time is up to 71% higher for HTTP/2 and 17% higher for HTTP/3 requests compared to the lowest average response time. This can be explained by the sleep/wakeup cycle of the active server threads. The server sends inactive threads to sleep in order to reduce the CPU load. However, when a request is received, the server needs to wakeup the inactive threads in order to process the request. For a higher number of concurrent connections, it is more likely that threads are already active as they are processing other connections. Therefore, the sleep/wakeup cycle takes less time. The linear behavior can also be explained by the CPU load. Here, the server is overloaded by the number of concurrent connections and cannot process all the active connections at the same time. Therefore, incoming data needs to wait until previous data was successfully processed by the webserver.

For the average response time, HTTP/3 outperforms HTTP/2. Here, the response time for HTTP/2 connections is 5.6 times higher than for HTTP/3. This difference comes from the improved connection establishment of HTTP/3 connections, where less RTTs are needed compared to HTTP/2.

5.3.4 Number of Connections

Another important aspect for the webserver performance is the number connections per second (Connections/s) it can handle. As each connection only sends one requests, we can use the number of requests as described in Section 4.4 for this evaluation. Figure 5.3c shows the number of connection per second for an increasing number of concurrent connections. For a single active connection, HTTP/2 and HTTP/3 only manage a low number of connections per second (40 Connections/s for HTTP/2 and 187 Connections/s for HTTP/3). However, the number increases rapidly, when increasing the number of concurrent connections. E.g., for 8 active concurrent connections, HTTP/2 handles 414 Connections/s and HTTP/3 handles 1268 Connections/s. Afterwards, the number of connections per second slowly reaches a limit, where an increasing number of concurrent connections does not lead to a higher number of handled connections per second. For our measurement setup this limit for was 673 Connections/s for HTTP/2 and 3854 Connections/s for HTTP/3. This behavior is directly linked to the CPU load of the webserver, where for a small number of concurrent connections the CPU load is low, while it reaches its maximum for a high number of concurrent connections, as described in Section 5.3.2.

HTTP/3 was able to handle more connections per second compared to HTTP/2. At the limit, HTTP/3 could handle 5.7 times more connections as HTTP/2. This performance was gained by the improved handshake of the HTTP/3 protocol compared to HTTP/2, where it needs less RTTs.

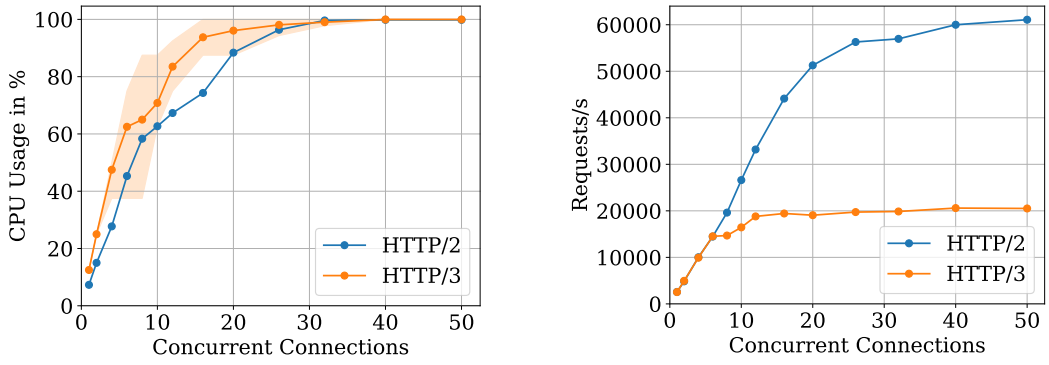
Based on the connection per second, we can also calculate the goodput for this scenario. As each connection only contains a single request for the same file, we can multiply the numbers of connections per second with the file size (5 kB). This means that for a single concurrent connection, the goodput is 0.2 MB/s for HTTP/2 and 0.94 MB/s for HTTP/3. At the limit, the goodput is 3.36 MB/s for HTTP/2 and 19.27 MB/s for HTTP/3. These values also match our measured goodput. Compared to the bandwidth limit of 10 Gbit/s, this scenario only uses a fraction of its capacities, which is due to the small requested file of 5 kB.

5.4 Scenario: HTTP Requests

Establishing a connection for each request to the same server increases the total transfer time. Therefore, it is better to only establish the connection once to the server and send all HTTP requests over this connection. Now the handshake only needs to be performed once for all requests to this server. With this scenario we want to evaluate the difference between HTTP/2 and HTTP/3 when we request multiple files over the same connection. Therefore, we establish a connection to the server and send a HTTP request for a 5 kB big file. Once the file was successfully transferred, we immediately re-requested the same file. This simulates requesting multiple files and keeps the server under constant load. We do this process for multiple connections in parallel. When 60 s have passed, we finish the currently active requests and afterwards stop our measurement. Additionally, we restarted the proxygen process in order to ensure that previous measurements do not have any effects on the current measurements.

5.4.1 CPU Utilization

Figure 5.4a shows the CPU utilization for an increasing number of concurrent connections. Here, 100% means that all 8 cores of the proxygen process are being used. For both HTTP/2 and HTTP/3, the CPU usage increases rapidly up to the limit of 100%. Interestingly, HTTP/3 uses 12.5% for a single active connection, while HTTP/2 only uses 7.31%. A per thread analysis showed, that proxygen only utilizes a single thread in order to handle this connection. However, HTTP/3 fully utilizes this thread (100%), while HTTP/2 only utilize 58%. This also shows the better computational efficiency of HTTP/2 compared to HTTP/3, as less resources are needed in order to handle the



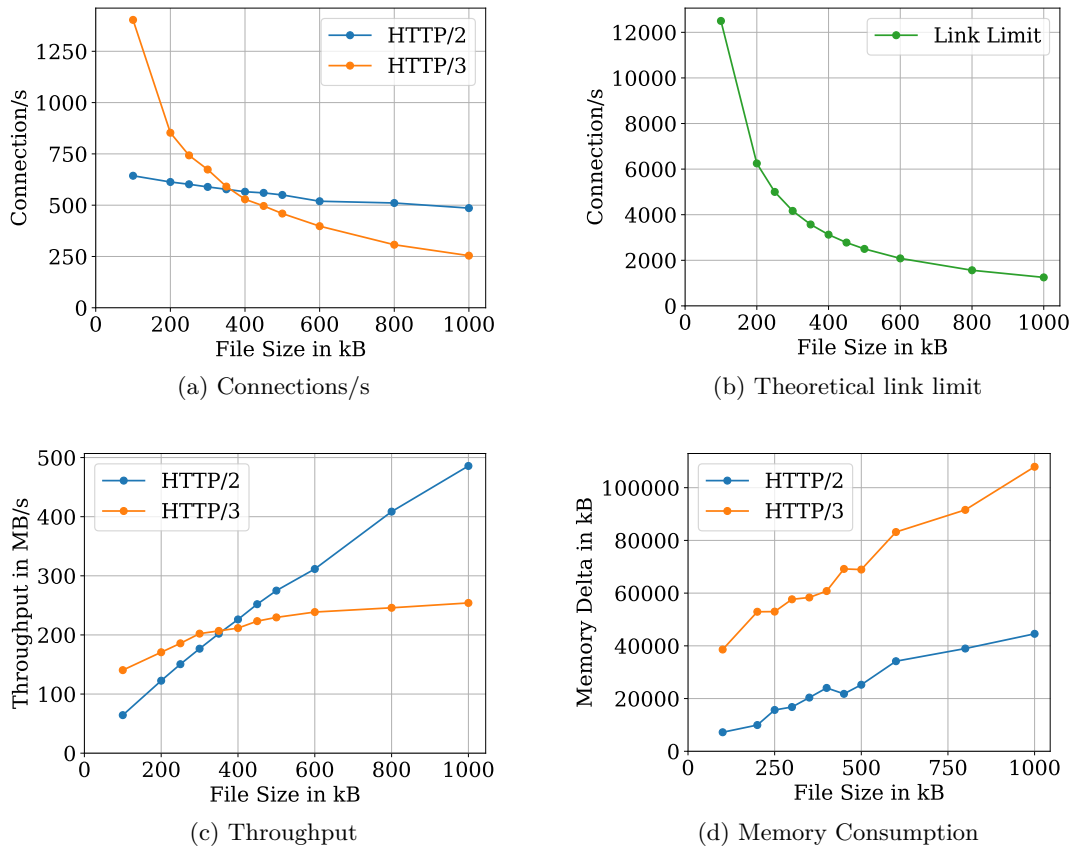
(a) CPU Utilization (b) Requests/s

Figure 5.4: HTTP Requests

connection [13]. Another effect we observed was that HTTP/3 could not always properly distribute the parallel connections over the available threads. We expected a linear increase of CPU utilization for up to 8 active parallel connections, as here each thread only needs to handle one connection, which is the exact behavior we could observe for HTTP/2. However, for HTTP/3, proxygen often mapped multiple connections to the same thread thus resulting in a lower CPU utilization and lower overall performance, as a thread is already fully occupied by a single connection. This behavior of HTTP/3 was very inconsistent for the same measurement. E.g., proxygen utilized a range of 3 to 7 threads in order to handle 8 concurrent connections. This is also shown in Figure 5.4a through the colored area. This effect became less relevant for a high number of concurrent connections (≥ 20). Here, all 8 available threads were always being used and the connections were better distributed between the threads. We argue that this effect is caused by the proxygen implementation.

5.4.2 Number of Requests

Figure 5.4b shows the number of requests per second (Requests/s) for an increasing number of concurrent connections. For a single connection, HTTP/2 achieves 2568 Requests/s and HTTP/3 achieves 2540 Requests/s. When comparing these values with the CPU utilization, as described in Section 5.4.1, it shows that HTTP/3 can keep up with HTTP/2 given enough CPU resources. However, while HTTP/3 used 100% of the associated thread, HTTP/2 only used 58%. This again shows the better computational efficiency of HTTP/2 [13]. The computational efficiency is also the reason for the better performance of HTTP/2 for a high number of concurrent connections. Here, HTTP/2 limits at 61 078 Requests/s, while HTTP/3 already limits at 20 594 Requests/s. Here, the number of requests per seconds are also fluctuating a lot



(a) Connections/s

(b) Theoretical link limit

(c) Throughput

(d) Memory Consumption

Figure 5.5: File Size

for HTTP/3 for concurrent connections ≤ 20, due to the inconsistent distribution of connections to threads, as describe in Section 5.4.1.

The goodput behaves similar to the number of requests, as each request is for the same file (5 kB). This means that HTTP/3 can keep up with HTTP/2 for a single active connection. Here, both HTTP/2 and HTTP/3 achieves a goodput of around 12.84 MB/s. For a high number of concurrent connections, where the better computational efficiency of HTTP/2 prevails, HTTP/3 limits at 103 MB/s and HTTP/2 at 305 MB/s. However, this shows, that in this scenario both HTTP/2 and HTTP/3 only use a fraction of the bandwidth limit of 10 Gbit/s

5.5 Scenario: File Size

With the previous scenarios, we observed that HTTP/3 performed better on a connection level (see Section 5.3), while HTTP/2 performed better on an HTTP level (see

Section 5.4). This suggests that HTTP/3 will perform better for small files, while HTTP/2 will perform better for big files. Therefore, with this scenario we want to observe the performance when increasing the requested file size. We start a measurement by establishing 32 parallel connections, which each connection sending an HTTP request to the webserver. Once the server successfully responded to the request, we close the connection and directly reestablish a new connection. With this we can ensure that always 32 connections are active in parallel, and the server is under constant load. When 60 s have passed, we finish the currently active requests and afterwards stop our measurement. Between each measurements, we restarted the proxygen process in order to ensure that previous measurements do not have any effects on the current measurements.

5.5.1 Number of Connections

Figure 5.5a shows the number of connections per second (Connections/s) for an increasing file size with 32 concurrent connections. For both HTTP/2 and HTTP/3, the number of connections per second drops for an increasing file size. However, the number drops faster for HTTP/3 than for HTTP/2. HTTP/3 drops from 1403 Connections/s for a 100 kB file to 254 Connections/s for a 1000 kB file, while HTTP/2 only drops from 643 Connections/s to 485 Connections/s. This is because the improved connection establishment of HTTP/3 becomes less relevant, while the better computational efficiency of HTTP/2 becomes more significant [13]. This also means that HTTP/2 performs better for large files, while HTTP/3 performs better for small files. Therefore, we measured this crossover point at around a file size of 350 kB for our measurement setup.

Figure 5.5b shows the theoretical link limit for connections per second. Here, we divide the bandwidth limit by the file size. This provides us an theoretical upper limit of the capabilities the link could achieve. When we compare the numbers of connections per second achieved by HTTP/2 and HTTP/3 with this link limit, we can see that both HTTP/2 and HTTP/3 only utilize a fraction of the bandwidth limit. For a 100 kB, HTTP/3 utilizes 11% of the link, while HTTP/2 only utilizes 5%. However, for larger files, both HTTP/2 and HTTP/3 are able to utilize more of the link. Here, HTTP/2 achieves 39%, while HTTP/3 achieves 20%.

5.5.2 Goodput

Figure 5.5c shows the goodput for an increasing file size with 32 concurrent connections. For a 100 kB big file, HTTP/3 achieves 140 MB/s, while HTTP/2 only achieves 64 MB/s. For larger file, the goodput of both HTTP/2 and HTTP/3 increases. However, the goodput of HTTP/2 increases faster than for HTTP/3. Here, for a 1000 kB big file,

HTTP/3 achieves 254 MB/s and HTTP/2 achieves 485 MB/s. This is, as explained in Section 5.5.1, due to the improved handshake of HTTP/3 becoming less relevant for an increasing file size, while the better computational efficiency of HTTP/2 becomes more significant [13]. The crossover point for better HTTP/2 performance is again at around a 350 kB big file.

5.5.3 Memory Consumption

The memory consumption was measured every second, starting before the first connection establishment and finishing after all connections are closed. As the initial memory consumption differs between the measurements, we calculate a memory delta for better comparison. The memory delta is the difference between the highest measured point and the initial memory consumption before the first connection establishment. Figure 5.5d shows this memory delta for an increasing file size with 32 concurrent connections. Although the graph does not provide a consistent curve, it shows a clear trend that the server needs more memory for larger files for both HTTP/2 and HTTP/3. This is because a larger file needs to be loaded into memory for processing the request. It also shows that the memory consumption of HTTP/3 is always higher than for HTTP/2. Here, for a 100 kB big file, HTTP/3 had a memory delta of 38 604 kB, while HTTP/2 only had a delta of 7216 kB and for a 1000 kB big file, HTTP/3 had a memory delta of 107 932 kB, while HTTP/2 only had a delta of 44 560 kB. The reason for this is that HTTP/3 connections generally use more memory compared to HTTP/2, as seen in Section 5.2.1.

Chapter 6

Conclusion

This thesis aimed to analyze the effects of high load on a webserver. Specifically, we evaluated the differences between HTTP/2 and the newest version HTTP/3. Therefore, we created a measurement setup consisting of two hosts, where one host acted as the server and the other host acted as the client. On the server side we used proxygen for both HTTP/2 and HTTP/3. On the client side we used lsquic for HTTP/3 and nghttp2 for HTTP/2. The server was set up for simple file transfers.

For our evaluation we focused on different metrics to describe server performance. Here, we used metrics like response times, number of served requests, CPU utilization and memory consumption. Additionally, we created five different scenarios, where we focused on different parameters like, number of concurrent connections and file size. These scenarios showed that HTTP/3 has always around 4.77 times higher memory consumption than HTTP/2. We also showed that HTTP/3 always handled more connections per second than HTTP/2 for a small file size. We concluded that this is due to the improved handshake of HTTP/3. However, for bigger files, HTTP/2 performs better, which is a result of the better computational efficiency of HTTP/2.

We also identified an issue with the default UDP buffer size. Here, the receiving buffer of the webserver is too small in order to handle higher loads. Therefore, we recommend to always increase this buffer size.

We also showed that when we constantly re-requesting the same file without establishing a new connection for each request, HTTP/2 is able to achieve more requests per second than HTTP/3, which is also due to the better computational efficiency of HTTP/2. However, for a single active connection, HTTP/3 was able to match HTTP/2 by using

more CPU and memory resources. Therefore, we concluded that HTTP/3 is able to match HTTP/2 given enough CPU and memory resources.

In this thesis, we only used the proxygen implementation to cover HTTP/2 and HTTP/3 on the server side. However, it would be interesting to see how other HTTP/2 and HTTP/3 implementations handle these scenarios and evaluate possible performance differences between those implementations.

Chapter A

Appendix

A.1 Reproducibility

In this section, we want to show more details of our measurement scripts and explain how these results can be reproduced.

A.1.1 Server

For the webserver, we use the *hq* sample implementation from the proxygen library. This implementation provides both HTTP/2 and HTTP/3 within a single executable. We did not make and modifications to the code, as the implementation already provides a static file transfer, which we use for the measurements. We start the server with the following command:

```
./hq --mode=server --port=8080 --h2port=8080 --host=10.0.0.1
--static_root=/root/server --threads=8
```

For the CPU measurement, we use the tool perf. We use the *perf stat* command to track the CPU time of the *hq* process. To ensure that the server is under constant load, we delay this measurement by 20 s. Afterwards, we run the measurement for 20 s. The following script is used for this measurement:

```
sleep 20
perf stat -d -p ‘pidof hq‘ sleep 20
```

For the memory consumption measurement, we use the tool pmap. We track the memory consumption measurement each second for the duration of the measurement. We start this measurement before sending any HTTP requests to the server, to get an ini-

tial memory consumption of the *hq* process. We use following script for the memory consumption measurement:

```
for i in {0..60}
do
    pmap -x `pidof hq` | grep total
    sleep 1
done
pmap -x `pidof hq` | grep total
```

For the open file descriptor measurement, we use the *lsof* command, which provides a detailed list of all open files. We then count the number of lines to retrieve the number of open file descriptors. Therefore, we use following command:

```
lsof -p `pidof hq` | wc -l
```

The default limit for open file descriptors is 1024. However, some scenarios exceed this limit [27]. Therefore, we increase it to 20 000 We do this for both the server and the client. Here, we use following command:

```
ulimit -n 20000
```

The default UDP buffer size was too small for high load scenarios to handle all incoming data. This resulted in packet loss for HTTP/3. Therefore, we increased the UDP buffer to 25 MiB with following command:

```
sysctl -w net.core.rmem_max=26214400
sysctl -w net.core.rmem_default=26214400
```

A.1.2 Client

On the client side, we modified the lsquic tutorial implementation by Dmitri Tikhonov[1] for HTTP/3 and the libevent sample implementation from nghttp2 [24] for HTTP/2. We added support for multiple connections in parallel. This is achieved by creating a lsquic or nghttp2 instance for each connection. For better CPU usage, we added multithreading. Here, we equally distribute the number of connections to each thread for optimal performance. In order to prevent race conditions, we track connection states and statistics per thread and accumulate it at the end of our measurement. We utilize the callback functions from the lsquic or nghttp2 instance to create a new request after the previous request was successfully responded. Additionally, we create a command line argument parser to change the number of concurrent connections and the measurement duration. Following command is used for a measurement duration of

[1] https://github.com/dtikhonov/lsquic-tutorial

60 s with 32 concurrent connections. Here, we close the connection, once the request was successfully responded, and establish a new connection to ensure that always 32 connections are active in parallel:

```
./prog --conn 32 --duration 60
```

With the *http_only* flag, we indicate that we want to constantly re-request the same file, without establishing a new connection for each request. This is shown by the following command, where we have 32 parallel connections active for a duration of 60 s:

```
./prog --conn 32 --duration 60 --http_only
```

With the *connect_only* flag, we indicate that we only want to establish a connection, without sending an HTTP request. This is shown by the following command, where we establish 32 connections and close them after a duration of 10 s:

```
./prog --conn 32 --duration 10 --connect_only
```

A.2 List of acronyms

CPU	central processing unit
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
NIC	network interface card
OS	operating system
RSS	resident set size
PTO	Probe Timeout
RAM	random-access memory
RFC	Requests for Comments
RTT	round-trip time
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UDP	User Datagram Protocol

Literatur

[1] Cloudflare, *Cloudflare Radar*, [Last accessed: Sep 5, 2022]. Adresse: `https://radar.cloudflare.com/`.

[2] M. Bishop, *HTTP/3*, [Last accessed: Oct 14, 2022], Juni 2022. Adresse: `https://datatracker.ietf.org/doc/html/rfc9114`.

[3] J. Iyengar und M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, [Last accessed: Oct 14, 2022], Mai 2021. Adresse: `https://datatracker.ietf.org/doc/html/rfc9000`.

[4] Wang, Peng and Bianco, Carmine and Riihijaervi, Janne and Petrova, Marin, "Implementation and Performance Evaluation of the QUIC Protocol in Linux Kernel," 2018.

[5] A. Yu und T. A. Benson, "Dissecting Performance of Production QUIC," Apr. 2021.

[6] T. Berners-Lee, R. Fielding und H. Frystyk, *Hypertext Transfer Protocol – HTTP/1.0*, [Last accessed: Oct 14, 2022], Mai 1996. Adresse: `https://www.rfc-editor.org/rfc/rfc1945`.

[7] R. Fielding, J. Gettys, J. Mogul u. a., *Hypertext Transfer Protocol – HTTP/1.1*, [Last accessed: Oct 14, 2022], Juni 1999. Adresse: `https://www.rfc-editor.org/rfc/rfc2616`.

[8] M. Belshe, R. Peon und M. Thomson, *Hypertext Transfer Protocol Version 2 (HTTP/2)*, [Last accessed: Oct 14, 2022], Mai 2015. Adresse: `https://datatracker.ietf.org/doc/html/rfc7540`.

[9] M. Thomson und C. Benfield, *HTTP/2*, [Last accessed: Oct 14, 2022], Juni 2022. Adresse: `https://datatracker.ietf.org/doc/html/rfc9113`.

[10] J. Postel, *Transmission Control Protocol*, [Last accessed: Oct 14, 2022], Sep. 1981. Adresse: `https://www.rfc-editor.org/rfc/rfc793`.

[11] R. Peon und H. Ruellan, *HPACK: Header Compression for HTTP/2*, [Last accessed: Oct 14, 2022], Mai 2015. Adresse: `https://datatracker.ietf.org/doc/html/rfc7541`.

[12] C. B. Krasic, M. Bishop und A. Frindell, *QPACK: Field Compression for HTTP/3*, [Last accessed: Oct 14, 2022], Juni 2022. Adresse: https://datatracker.ietf.org/doc/html/rfc7541.

[13] K. Oku und J. Iyengar, *Can QUIC match TCP's computational efficiency?* https://www.fastly.com/blog/measuring-quic-vs-tcp-computational-efficiency, [Last accessed: Oct 14, 2022], Apr. 2020.

[14] K. Jacksi, S. Zeebaree und R. Zebari, "Impact Analysis of HTTP and SYN Flood DDoS Attacks on Apache 2 and IIS 10.0 Web Servers," Okt. 2018.

[15] G. Liu, J. Xu, C. Wang und J. Zhang, "A Performance Comparison of HTTP Servers in a 10G/40G Network," Apr. 2018.

[16] A.-P. Barzu, M. Carabas und N. Tapus, "Scalability of a Web Server," Mai 2017.

[17] Meta Platforms, Inc., *Proxygen*, [Last accessed: Oct 14, 2022]. Adresse: https://github.com/facebook/proxygen.

[18] D. Sommermann und A. Frindell, *Introducing Proxygen, Facebook's C++ HTTP framework*, [Last accessed: Oct 14, 2022], Nov. 2014. Adresse: https://engineering.fb.com/2014/11/05/production-engineering/introducing-proxygen-facebook-s-c-http-framework/.

[19] Meta Platforms, Inc., *mvfst*, [Last accessed: Oct 14, 2022]. Adresse: https://github.com/facebookincubator/mvfst.

[20] N. Bawa, *ELI5: Proxygen - High performance HTTP framework*, [Last accessed: Oct 14, 2022], Jan. 2022. Adresse: https://developers.facebook.com/blog/post/2022/01/10/eli5-proxygen-high-performance-http-framework/.

[21] *perf: Linux profiling with performance counters*, [Last accessed: Oct 14, 2022]. Adresse: https://perf.wiki.kernel.org/index.php/Main_Page.

[22] A. Cahalan, *pmap*, [Last accessed: Oct 14, 2022]. Adresse: https://linux.die.net/man/1/pmap.

[23] LiteSpeed Technologies, *lsquic*, [Last accessed: Oct 14, 2022]. Adresse: https://github.com/litespeedtech/lsquic.

[24] T. Tsujikawa, *nghttp2*, [Last accessed: Oct 14, 2022]. Adresse: https://github.com/nghttp2/nghttp2.

[25] *HTTP/2 with curl*, [Last accessed: Oct 14, 2022]. Adresse: https://curl.se/docs/http2.html.

[26] Computer Hope, *File descriptor*, [Last accessed: Oct 14, 2022]. Adresse: https://www.computerhope.com/jargon/f/file-descriptor.htm.

[27] baeldung, *Limits on the Number of Linux File Descriptors*, [Last accessed: Oct 14, 2022]. Adresse: https://www.baeldung.com/linux/limit-file-descriptors.

[28] J. Iyengar und I. Swett, *QUIC Loss Detection and Congestion Control*, [Last accessed: Oct 14, 2022], Mai 2021. Adresse: `https://datatracker.ietf.org/doc/html/rfc9002`.