HAW HAMBURG

Masterarbeit

Marvin Butkereit

Evaluation von modernen Webkommunikationsprotokollen in Kubernetes.

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Marvin Butkereit

Evaluation von modernen Webkommunikationsprotokollen in Kubernetes.

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang *Master of Science Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Franz Korf

Eingereicht am: 13. Juli 2023

Marvin Butkereit

Thema der Arbeit

Evaluation von modernen Webkommunikationsprotokollen in Kubernetes.

Stichworte

Kubernetes, HTTP/3, HTTP/2, HTTP/1.1, Microservices, Cluster, QUIC

Kurzzusammenfassung

Für das Transportprotokoll QUIC wurde eine eigene HTTP-Implementierung erstellt. Diese trägt die Versionsnummer 3. Diese zwei Protokolle sind stark ineinander verzahnt, so enthält das Transportprotokoll QUIC zum Beispiel schon eine Logik zum Handhaben von Streams und die TLS 1.3 Funktionalitäten. Diese Funktionalitäten können von HTTP/3 direkt genutzt werden und daraus entstehen neue Möglichkeiten der Optimierung und der Strukturierung. Das macht es aber auch zu den Vorgängerversion signifikant unterschiedlich, da viele Funktionalitäten, die in HTTP/2 hinzugefügt wurden, nun direkt von QUIC gelöst werden. Auch die Änderung von TCP zu UDP als unterliegendes Transportprotokoll macht einen dieser Unterschiede aus. Desweiteren wird der Gebrauch von Kubernetes als Plattform für verteilte Softwaresysteme immer beliebter, was die Frage aufstellt, ob die aktuellen Implementierungen von HTTP/3 einen Vorteil gegenüber den altbewährten Implementierungen von HTTP/2 und HTTP/1 in diesem Kontext bringen können. In dieser Evaluation soll dieses untersucht werden. Dabei sollen die Unterschiede der konkreten HTTP-Implementierungen aufgezeigt, die Vorteile beschrieben und diskutiert werden, ob es einen Mehrwert bringt, HTTP/3 innerhalb eines Kubernetes Clusters zu verwenden.

Marvin Butkereit

Title of Thesis

Evaluation of modern web communication protocols in Kubernetes.

Keywords

Kubernetes, HTTP/3, HTTP/2, HTTP/1.1, Microservices, Cluster, QUIC

Abstract

A separate HTTP implementation was created for the QUIC transport protocol. It carries the version number 3. These two protocols are strongly intertwined, for example, the QUIC transport protocol already contains a logic for handling streams and the TLS 1.3 functionalities. These functionalities can be used directly by HTTP/3 and new possibilities for optimisation and structuring arise from this. However, this also makes it significantly different from the previous versions, as many functionalities that were added in HTTP/2 are now solved directly by QUIC. The change from TCP to UDP as the underlying transport protocol also accounts for one of these differences. Furthermore, the use of Kubernetes as a platform for distributed software systems is becoming increasingly popular, which raises the question of whether the current implementations of HTTP/3 can provide an advantage over the long-established implementations of HTTP/2 and HTTP/1 in this context. In this evaluation, this will be investigated. The differences between the concrete HTTP implementations will be shown, the advantages will be described and it will be discussed whether there is added value in using HTTP/3 within a Kubernetes cluster.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

ACK Acknowledgment.

AEAD Authenticated Encryption with Associated Data.

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

BGP Border Gateway Protocol.

CIDR Classless Inter-Domain Routing.

CNCF Cloud Native Computing Foundation.

CNI Container Network Interface.

CPU Central processing unit.

CRD Custom Resource Definition.

CRI Container Runtime Interface.

CWND Congestion Window.

CWR congestion windows reduced.

DNS Domain Name System.

DPLPMTUD Datagram Packetization Layer Path MTU Discovery.

ECN Explicit Congestion Notification.

GSO Generic segmentation offload.

HoL Head of Line.

HTTP Hypertext Transfer Protocol.

HTTP/1.1 Hypertext Transfer Protocol Version 1.1.

HTTP/2 Hypertext Transfer Protocol Version 2.

HTTP/3 Hypertext Transfer Protocol Version 3.

IETF Internet Engineering Task Force.

IHL Internet Header Length.

IP Internet Protocol.

IPv4 Internet Protocol Version 4.

IPVS IP Virtual Server.

JSON JavaScript Object Notation.

MPTCP Multipath TCP.

MTU Maximum Transmission Unit.

NAT Network Address Translation.

PAT Port Address Translation.

PMTUD Path MTU Discovery.

QUIC Quick UDP Internet Connections.

RAM Random Access Memory.

REST Representational State Transfer.

RFC Requests for Comments.

RPC Remote Procedure Call.

RTT Round-Trip Time.

SOAP Simple Object Access Protocol.

TCP Transmission Control Protocol.

TCPCRYPT Cryptographic Protection of TCP Streams.

TCPCT TCP Cookie Transactions.

TFO TCP Fast Open.

TLS Transport Layer Security.

TSO TCP segmentation offload.

UDP User Datagram Protocol.

URI Uniform Resource Identifier.

VM Virtuelle Maschine.

VTEP VXLAN Tunnel Endpoint.

VXLAN Virtual Extensible LAN.

YAML Yet Another Markup Language.

1 Einleitung

Die Welt wird immer vernetzter. Immer mehr Services nutzen das Internet, um Kunden zu erreichen [56, 171]. Damit verbunden steigt auch die Erwartung der Endverbraucher an einen Dienst im Internet stetig. Um diesen Anforderungen gerecht zu werden, müssen Anwendungen entwickelt werden, die mehr Funktionalitäten bereitstellen und dabei skalierbarer sind. Dafür werden entweder alte, monolithische Anwendungen zerlegt [39] oder neue, kleine Services erstellt, die als *microservices* [211] bekannt sind. Diese müssen wiederum miteinander kommunizieren, um einen Mehrwert zu bieten. Das geschieht meist über ein Netzwerk und, um es genau zu spezifizieren, über ein Transmission Control Protocol (TCP) [53] stack [235]. Als aktuell sehr populäre Handhabung dieser Kommunikation hat sich dabei Representational State Transfer (REST) [114] oder REST-ähnliche [183, 231] Kommunikation herausgestellt [168]. Dabei wird bei Nachfrage eine Ressource bearbeitet, angefordert oder gelöscht. Dieser Mechanismus wird meistens auf die Hypertext Transfer Protocol (HTTP) Semantik gemappt [61].

HTTP besteht nicht nur aus einem semantischen Teil, sondern besitzt auch eine konkrete Anwendungsprotokoll-Definition [62]. Bislang hat HTTP schon mehrere Iterationen der Entwicklung miterlebt. Von einem textbasierten Protokoll in der ersten Version, zu einer Byte-orientierten in der zweiten Version. Außerdem wurden auch mehrere *streams* erlaubt und viele weitere Funktionalitäten hinzugefügt [208, 13]. Diese vermehrte Entwicklung zeigt das große Interesse an diesem Protokoll. Nun wurde am 6. Juni 2022 Version 3 bei der Internet Engineering Task Force (IETF) [89] standardisiert [13]. Diese Änderung ist im Vergleich zu den vorhergehenden weitaus signifikanter. Im Gegensatz zu den anderen zwei Versionen setzt Hypertext Transfer Protocol Version 3 (HTTP/3) [13] auf User Datagram Protocol (UDP) [143] als Transportprotokoll anstelle von TCP. Wobei als Basis nur UDP für das Protokoll Quick UDP Internet Connections (QUIC) [92] eingesetzt wird.

Das Protokoll QUIC wird auch als Transportprotokoll bezeichnet. Es entstand ursprünglich bei Google LLC [72], wurde dann der IETF zur Standardisierung übergeben und

hat sich innerhalb dieses Prozesses signifikant von der Ursprungsversion verändert [77]. QUIC wurde im Mai 2021 von der IETF standardisiert und stellt damit eine definierte Spezifikation bereit. Die Vorteile von QUIC sind in erster Linie, dass einige Konzepte, die im Web genutzt werden, von z.B. Transport Layer Security (TLS) [160], Hypertext Transfer Protocol Version 2 (HTTP/2) [208] und TCP, auf die Transportschicht kombiniert werden konnten.

Durch die angesprochene Teilung der Anwendung in kleinere Funktionsblöcke bzw. *microservices* ist sie sowohl aus der Perspektive personeller Ressourcen als auch in physikalischer Hinsicht [97] besser aufteilbar. Die gesamte Anwendung kann damit über mehrere Systeme verteilt werden. Dabei entsteht aber die Herausforderung, sie auch verwaltbar zu halten und Deployments zu managen.

Um das zu ermöglichen, wird sowohl in Konzernen als auch kleineren Betrieben Orchestrationssoftware eingesetzt [76, 8, 34]. Ein Beispiel dafür ist das Tool Kubernetes [192]. Dieses kümmert sich darum, dass die Anwendung sich sauber auf mehreren Systemen verteilen lässt, während es für den Anwender so erscheint, als wäre es ein System. Um das zu ermöglichen, benötigt es eine Menge Arbeit, von der Vernetzung bis zur Prozessverwaltung innerhalb dieser Software. Dieses Tool wurde ursprünglich von Google LLC [72] erstellt, wo es basierend auf der Erfahrung mit der internen Softwarelösung Borg erstellt wurde [185]. In der Zwischenzeit ist es der Cloud Native Computing Foundation (CNCF) [194] übergeben worden und zu einem ihrer wichtigsten Projekte geworden. Der Dienst wird von vielen größeren Cloudprovidern direkt bereitgestellt und auch genutzt [18, 195, 8, 199, 184, 34, 76, 117].

Durch die damit entstandene Komplexität der heutigen Cloud Systeme und Umgebungen in Konzernen und im Hostingbereich, soll nun in dieser Evaluation untersucht werden, ob es Vorteile bringt, die Software bzw. *microservices* mit einer aktuellen Implementierung von HTTP/3 [13] innerhalb eines Kubernetes-*clusters* für die Kommunikation zwischen den Softwareteilen zu verwenden oder ob die bislang verbreitete HTTP/2 [208] und Hypertext Transfer Protocol Version 1.1 (HTTP/1.1) [62] dort einen größeren Mehrwert bieten.

Im Folgenden werden die Grundlagen der beteiligten Systeme erörtert. Anschließend wird eine Abgrenzung zu anderen Untersuchungen durchgeführt. Darauf folgt eine Analyse des Umfelds, anschließend werden die Aufbauten von Kubernetes betrachtet. Außerdem wird

das aufgebaute Netzwerk genauer analysiert und der Aufbau des Testsetup beschrieben, gefolgt von der Auswahl der zu evaluierenden Implementierungen. Danach kommt die Evaluation, die die Vergleichskriterien beinhaltet, sowie Aufbau, Experimente, Erwartungen, Ergebnisse und Interpretation der gesammelten Daten. Am Ende gibt es eine Zusammenfassung und einen Ausblick.

2 Grundlagen

Im folgenden Kapitel werden die Grundlagen besprochen. Dafür werden die für die Evaluation relevanten Systemkomponenten kurz erörtert, angefangen bei den Protokollen über die Betriebssysteme, zu den verteilten Systemen und mit anschließenden Blick auf die verwandten Arbeiten.

2.1 Protokolle

In diesem Abschnitt wird auf die verwendeten Protokolle eingegangen. Dabei wird die Struktur des OSI-Modells [236] benutzt. Das bedeutet, es werden schichtweise die benötigten Protokolle vorgestellt und grundliegend für die Evaluation notwendigen Inhalte erklärt. Zuerst wird die Vermittlungsschicht mit Internet Protocol (IP) [145] angeschaut, dann die Transportschicht mit TCP [53], UDP [143] und QUIC [92]. Als nächstes folgt das Sitzungsschicht-Protokoll TLS 1.3 [160]. Darauffolgend werden die Anwendungsschicht-Protokolle HTTP 1 bis 3 [62, 208, 13] betrachtet.

2.1.1 Vermittlungsschicht

In der Vermittlungsschicht gibt es ein signifikantes Protokoll namens IP [145], das meist verwendet wird. Dieses existiert aktuell in zwei verbreiteten Versionen: Version 4 [145] und Version 6 [37]. Im Weiteren wird nur die Version 4 betrachtet, da nur sie für diese Evaluation relevant ist.

IPv4

Version 4 des IP ist im RFC 791 [145] spezifiziert und ist dafür zuständig, in Computernetzwerken mithilfe von Paketvermittlung eine Kommunikation zu ermöglichen. Dafür

stellt das Protokoll einen *header* zu Verfügung, mit dem es möglich ist, die Informationen durch dieses Netzwerk zu transferieren. Dieses Vorgehen wird auch als Routing bezeichnet. Die zu verwendende Nutzlast wird als Datagramm bezeichnet.

Es ist ein *host-to-host*-Protokoll, da es von *host* zu *host* geschickt wird, bis es sein Ziel erreicht. In dem Protokollstandard wird jede Nachricht als unabhängig von der nächsten definiert. Dabei gibt es neben der Adressierung selbst vier Hauptfunktionen, die von dem Protokoll bereitgestellt werden. Diese sind *type of service*, *time to live*, *options* und *header checksum*.

- Das Attribut *type of service* ist eine Ansammlung von Optionen, die es ermöglicht, Informationen an das Routing-System zu geben, um den nächsten Hop zu erreichen. Um genau zu sein, handelt es sich hierbei um *quality of service attributes* [145, p.12].
- *Time to live* entspricht der Lebenszeit eines Pakets. Jedes Mal, wenn ein Router erreicht wird, wird die Anzahl an Leben reduziert. Das ist notwendig, um bei Ausfällen oder falschen Konfigurationen der Routing-Optionen Pakete nicht unendlich lang im Netzwerk zu verschicken [145, p.30].
- Die *Options* ermöglichen es, spezielle Konfigurationen zum Beispiel für das Routing zu konfigurieren.
- Als letzte aufgeführte Option steht die *header checksum*. Diese ermöglicht die Prüfung, ob die versendete Nutzlast richtig verschickt wurde. Für den Fall, das die Prüfsumme nicht valide ist, wird die Nachricht vom erkennenden System verworfen [145, p.3].

Wie durch die Zerstörungen der Pakete in vielen Fällen festzustellen ist, bietet das Protokoll keine zuverlässige Übertragung von Daten an. Es gibt keinen Mechanismus, um sicher zu stellen, dass eine Nachricht angekommen ist. Folglich wird auch keine erneute Übertragung im Falle eines Fehlers angestoßen. Auch sind weder eine *congestion control* noch eine *flow control* vorhanden, was bedeutet, das weder das Netzwerk noch der Empfänger der Nachrichten vor zu vielen Nachrichten geschützt ist.

Die Funktionalität des Protokolls ist einzig und allein das Verschicken von Nachrichten durch eine Ansammlung von Netzwerken. Dieses ist möglich durch das Senden eines Pakets von einem Router zum nächsten bis zum Zielsystem.

In der domainspezifischen Sprache von IP wird zwischen Routen, Namen und Adressen unterschieden. Dabei ist ein Name das, wonach wir suchen, eine Adresse, wo es ist und eine Route der Weg dorthin. Dabei liegt die Zuständigkeit des Internetprotokolls überwiegend bei der Adressierung. Damit wird der Aufgabenbereich klar getrennt: Die Zuständigkeit für Namen wird in höhere Schichten zu Diensten wie Domain Name System (DNS) [122] verschoben und das Routing zu niedrigeren Schichten zu Protokollen wie Border Gateway Protocol (BGP) [159].

<table>
<tr><td colspan="10">0</td><td colspan="10">1</td><td colspan="10">2</td><td colspan="2">3</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>0</td><td>1</td></tr>
<tr><td colspan="4">Version</td><td colspan="4">IHL</td><td colspan="8">Type of Service</td><td colspan="20">Total Length</td></tr>
<tr><td colspan="16">Identification</td><td colspan="4">Flags</td><td colspan="12">Fragment Offset</td></tr>
<tr><td colspan="8">Time to Live</td><td colspan="8">Protocol</td><td colspan="16">Header Checksum</td></tr>
<tr><td colspan="32">Source Address</td></tr>
<tr><td colspan="32">Destination Address</td></tr>
<tr><td colspan="32">Options and Padding</td></tr>
</table>

Tabelle 2.1: *header* eines IPv4 Pakets (nach RFC791)

In der Tabelle 2.1 ist der Aufbau des Internet Protocol Version 4 (IPv4) *headers* zu sehen.

Oben sind Indizies definiert, darunter die Bits und darunter die jeweiligen Elemente. Damit soll einfach verdeutlicht werden, welche Elemente wie viel Platz im *header* benötigen. Im weiteren Verlauf wird dieses Modell mehrfach verwendet, um Kontrollinformationen eines Protokolls darzustellen. Dabei werden immer 4 Bytes pro Reihe dargestellt, also 32 Bits. Das ist auch das Vorgehen der IETF [145] in vielen Requests for Comments (RFC) und *draft* [143, 53]. Dieses Vorgehen wird im weiteren in dieser Arbeit übernommen.

Der *header* besteht aus folgenden Elementen:

Zum Anfang des *headers* gibt es eine Version mit einer Größe von 4 Bits. Dieses zeigt die Version an, die aktuell benutzt wird und ist maßgeblich für das Format des weiteren *headers*. Bei IPv4 ist dieses in der Regel die Version 4.

Das nächste Feld ist die Internet Header Length (IHL), die 4 Bits groß ist. Dieses Feld zeigt in Schritten von 4 Bytes an, wie groß der *header* ist. Deshalb ist der minimale Wert in diesem Feld 5 und das Maximum 15.

Eine *type of service* mit 8 Bits Größe, um *quality of service* Konfigurationen durch-

zuführen, wie in der Aufzählung oben erwähnt.

Eine *total lenght* mit einer Größe von 16 Bits. Dies ist die Länge des gesamten Pakets.

Ein Feld für die *identification* mit 16 Bits, um die Fragmente des Datagrammes wieder zusammenfügen zu können.

Das 4 Bits große *flags*-Feld kann benutzt werden, um einfach unterschiedliche *flag*-Optionen zu setzen, die für die Fragmentierungsmöglichkeit relevant sind [145, p.13].

Ein *Fragment Offset* von 13 Bits ist dazu da, um zu definieren, zu welchen Fragment diese Daten gehören.

Das *time to live* definiert wie oben beschrieben die Lebenserwartung eines Pakets in 8 Bits.

Das nächste Feld heißt *Protocol* und ist 8 Bits groß, es definiert das nächste Protokoll, das innerhalb der Nutzdaten verwendet wird. Dafür gibt es sogenannte *Assigned Numbers* [144], also vordefinierte Identifications für bestimmte Protokolle, die von der IETF in RFC 790 [144] bereitgestellt werden. Dieses wurde aber in heutigen Versionen zu einer Webseite geändert, damit nicht so viele RFC-Updates durch neue Nummern entstehen [55].

Die 16 Bits *header checksum* wurde oben bereits erwähnt. Dieses wird nur von dem *header* berechnet. An jeder Stelle, wo sie bearbeitet wird, wird sie neu berechnet. Das ist nötig, weil sich der *header* allein schon durch das *time to live* stetig ändert.

Die *source address*, die 32 Bits umfasst und die Ursprungsadresse der Nachricht darstellt.

Die *destination address*, die auch 32 Bits umfasst und die Zieladresse der Nachricht darstellt.

Es gibt noch eine weitere variable Anzahl an Bits für Optionen wie *timestamp*, *security* oder *routing information* [145, p.15-23], außerdem muss gegebenenfalls *padding* hinzugefügt werden, so dass ein *header* an einer 32-Bit-Grenze endet. Da beides in dieser Arbeit

nicht verwendet wird, wird der Bereich generalisiert dargestellt.

Ein bislang nicht erwähnter Bereich zum Thema *IP* sind die Netzklassen (*classful network*) [145] und das Classless Inter-Domain Routing (CIDR) [69]. Es ist damit möglich, sogenanntes Subnetting [69] durchzuführen, um eine bessere Routing-Entscheidung zu treffen und das Netzwerk aufzuteilen. Dieses ist aber ein Thema für das Routing und nicht das Adressing, weshalb es hier nicht genauer beleuchtet werden soll. Genauso wird die Fragmentierung nicht genauer behandelt, da sie in heutigen Netzwerken nicht mehr so häufig auftaucht [121].

Aus der Analyse des *headers* resultierend ist der *header* meistens 20 Bytes lang, was auch den minimalen *header* entspricht. Dann sind folglich keine Options oder Padding gesetzt.

2.1.2 Transportschicht

Im folgenden Abschnitt werden die Transportprotokolle TCP [53], UDP [143] und QUIC [92] kurz vorgestellt und auf deren Eigenschaften eingegangen. Dabei werden die für die Protokolle relevanten *header*-Strukturen erörtert, ein kleiner Überblick über im Kontext relevante Funktionalitäten gegeben und die Protokoll-Eigenheiten aufgezeigt.

UDP

UDP ist ein Transportprotokoll, das auch als “kein Transportprotokoll“ bezeichnet wird [143]. Es stellt nur die minimalen Anforderung für ein Transportprotokoll in seinem *header* bereit.

Im Gegensatz zu TCP [53] bietet es keine Garantien für die Übertragung an. Wie erwähnt hat es kaum eigene Funktionalitäten. UDP ist transaktionenorientiert [143, p.1] und bietet von sich aus keine Zustellungssicherheit oder einen Schutz vor doppelten Nachrichten, ebenso ist die Reihenfolge der Nachrichten nicht sichergestellt.

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Soure Port (16)										Destination Port (16)																					
Length (16)										Checksum (16)																					

Tabelle 2.2: *header* eines UDP-Datagrammes (nach RFC768)

In der Tabelle 2.2 ist die Struktur eines UDP-*headers* zu sehen. Dieser besteht aus insgesamt 8 Bytes, diese sind wie folgt strukturiert:

Das Feld *Source Port*, welches 16 Bits groß ist, ist optional und muss nicht vorhanden sein bzw. wird auf 0 gesetzt, wenn es nicht genutzt werden soll. Dennoch bleibt die Feldgröße erhalten. Es ist dafür vorgesehen, eine Rücksende-Adresse mitzuschicken, falls diese Information an keiner anderen Stelle mitgeführt ist.

Der *Destination Port* mit 16 Bits ist der Port, wo das Datagramm auf der Empfängerseite bearbeitet werden soll.

Das nächste Feld *Length* mit ebenfalls 16 Bits Größe, definiert die Größe des gesamten Datagramms, also *header* und Nutzdaten. Dieses wird in Bytes angegeben und ist mindestens 8 Bytes groß, da es der *header*-Größe entspricht.

Das 16 Bits große Feld der *checksum* ist in UDP besonders, da es einen sogenannten *pseudo header* benutzt, um die *checksum* zu berechnen. Dabei greift es auf das darunterliegende IP Protokoll zu, um dort die *source adress*, *destination adress* und *protocol* Information herauszubekommen. Das wird gemacht, um vor falsch gerouteten Datagrammen zu schützen [143, p.2-3].

Es ist auch möglich, die Checksumme auf 0 zu setzen. Dann wird davon ausgegangen, dass keine Checksumme verwendet wird.

Zusammenfassend ist der *header* 8 Byte groß und das Protokoll erweitert die IP lediglich um die Addressierungsmöglichkeiten von Ports.

TCP

TCP[53] ist ein zuverlässiges in-order Byte-*stream*-Protokoll. Dabei werden die Nutzdaten innerhalb eines Byte-*streams* transferiert. Das passiert mithilfe von sogenannten TCP Segmenten. Dabei entspricht jedes Segment einem darunter liegendes Paket [53, sec.2.2].

Die Zuverlässigkeit von TCP wird durch folgende drei Faktoren gewährleistet: Erkennung von verlorenen Paketen über die *sequence number*, die jedes Segment trägt, durch Fehler bei der *segment checksum*, und zuletzt durch *retransmission*. Der TCP-*stream* ist dabei als ein *unicast* gedacht, es ist also eine *end-to-end*-Kommunikation. Dabei ist TCP verbindungsorientiert, hat aber keine *liveness detection*. Der Datenfluss des Protokolls unterstützt bidirektionale Kommunikation. TCP nutzt wie UDP Port-Nummern, um am Zielsystem die Anwendung zu finden.

Hier noch einmal die besonderen Eigenschaften von TCP:

- Flusskontrolle (siehe Abschnitt 2.1.2)
- *congestion control* (siehe Abschnitt 2.1.2)
- Zuverlässigkeit
- Reihenfolgegesichert
- *end-to-end*-Kommunikation

In der Tabelle 2.3 ist der *header*-Aufbau eines TCP-Segments zu sehen, die Definition der Felder folgt direkt darauf.

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Source Port (16)																Destination Port (16)															
Sequence Number (32)																															
Acknowledgment Number (32)																															
Data Offset (4)				Reserved (4)			CWR		ECE		URG		ACK		PSH		RST		SYN		FIN		Window (16)								
Checksum (16)																							Urgent Pointer (16)								
Options (*)																															

Tabelle 2.3: *header* eines TCP-Segments (nach RFC9293)

Source Port (16 Bits)
: Die *source-port*-Nummer (wie bei UDP 2.1.2)

Destination Port (16 Bits)
: Die *destination-port*-Nummer (wie bei UDP 2.1.2)

Sequence Number (32 Bits)
: Die *sequence number* beginnt mit einem zufälligen Anfangswert. Sie wird nach jeder Übertragung um die Größe der Nutzdaten erhöht. Ausnahme ist, wenn das SYN Bit gesetzt ist, dann wird der Initialwert +1 durchgeführt.

Acknowledgment Number (32 Bits)
: Wenn das ACK Bit gesetzt ist, beinhaltet dieses Feld den Wert der nächsten *sequence*-Nummer, die der Sender erwartet. Sobald eine Verbindung aufgebaut ist, wird diese immer versendet. Gleicht dem Prinzip der *sequence number*.

Data Offset (4 Bits)
: Definiert, ab wann der *header* beendet ist. Dabei ist jede Nummer 4 Byte groß.

Reserved (4 Bits)
: Reservierte Bits für zukünftige Funktionalitäten.

CWR (1 Bit)
: Das Feld congestion windows reduced (CWR) wird benutzt, um den Sender anzuzeigen, dass er ein Paket mit einer gesetzten ECE Option erhalten hat. [65]

ECE (1 Bit)
: Diese Option zeigt an, ob das System Explicit Congestion Notification (ECN) unterstützt. [65]

URG (1 Bit)
: Die *urgent*-Option informiert den Empfänger darüber, dass das *urgent*-Paket vor allen anderen Paketen bearbeitet werden muss.

ACK (1 Bit)
: Wird gesetzt, um zu zeigen, dass Pakete erfolgreich empfangen wurden.

PSH (1 Bit)
: Die Push-Option sagt dem Empfänger, dass das Paket direkt verarbeitet werden muss, anstatt es zu buffern.

RST (1 Bit)
: Reset der Verbindung.

SYN (1 Bit)
: Synchronisierung der *sequence*-Nummern.

FIN (1 Bit)
: Es werden keine Daten mehr vom Sender verschickt.

Window (16 Bits)
: Die Anzahl an Bytes, die der Sender bereit ist zu empfangen. Dieser Wert kann durch einen Multiplikator erhöht werden, der im *handshake* ausgetauscht werden kann [17]. Außerdem ist dieses Feld für die *flow control* von TCP notwendig (siehe 2.1.2).

Checksum (16 Bits)
: Wird genau so wie in UDP (siehe *checksum*-Feld in UDP Sektion) anhand des *psuedo headers* generiert. Im Gegensatz zu UDP ist in TCP dieses Feld nicht optional.

Urgent Pointer (16 Bits)
: Ein Pointer auf das Ende der *urgent data*. Dieses Feld wird nur benutzt, wenn auch die URG-*flag* gesetzt ist.

Options (*)
: Die TCP Optionen.

Der *header* ist damit folglich mindestens 20 Byte groß und wird wie bei bislang allen vorgestellten Protokollen bei jeder Nachricht mitgeschickt.

TCP Ablauf

Der Lebenszyklus von TCP ist in drei Phasen aufgebaut: eine Verbindungsaufbau-Phase, eine offene Verbindung und die Verbindungsbeendigungsphase. Dabei ist die Verbindungsaufbau-Phase weiter sehr interessant, da diese notwendig ist, um den initialen Transport von Nutzdaten zu ermöglichen.

In dieser Phase wird der *3-Way-Handshake* [53, sec. 3.5] ausgeführt. Dieser besteht aus drei Elementen und soll helfen, die Problematik des Zwei-Armeen-Problems [79] zu lösen, um eine Verbindung zweier Parteien herzustellen. Das Problem ist nicht lösbar, aber es ist eine gute Annäherung mit dem *3-Way-Handshake* möglich. In der Abbildung 2.1 sind zwei Lebenslinien zu erkennen, eine markiert mit den Namen Server und eine mit Client. Dieses sind die Partner, die eine Verbindung aufbauen wollen. Um das zu erreichen, muss

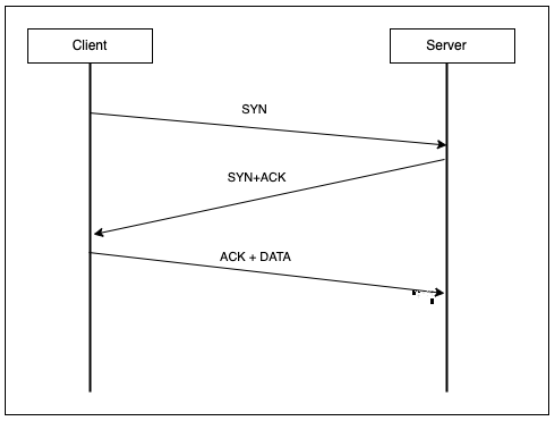

Abbildung 2.1: *3-Way-Handshake* von TCP

der initiierende Partner, in diesem Falle der Client, einen Verbindungsaufbau beim Server erfragen. Dieser antwortet mit einer Bestätigung und einer Verbindungsaufbau-Anfrage, die wiederum durch den Client bestätigt wird. Dieses Verhalten wird durch die Pfeile dargestellt.

Der ganze Prozess dauert damit zwei Round-Trip Time (RTT), um überhaupt die Option zu bieten, seinem Empfänger Information mitteilen zu können. Dieses Verhalten tritt für jeden Verbindungaufbau zu, ob neu oder bereits bekannt spielt dabei keine Rolle. Deshalb wurde eine Erweiterung entwickelt, die es ermöglicht, bei einer erneuten Verbindung den *handshake* zu reduzieren.

TCP Fast Open

TCP Fast Open (TFO) ist eine Erweiterung, die es ermöglicht, die angefragte Seite einen sogenannten Cookie generieren zu lassen, mit dem beim nächsten Verbindungsaufbau der Prozess reduziert werden kann. Dafür wird, wie oben bei dem initialen Verbindungsaufbau, anstatt nur eines SYN auch ein Cookie angefordert, der von der Empfängerseite mitgeschickt wird und der bei dem Initiator und Empfänger gespeichert wird. Es ist vom Aufwand der Kommunikation hier noch kein Gewinn, jedoch kann der Initiator bei der zweiten Anfrage einen Verbindungsaufbau anfordern und seinen Cookie und Nutzdaten direkt mitschicken. Der Empfänger validiert den TFO Cookie und sendet einen Verbindungsaufbau und eine Bestätigung mit seinen Nutzdaten. Damit kann sich bei bereits bekannten Diensten eine RTT gespart werden.

TCP bietet außerdem weitere Extensions wie zum Beispiel TCP Cookie Transactions (TCPCT) [169], Multipath TCP (MPTCP) [66] und Cryptographic Protection of TCP Streams (TCPCRYPT) [15]. Diese sollen nicht genauer betrachtet werden, da sie den Rahmen der Grundlagen übersteigen würden.

QUIC

QUIC ist ein neu entwickeltes Transportprotokoll. Ursprünglich wurde es von Google LLC [72] mit der Absicht designt, ein neues Protokoll zu erstellen, das ideal in der neuen mobilen Welt funktioniert, die mit hohen Latenzen zu kämpfen hat [80, Kapitel 7 p.99-137]. Außerdem sollten Funktionalitäten, die in vielen Anwendungen Standard sind, direkt mit integriert werden, um nicht ähnliche Funktionalitäten zwei Mal durchführen zu müssen und mehreren Ansprüchen gerecht zu werden. Die Implementierung benutzt dabei TLS-, TCP- und HTTP/2-Konzepte.

QUIC ist wie TCP ein zuverlässiges, verbindungsorientiertes Protokoll, bietet aber keine Reihenfolge-Sicherung zwischen allen *streams*. Eine Reihenfolge ist nur auf *streams*-Ebene gesichert, nicht dazwischen [13, sec. 2][102, sec. 1].

Das Protokoll basiert auf einem anderen Transportprotokoll. Es baut auf UDP auf, um die Problematiken des Deployments in der aktuellen Netzwerkwelt zu umgehen, die z.B. durch Network Address Translation (NAT)/Port Address Translation (PAT) Systeme oder andere Middleboxes existieren [31]. Dieses wird auch Protokoll-Ossifikation [174] bezeichnet.

Die folgenden Features zeichnen das Design von QUIC aus:

- Flusskontrolle (siehe Abschnitt 2.1.2)
- *congestion control* (siehe Abschnitt 2.1.2)
- Zuverlässigkeit
- Designt/implementiert in *user space* [225]
- Mehrere *streams* sind möglich
- Die Nachrichten sind verschlüsselt

- Optimiertes *handshake*-Verfahren
- Connection ID als Identifizierung anstelle des 5-Tuples [92, sec. 5.1]

Die Pakete von QUIC sind unterschiedlich aufgebaut. Generell wird zwischen zwei *header*-Typen unterschieden, einem *long header* und einem *short header*. Der *long header* ist in der Tabelle 2.4 zu sehen. Dieser wird nur verwendet, solange die Verbindung nicht aufgebaut worden ist und kein 1-RTT Schlüssel zu Verfügung steht, dazu aber im Abschnitt Verbindungsphasen mehr. Er besteht aus mindestens 7 Byte, die sich wie folgt aufteilen (siehe Tabelle 2.4):

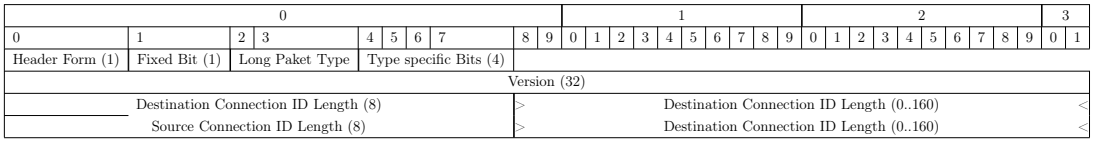

Tabelle 2.4: *header* eines QUIC *long headers* (nach RFC9000)

Header Form (1 Bit)
: Das Bit, das zwischen *long* und *short header* differenziert. Beim *long header* wird es immer auf 1 gesetzt.

Fixed Bit (1 Bit)
: Das Fixed Bit wird zu 1 gesetzt. Eine 0 ist in dieser Version von QUIC nicht erlaubt. Das soll ein koexistierendes Problem mit anderen Protokollen lösen (siehe RFC7983 [141]) [92, sec. 17.2].

Long Paket Type (2 Bits)
: Diese 2 Bits definieren, welches von den aktuell vier *long-header*-Paketen es ist.

Type specific Bits (4 Bits)
: 1 Byte, das abhängig vom Paket-Type benutzt wird.

Version (32 Bits)
: Die Versionsnummer von QUIC. Sie bestimmt, wie das weitere Paket interpretiert wird.

Destination Connection ID Length (8 Bits)
: Dieses Byte gibt die Länge der *Destination Connection ID* in Bytes an.

Destination Connection ID (0 - 160 Bits)
: Die *Destination Connection ID*.

Source Connection ID Length (8 Bits)
: Dieses Byte gibt die Länge der *Source Connection ID* in Bytes an.

Source Connection ID Length (0 - 160 Bits)
: Die *Source Connection ID*.

Es gibt insgesamt vier Arten von *long header* in der Spezifikiation: *Initial*, *0-RTT*, *Handshake* und *Retry*. Dabei passt jede Version den *header* leicht an und hat eine spezielle Nutzlast.

Der zweite *header* hingegen besteht aus mindestens 2 Byte und wird nach dem Verbindungsaufbau verwendet. In Tabelle 2.5 folgt eine Darstellung:

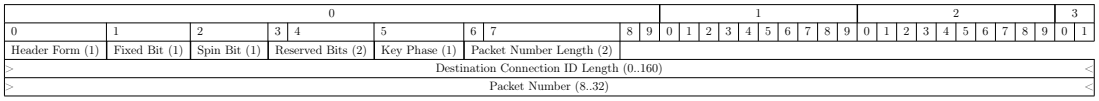

Tabelle 2.5: *header* eines QUIC *short headers* (nach RFC9000)

Header Form (1 Bit)
: Das Bit, das zwischen *long* und *short header* differenziert. Bei dem *short header* wird es immer auf 0 gesetzt

Fixed Bit (1 Bit)
: Das *fixed Bit* wird zu 1 gesetzt. Eine 0 ist in dieser Version von QUIC nicht erlaubt. Das soll ein koexistierendes Problem mit anderen Protokollen lösen RFC7983 [141].

Spin Bit (1 Bits)
: Ist das *latency spin bit*, es kann genutzt werden, um die Latenzen zu messen. Es ist aber in der aktuellen Version optional.

Reserved Bits (2 Bits)
: Reservierter Bits

Key Phase (1 Bit)
: Dieses Bit zeigt, welche *protection keys* verwendet worden sind und switched den Zustand, sobald ein Schlüssel Erneuerung durchgeführt wurde [210, sec.5.4].

Packet Number Length (2 Bits)
: Besteht aus 2 Bits, also bis zu vier Werten. Da das Feld als *unsigned integer* ge-

handhabt wird, ist dieser Wert für die Länge des Paket-Nummernraums immer plus 1 zu nehmen.

Destination Connection ID (0..160 Bits)
: Die *Destination Connection ID*

Packet Number (8..32 Bits)
: Dieses Feld beinhaltet die Paketnummer und wird von der Größe des Packet Number Length Feld bestimmt.

Frames

Innerhalb dieser Pakete werden sogenannte *frames* verwendet, dabei wird mithilfe einer *variable-length integer* [92, sec. 16] in den ersten Bits der Type definiert und danach die Type-spezifische Länge des *frames* benutzt. Dabei können dann in dem *frame*, wie z.B. beim *stream frame*, weitere Angaben zur Länge vorhanden sein.

Type (i) = 0x08..0x0f
Stream ID (i)
[Offset (i)]
[Length (i)]
Stream Data (..)

Tabelle 2.6: Beispielhafter QUIC *stream frame* (nach RFC9000)

In der Tabelle 2.6 ist die Definition eines *stream frames* zu sehen. Dabei stehen die eckigen Klammern für optional. Die ".." kennzeichnen eine variable Größe und die Kennzeichnung mit i steht für *variable-length integer*. In diesen wird bei den ersten zwei Bits die darauffolgende Länge definiert, damit gibt es die Möglichkeit, dass ein Feld zwischen 1 bis 8 Byte groß ist [92, sec. 16]. Hier folgt die Beschreibung der jeweiligen Felder:

Type (i)
: Anhand des Types können das *Offset*-Feld und *Length*-Feld deaktiviert und das Ende eines *streams* definiert werden. Dafür werden die letzten 3 Bits in dem Wert verwendet [92, sec. 19.8]. Deshalb sind dort die Werte von 0x08 bis 0x0f möglich. In binärer Schreibweise wäre das 00001XXX, dabei geben die ersten zwei Nullen die Größe an, also 1 Byte. Die 1 definiert den Type *stream* und die drei X repräsentieren die drei Optionen.

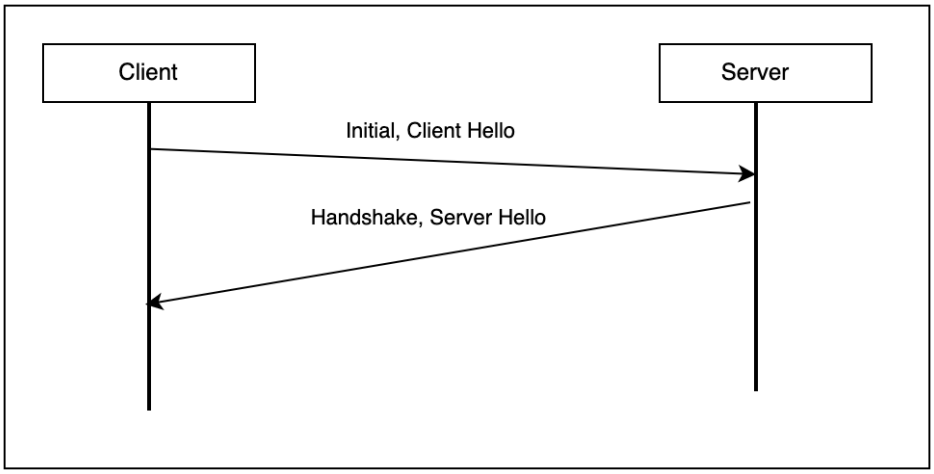

Abbildung 2.2: QUIC *handshake* 1-RTT

Stream ID (i)
: Die ID des Streams.

Offset (i)
: Das Byte-*offset* für die Streamdaten in diesem *frame*.

Length (i)
: Die Länge des *stream-data*-Felds. Wenn dieses 0 gesetzt wird, werden einfach die restlichen Bytes des Pakets als Länge der Nutzdaten angenommen.

Stream Data (..)
: Die *stream*-Daten oder die Nutzdaten, die transferiert werden sollen.

Daraus resultierend ist ein *stream frame* mindestens 2 Byte + Nutzdaten groß. Dies ist nur ein exemplarischer *frame*, es gibt noch viele weitere *frame*-Typen [92, sec. 12.4].

QUIC Ablauf

Das Protokoll ist wie TCP in drei Phasen gegliedert: Die Aufbau-, die Datenübertragungs- und die Abbauphase. Interessant ist dabei die Aufbauphase, da sie aus einem 1-RTT besteht und bei einem weiteren erneuten Verbindungsaufbau aus 0-RTT bestehen kann. Dabei benutzt QUIC für den *handshake* gleich den kryptographischen *handshake* von TLS 1.3 [160] mit.

Um dieses Verhalten zu verdeutlichen, wird in Abbildung 2.2 dieser Sachverhalten bildlich dargestellt. Zu sehen sind zwei Lebenslinien, gekennzeichnet mit Client und Server. Der Client startet einen Verbindungsaufbau, indem er einen *long heander* mit dem Inhalt eines *crypto frames* verschickt, wo die Information für ein *initial hello* enthalten sind. Das wird hier mit einem beschrifteten Pfeil mit der Aufschrift *inital, client hello* dargstellt. Anschließend antwortet der Server mit *handshake, server hello*, woraufhin der Client die Daten schicken kann. Hierbei handelt es sich um eine vereinfachte Darstellung und beinhaltet nicht alle gesetzten *extensions*.

Zu beachten ist: Der 1-RTT ist nur dann möglich, wenn keine *negotiation* des Protokolls oder der Verschlüsselung durchgeführt werden muss. Bei der 0-RTT Variante muss es schon eine intitale Kommunikation gegeben haben, dann wird ähnlich wie bei TFO ein Token direkt zum Server mit dem *client hello* in einen *crypto frame* gesendet, welches neben den Daten liegt und vom Server direkt verarbeitet werden kann, wenn der Token valide ist.

QUIC V2

Im April 2021 [52] wurde der erste *draft* für eine Version 2 von QUIC erstellt. Die letzte Version ist vom 15. Dezember 2022 [51] und einige Implementierung haben angefangen, diese umzusetzen [224]. Es wurden Kleinigkeiten innerhalb des Protokolls geändert, um z.B. einige der Protokoll-Ossification-Probleme zu lösen.

Auch bei QUIC ignorieren wir einige größere Features wie z.B. die Verbindungsmigration. Außerdem gibt es wie bei TCP einiges an Erweiterungen, von *multipath* [110] bis hin zu *unrealibe streams* [139], diese sollen hier nicht weiter erörtert werden.

Gemeinsame Informationen

Im folgenden Abschnitt sollen Konzepte vorgestellt werden, die innerhalb QUIC und innerhalb TCP verwendet werden oder Einflüsse auf deren Verhalten haben.

Congestion Control

Um das Netzwerk vor Überlast zu schützen, verwenden QUIC und TCP sogenannte

congestion-control-Algorithmen. Von diesen existieren eine große Anzahl an verschiedensten Variationen [19, 162, 81]. Sie können meistens in *lost-basiert* oder *delay-basiert* [70] eingeordnet werden. Die Einordnung geschieht hier anhand der Erkennung einer Überlast, also entweder anhand von verlorenen Daten oder der erhöhten Antwortzeiten. Natürlich gibt es auch Zwischenlösungen oder andere Ansätze, aber ein großer Teil der Algorithmen ist diesen Kategorien einzuordnen [229]. Dabei ist ein Ziel von *congestion control* Algorithmen, *tcp friendly* [230] zu sein. Das bedeutet, sie sollen sich fair gegenüber anderen TCP-artigen *streams* verhalten. Der zweite Faktor ist, den maximalen Durchsatz zu ermöglichen.

Der meistverbreitete und aktuell häufig als Standard eingestellte *congestion-control*-Algorithmus ist Cubic [162]. Cubic wurde anhand dieser Designprinzipien entworfen:

1. Prinzip für bessere Netzwerkausnutzung und Stabilität. Um das zu erreichen, existieren die Verhaltensweisen *concave* und *convex* innerhalb des Cubic-Algorithmus, um das *congestion window* zu vergrößern, anstatt nur eine *convex*-Funktionslogik.
2. Prinzip, um TCP-freundlich zu bleiben. Dafür ist Cubic designt worden, um sich so zu verhalten, wie normaler TCP-Traffic in Netzwerken mit kurzen RTT und geringer Bandbreite, wo das normale TCP gut funktioniert.
3. Prinzip für RTT-Fairness. Cubic ist designt, um linear Bandbreiten-Verteilung zwischen verschiedenen *flows* mit unterschiedlichen RTTs zu ermöglichen.
4. Prinzip Balance: Cubic setzt seinen *multiplicate window decrease* Faktor ein, um zwischen Skalierbarkeit und *convergence*-Geschwindigkeit zu balancieren.

Abhängig vom aktuellen Zustand des Congestion Window (CWND) läuft Cubic in drei verschiedenen Modi:

1. Der TCP-freundliche Modus, welcher sicherstellt, dass Cubic mindestens den selben Durchsatz hat wie Standard TCP.
2. Der *concave*-Modus, wenn TCP nicht in der freundlichen Region ist und das CWND ist kleiner als W_max.
3. Der *convex*-Modus, wenn TCP nicht in der freundlichen Region ist und das CWND ist größer als W_max

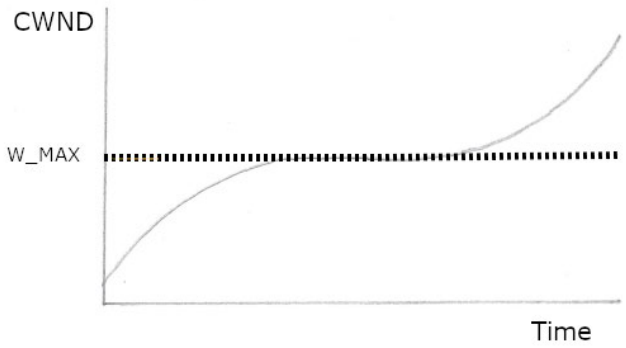

Abbildung 2.3: *congestion control* Cubic Verlauf

Dabei ist W_max die maximale Größe, bevor das Fenster nach dem letzten *congestion*-Event reduziert wird.

Am Anfang der Verbindung wird ein *slow start* ausgeführt, um das Limit zu finden und W_max zu setzen. Dabei ist der *slow-start*-Algorithmus nicht fest definiert [162, sec. 4.8].

Die *slow-start*-Algorithmen wie z.B. SlowStart (Standard) [16, 3.1], SlowStart Limited [64], Hystart++ [11], Hystart [82] oder andere sind möglich. Dadurch kann auch eine gewisse Varianz im Verhalten entstehen.

In der Abbildung 2.3 ist ein idealisierter Verlauf von Cubic zu sehen. Die horizontale Achse bildet den Zeitverlauf (*Time*) ab, die vertikale Achse CWND. Außerdem sehen wir W_max. Die Kurve bildet einen typischen Cubic-Verlauf ab. Es wird erst eine *concave* Wachstumsphase unter W_max durchgeführt, anschließend wird W_max gehalten und darauffolgend ein *convexes* Wachstum durchgeführt.

Flow Control

Die *flow control*, oder Flusskontrolle, ist die Funktionalität von Protokollen zur Anpassung der zu übertragenden Datenmengen. Diese Regulierung findet bei TCP auf

Protokoll-Level statt und auf *multistream*-Systemen wie QUIC und HTTP/2 auf Protokoll-Level und *stream*-Level.

Die Empfänger müssen die Menge an empfangbaren Daten limitieren, um ihre Buffer optimal zu nutzen. Dabei haben die Protokolle leicht unterschiedliche Vorgehen, die im folgenden kurz erörtert werden sollen.

QUIC benutzt eine *limit-based flow control* [92, sec. 4.1], bei der der Empfänger informiert wird, wie viel er auf einem *stream* oder der gesamten Verbindung empfangen kann. Dabei wird beim *handshake* ein initiales Limit für alle *streams* durch ein Transport Parameter im TLS [92, sec. 18,18.2] gesetzt. Ab diesem Zeitpunkt sendet der Empfänger MAX_STREAM_DATA-*frames* oder MAX_DATA-*frames*, um mitzuteilen, wie viel er empfangen kann. Der Empfänger dokumentiert die empfangenen Daten auf allen *streams* und überprüft auf falsches Verhalten des Senders. Wenn ein falsches Verhalten erkannt wird, muss die Verbindung beendet werden mit einem Error. Der Empfänger kann auch nachträglich geringere Limits versenden, diese werden dann aber vom Sender ignoriert. Falls beim Sender *flow-control limited blocked* ist, also kein Senden mehr möglich ist, weil das Limit nicht erhöht wurde, sollten regelmäßig STREAM_DATA_BLOCKED oder DATA_BLOCKED *frames* verschickt werden.

Das Protokoll TCP nutzt ein *sliding window protocol* [177, p.565-568]. Das Empfängerfenster zeigt an, wie viel Daten der Empfänger bereit ist zu akzeptieren. Der Sender sendet bis zur maximalen *window size* seine Segmente in numerischer Reihenfolge. Sobald das erste Segment am Anfang *acknowledged* wurde, wird vom Sender ein weiteres Paket vom Ende des Fensters gesendet.

Ein Beispiel: Eine Nachricht ist ein Byte groß und der Empfänger erlaubt, drei zu erhalten. Der Sender schickt fünf Nachrichten, die ersten drei kann er direkt schicken. Sobald der Sender das Acknowledgment (ACK) für die erste Nachricht erhalten hat, kann er die vierte Nachricht senden. Das Fenster hat sich also um einen Byte verschoben. Dieses Beispiel ist ein vereinfachtes Verhalten und soll nur die Funktionalität kurz erörtern.

In HTTP/2 managed der Empfänger, wie viele Bytes er empfangen möchte, sowohl auf *stream level* als auch für die gesamte Verbindung. Der Vorgang basiert auf dem WINDOW_UPDATE *frame*. Der Sender muss das gesetzte Fenster respektieren und der Empfänger ist frei, es auszufüllen. Dabei gibt es keine *end-to-end*-Kontrolle – jede

beteiligte Partei managed seine eigene *flow control*. Die *flow control* ist nur gültig für Data-*frames* nach dem RFC [208, sec. 5.2.1 p.5]. Ein Endpoint kann *flow control* auch deaktivieren, muss aber Signale von den anderen weiterhin behandeln. Im Gegensatz zu anderen *flow-control*-Definitionen wird hier nicht definiert, wann ein WINDOW_UPDATE durchgeführt werden soll, es stellt nur das Format und die Semantik bereit.

2.1.3 Sitzungsschicht

Die Sitzungsschicht ist gemäß des OSI-Modells [236] dafür zuständig, einen Kommunikationkanal zwischen zwei Systemen herzustellen, über die weitere Kommunikation stattfinden kann. Ein Protokoll, welches diese Aufgabe erfüllt und außerdem die Kommunikation verschlüsselt, ist TLS [160]. Dieses soll im Folgenden kurz erörtert werden.

TLS

TLS [160] wird eingesetzt, um eine verschlüsselte Datenübertragung zwischen zwei Systemen herstellen zu können. Es gibt unterschiedliche Versionen von TLS [41, 161, 160]. Da QUIC zum aktuellen Zeitpunkt nur TLS 1.3 verwendet, soll sich dieses hier genauer angeschaut werden. Dabei besteht das Protokoll aus zwei Phasen oder zwei Unterprotokollen, *handshake* [160, sec. 4] und *record* [160, sec. 5]. TLS setzt voraus, dass das Protokoll, auf dem es aufsetzt, zuverlässig und *in-order*-Daten-*streams* bereitstellt. TLS ist unabhängig vom Anwendungsprotokoll, das Anwendungsprotokoll muss sich aber darum kümmern, es zu initialisieren und zu verwenden.

Die Phase *handshake* wird benutzt, um die Kommunikationspartner gegenseitig zu authentifizieren, kryptographische Regeln und den Schlüssel für die weitere Kommunikation auszutauschen. Für einen beispielhaften *handshake* siehe Abbildung 2.4.

Dort zu erkennen sind zwei Lebenslinien, wieder gekennzeichnet mit Client und Server. Der Initiator, also der Client, sendet ein *client hello* mit einigen *extensions* wie *key share request* [160, sec. 2], dann antwortet der Server mit einem *server hello* und weiteren *extensions*. Anschließend können die Daten über das *record*-Protokoll verschlüsselt versendet werden.

Es ist sehr ähnlich der Abbildung wie bei QUIC (sehe Abbildung 2.2), da es genau das

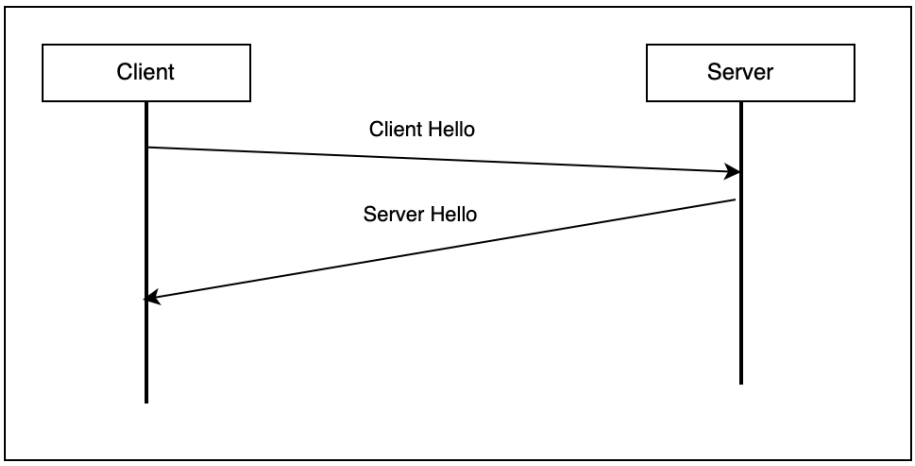

Abbildung 2.4: Verbindungsaufbau TLS

gleiche Verhalten zeigt – mit einem kleinen Unterschied, der aus dem Bild nicht einfach ersichtlich wird: Dass bei QUIC die Nachrichten in *frames* versendet werden, also transportprotokollspezifisch, während die Übertragung der Nachrichten hier im Sitzungsschicht-Protokoll unabhängig vom Transport durchgeführt wird. Auch diese Darstellung des Protokolls ist sehr vereinfacht, um nur die nötigsten Mechanismen darzustellen.

Dies ist das Protokoll, welches in QUIC verwendet wird, darum existiert auch hier eine 0-RTT-Variante, die es ermöglicht, direkt Daten zu versenden. Diese hat das gleiche Verhalten wie im Beispiel von QUIC erörtert (siehe Abbildung 2.2).

2.1.4 Anwendungsschicht

Im Folgenden soll die aktuell im Web hochrelevante Protokoll-Familie der Anwendungsschicht vorgestellt werden: HTTP [62, 208, 13]. Diese Protokolle können als ein Grundgerüst des heutigen Internets bezeichnet werden. Fast jede Webseite wird heutzutage mit dieser Protokoll-Familie bereitgestellt [22]. HTTP ist als ein zustandsloses Anfrage/Antwort-System definiert.

HTTP-Semantik

Es gibt die konkreten HTTP-Implementierungen, die unterschiedlichste Funktionalitäten bereitstellen, und es gibt die Semantik, wie diese strukturiert sein sollen. Hier soll ein Ausschnitt der Semantik grundlegender Elemente mit häufiger Verwendung vorgestellt werden [61].

Das HTTP ist eine Familie aus zustandslosen Anfrage/Antwort-Protokollen auf Anwendungsebene, die eine gemeinsame Schnittstelle haben. Diese hat eine erweiterbare Semantik und besteht aus selbst-beschreibenden Nachrichten. Diese Familie erlaubt, flexible Interaktionen mit Netzwerk-basierten Hypertext-Informationssystemen zu führen.

HTTP ist ein *client-server*-Protokoll, das über ein zuverlässiges Transportprotokoll oder eine *session-layer*-Verbindung läuft.

In HTTP wird zwischen *client* und *server* unterschieden. Ein *client* ist ein Programm, das eine Verbindung zu einem *server* aufbaut, um eine oder mehrere HTTP-Anfragen zu stellen. Der *server* hingegen nimmt diese Anfragen entgegen, bearbeitet sie und sendet eine Antwort zurück.

Jede Version von HTTP definiert seine eigene Syntax für die Message-Kommunikation. Es gibt aber eine gemeinsame, abstrakte Basis für diese Nachrichten. Damit wird ein Vertrag erstellt, der unabhängig von der HTTP-Version ist, so dass eine Nachricht, die über eine Version geschickt wurde, genau die gleiche Bedeutung hat, wie die Nachricht, wenn sie über eine andere Version gesendet worden wäre.

Eine Nachricht besteht aus folgenden Elementen:

- *control data*, um die Nachricht zu beschreiben und zu Routen.
- Einem *header* mit *key-value*-Paaren, um die *control data* zu erweitern und weitere Information über die Nachricht, den Inhalt oder den Kontext mit zu senden.
- Dann einen Inhalt, der verschickt wird.
- Optional ein *trail* mit *key-value*-Paaren, um Information zu transferieren, die während des Senden des Inhalts gesammelt worden sind.

Das *framing*, das für jede HTTP-Version unterschiedlich ist, soll erkennbar machen, wo Anfang und das Ende der Nachricht sind. *control data* wird zuerst verschickt, gefolgt von der *header field section*. Dann wird der Inhalt gesendet, falls es dieses gibt. Gegebenfalls wird danach noch der *trail* versendet.

Es wird erwartet, dass Nachrichten als ein *stream* behandelt werden. *Control data* beschreibt, was der Empfänger direkt über die Nachricht wissen muss. *header fields* geben die Information, die benötigt werden, bevor der Empfänger den Inhalt empfängt. Dann folgt der intendierte Inhalt. Und zum Schluss kommt das Feld *trailer field*, das optionale Metadaten enthält, die vor dem Senden des Inhalts nicht bekannt waren.

Nachrichten sollen selbsterklärend sein. Alles, was der Empfänger wissen muss, sollte beim Betrachten der Nachricht erkennbar sein, auch ohne Wissen über den Zustand des Senders selbst.

Die eben vorgestellte Semantik der Nachrichten kann unterschiedlich implementiert werden, je nach der konkreten Versionsbeschreibung dieser Semantik. Diese konkrete Versionsbeschreibung ist RFC-spezifisch und wird in den jeweiligen Abschnitten behandelt.

Hier folgen nochmal Beispiele eines *requests*, der in der HTTP-Semantik definiert ist.

Bei einem *request* vom *client* wird die semantische Beschreibung: *method request-target version* für die *control data* verwendet. Diese drei Begriffe beschreiben erstens die Methode einer der untenstehenden definierten Methoden, zweitens die Resource, die nachgefragt wird und drittens die Protokoll-Version [61, sec. 6.2].

Dabei sind die Methoden wie folgt definiert:

GET fordert die ausgewählte Ressource an.

HEAD fordert die ausgewählte Ressource ohne die Daten an.

POST schickt Daten, die verarbeitet werden sollen.

PUT ersetzt die bereits vorhandene Ressource mit der gesendeten.

DELETE löscht die ausgewählte Ressource.

CONNECT erstellt einen Tunnel zu der Ressource.

OPTIONS gibt die Kommunikationsoptionen für die Ressource zurück.

TRACE ist ein *loop-back interface* für Tests.

Interessant ist dabei, dass PATCH nicht zu den eigentlichen Methoden gehört, die in der Semantik beschrieben sind, da in der REST-Semantik dieses verwendet wird [114]. Aber bis auf GET und HEAD sind alle anderen Methoden auch optional.

Nun ein Beispiel für die Semantik bei einem HTTP-*response*:

Als erste Information in den *control data* eines *response* ist der *status code* definiert [61, sec. 15]. Gefolgt von einer optionalen *reason phrase* und der Version. Dabei ist der *status code* eine Nummer. Dieser Nummernbereich wird in der Semantik in verschiedenen Bereichen eingeteilt:

- *Informational*: Der Bereich im 100er Raum.
- *Successful*: 200er Raum.
- *Redirection*: 300er Raum.
- *Client Error* 400er Raum.
- *Server Error* 500er Raum.

Auch dieses Feld kann wie das *method*-Feld um weitere Error-Codes [61, sec. 16.2] erweitert werden. Die *reason phrase* ist meist der Name des *status code*, wenn dieses Feld gesetzt ist. Die Version ist die Protokoll-Version, die genutzt wird. Die Reihenfolge der einzelnen Elemente ist hier nicht genauer spezifiziert.

HTTP/1.1

Die Version HTTP 1 ist die erste Iteration eines Protokolls dieser Semantik. Wir werden uns hier die letzte Revision Version 1.1 anschauen, da diese heutzutage immer noch weit verbreitet ist [163]. In HTTP/1.1 wird mit Nachrichten kommuniziert. Das bedeutet im Gegensatz zu den bisher beschriebenen Protokollen, dass es American Standard Code for Information Interchange (ASCII) Symbole in der Größe von einem Byte zur Kommunikation nutzt [62].

```
GET / HTTP/1.1
HOST: master.thesis.example
```

Abbildung 2.5: Beispielhafter HTTP1.1 *request*

```
method SP request-target SP HTTP-version CRLF
```

Abbildung 2.6: Beispielhafter HTTP1.1 *request line*

Eine Standard-*request* sieht dann etwa aus, wie in der Abbildung 2.6 zu sehen ist.

Wie wir in der Semantik gesehen haben, ist das der Aufbau der *control data* von einem *request*, nur in einer konkreten, versionsspezifischen Ausführung. Das SP ist das ASCII Symbol für space. *method*, *request-target* und *version* entsprechen genau der *request*-Semantik. Das CRLF *carriage return* entspricht einem Zeilenumbruch zum Ende.

Ein echter beispielhafter *request* in HTTP/1.1 ist in Abbildung 2.5 zu sehen. Dort wird die Methode "GET" verwendet, um die Ressource "/" zu erhalten, dabei soll die Version HTTP/1.1 verwendet werden. Außerdem wird der *host* in dem *header*-Feld versendet, um das Routing korrekt durchführen zu können.

In der Abbildung 2.7 sehen wir einen beispielhaften *response* von einem *server*, dabei sind die Version und der *status code* sichtbar, gefolgt von der *reason phrase* und einigen *header*-Informationen. Der folgende *body* wurde aus Platzgründen weggelassen und es wurden keine *trail*-Daten versendet.

```
HTTP/1.1 200 OK
Date: Fri, 17 Mar 2023 13:25:40 GMT
Server: Apache/2.4.56 (Unix)
Last-Modified: Mon, 11 Jun 2007 18:53:14 GMT
ETag: "2d-432a5e4a73a80"
Accept-Ranges: bytes
Content-Length: 45
Content-Type: text/html
```

Abbildung 2.7: Beispielhafter HTTP1.1 *response*

Wie im Beispiel gesehen, sendet ein *client* A eine Anfrage an *server* B und erhält eine Antwort. Interessant ist dabei, dass alle *header*-Information in ASCII- und *control*-Symbolen übertragen werden und jeweils ein Byte benutzen. Das bedeutet, dass ein extrem großer Überhang entsteht – allein dadurch, dass einfache Aussagen nicht als Bit oder Byte kodiert werden. Dieses Problem relativiert sich aber dadurch, dass im Anwendungsprotokoll die *header* nur einmal transferiert werden müssen, was bei darunterliegenden Protokollen nicht der Fall ist.

HTTP/2

Ein verbesserter Transport für *HTTP*-Semantik [208] ist der Versuch, die bisherige Version von HTTP effizienter zu machen. Dafür benutzt das HTTP/2 genannte Protokoll genau wie TCP, QUIC oder UDP eine Bitcodierungslogik anstelle von ASCII-Symbolen. Außerdem wurde eine *stream*-Abstraktion hinzugefügt, welche ein *multiplexing* über mehrere *streams* ermöglicht. Diese *streams* sind unabhängig voneinander und blocken sich nicht gegenseitig. Sie haben wie in Abschnitt 2.1.2 erwähnt eine *flow control*. Es wurde außerdem die Möglichkeit geschaffen, Priorisierung zu setzen.

HTTP/2 führt *binary framing* und *multiplexing* von *streams* ein, um die Latenzen zu optimieren, ohne dabei die Transportschicht verändern zu müssen. Damit verbunden ist aber auch, dass die darunterliegende Schicht nichts von den unterschiedlichen *streams* weiß. Daraus kann resultieren, dass das darunterliegende Protokoll ein Head of Line (HoL) Blocking für alle darüberliegenden *streams* bewirkt.

Anders als bei HTTP/1.1 wird hier die Kommunikation über *streams* durchgeführt. Dafür fügt HTTP/2 sogenannte *frames* ein. In der Tabelle 2.7 ist die Grundstruktur eines *frames* zu sehen und wird im folgenden kurz erörtert:

Length (24)
Type (8)
Flags (8)
Reserved (1)
Stream Identifier (31)
Frame Payload (..)

Tabelle 2.7: Beispielhafter HTTP/2 *frame* (nach RFC9113)

Length (24 Bits)
: Die Länge des *frame payload*, ohne den *header* selbst.

Type (8 Bits)
: Die nächsten 8 Bits definieren den *frame*-Type. *Frames*, die nicht definiert sind, werden einfach ignoriert oder verworfen (Eine Liste aller *frames* in diesem RFC sind in [208, sec. 6] zu finden).

Flags (8 Bits)
: Das darauf folgende Byte ist für *flags* reserviert. Diese sind abhängig vom *frame*-Type.

Reserved (1 Bit)
: Ein reserviertes Bit, das auf 0 gesetzt sein muss und beim Empfangen ignoriert wird.

Stream Identifier (31 Bits)
: Ein *Stream Identifier* in 31 Bits, wobei der 0 *stream identifier* reserviert für den *connection control stream* ist.

Frame Payload
: Individuelle *payload* vom spezifischen *frame*.

Aus dieser Aufstellung ergibt sich, dass ein *frame* per Definition mindestens 9 Bytes groß ist. Außerdem sind die Struktur und der Inhalt des *frame payloads* komplett abhängig vom jeweiligen *frame*-Type.

HTTP/2 baut zwei *streams* auf, erstens für eine Kommunikation einen globalen *connection control stream* und zweitens einen *stream* für die Anwendung, über den die individuelle Kommunikation stattfindet [208, sec. 5.1.1].

Auch die in HTTP/1.1 an erster Stelle gestellte Anfrage-Zeile wird hier anders behandelt. Dafür werden sogenannte *pseudo header fields* hinzugefügt, welche mit dem Symbol ":" (ASCII 0x3a) beginnen. Es wird explizit darauf hingewiesen, dass diese *header*-Information keine HTTP-*header* sind [208, sec. 8.3]. Um das auf einfache Art zu erörtern, soll folgend ein Beispiel aus dem RFC9113 gezeigt werden, wo ein Mapping von HTTP/1.1 zu HTTP/2 gezeigt wird.

```
GET /resource HTTP/1.1              HEADERS
Host: example.org          ==>        + END_STREAM
Accept: image/jpeg                    + END_HEADERS
                                        :method = GET
                                        :scheme = https
                                        :authority = example.org
                                        :path = /resource
                                        host = example.org
                                        accept = image/jpeg
```

Abbildung 2.8: Beispielhaftes HTTP1.1 zu HTTP/2 *request*-Mapping aus dem RFC9113

```
HTTP/1.1 304 Not Modified           HEADERS
ETag: "xyzzy"              ==>        + END_STREAM
Expires: Thu, 23 Jan ...              + END_HEADERS
                                        :status = 304
                                        etag = "xyzzy"
                                        expires = Thu, 23 Jan ..
```

Abbildung 2.9: Beispielhafter HTTP1.1 zu HTTP/2 *response*-Mapping aus dem RFC9113

In der Abbildung 2.8 ist auf der linken Seite ein *request* in HTTP/1.1 abgebildet, auf der rechten Seite ein *request* in HTTP/2. Zu erkennen ist, dass die *request lines* komplett in die *pseudo header* verschoben wurden und die *host* und *accept header* weiterhin wie bei HTTP/1.1 in der *header*-Sektion vorhanden sind. Das rechte Beispiel soll einen *header frame* darstellen, in dem die aufgelisteten *header* vorhanden sind. Die mit "+" gekennzeichneten Werte sind Felder, die in dem *frame* gesetzt sind.

Das gleiche Phänomen wie mit den *pseudo headern* ist auch in der *response* zu sehen. In der Abbildung 2.9 ist auf der linken Seite wieder der HTTP/1.1-*response* zu sehen, auf der rechten Seite der HTTP/2-*response*. Verhalten und Struktur sind die gleichen wie beim *request*. Besonders ist hier, dass die Version wegfällt, da sie durch das Format impliziert wird. Auch der Text für den *status code* entfällt.

Werfen wir einen detaillierteren Blick auf die *frames*: Es gibt einen Data-*frame*, in dem die Nutzdaten transportiert werden. Dieser ist in der Tabelle 2.8 zu sehen und soll hier kurz vorgestellt werden:

Length (24)
Type (8) = 0x00
Unused Flags (4)
PADDED Flag (1)
Unused Flags (2)
END_STREAM Flag (1)
Reserved (1)
Stream Identifier (31)
[Pad Length (8)]
Data (..)
Padding (..2040)

Tabelle 2.8: Beispielhafter HTTP/2-Data-*frame* (nach RFC9113)

Bis zum *Stream Identifier* ist dieser *frame* aufgebaut wie der Standard-*frame* aus 2.7, weshalb im folgenden nur die neuen Felder vorgestellt werden.

Pad Length (8 Bits)
: Ein 8 Bit großes, optionales Feld, welches die Länge des Paddings in Bytes definiert.

Data
: Die Daten, die transferiert werden sollen.

Padding
: Das Padding Feld.

Die zwei *flags*, die hier hinzugefügt wurden, sind einmal die *flag* PADDED (0x08), welche anzeigt, dass das Feld *Pad Length* vorhanden ist und dass ein *Padding* definiert werden kann, und die *flag* END_STREAM (0x01) welche anzeigen kann, dass dieser *frame* der letzte *frame* für diesen *stream* ist.

Daraus resultierend ist im optimalen Fall ein Data-*frame* genau so groß wie ein Standard-*frame*.

Hpack

In der ersten Version von HTTP waren die *Fields* noch nicht komprimiert. Das führte dazu, dass bei mehreren *requests* die redundanten Felder unnötiger Weise Bandbreite

konsumierten und außerdem die Latenz erhöht wurde [208, sec. 1].

Um dieses Problem zu lösen, wurde Hpack [140] erstellt. Hpack wurde so designt, dass es unflexibel ist, aber simpel zu implementieren. Dadurch reduziert sich die Gefahr, dass es falsch implementiert wird und dadurch nicht mehr kompatibel mit anderen Version ist. Außerdem wird dadurch die Sicherheit erhöht. Daraus resultierend ist kein Erweiterungsmechanismus hinzugefügt worden, was bedeutet, dass Veränderungen nicht möglich sind, ohne dass ein kompletter Ersatz durchgeführt werden muss [140, sec. 1].

Die Definition von Hpack behandelt eine Liste von Feldern als eine *ordered collection* von *key-value*-Paaren, die doppelte Einträge enthalten können [140, sec. 1.1]. Dabei ist die Ordnung der Felder vor und nach der Kompression gleichbleibend.

HTTP/3

Die Implementierung von HTTP/3 erlaubt es, die HTTP Semantik über QUIC zu transportierten. Dadurch ist es möglich, viele Funktionen aus QUIC weiterzuverwenden, wie *stream multiplexing* oder *flow control*. Diese wurden in HTTP/2 extra hinzugefügt, weil TCP es nicht unterstützt hat. Dadurch, dass QUIC *congestion control* auf der gesamten Verbindung und *reliability* auch auf dem *stream level* bereitstellt, bietet es HTTP/3 die Möglichkeit, eine Performance-Verbesserung zu erhalten, im Gegensatz zum Mapping auf TCP, wie es HTTP/2 macht.

Das Design von HTTP/3 gleicht damit sehr dem Design von HTTP/2, nur dass viele Funktionalitäten dem Transportprotokoll überlassen werden, *Stream lifetime* und *flow control* zum Beispiel. Das *binary framing* ist ähnlich wie bei HTTP/2, welches auf jeden *stream* genutzt wird.

In der Spezifikation von HTTP/3 ist es ein wenig schwammig formuliert, aber es soll eine direkte QUIC-Verbindung zu der Uniform Resource Identifier (URI) hergestellt werden. Dazu ist ein unbestimmter UDP-Port notwendig, da er nicht explizit im RFC spezifiziert worden ist. Meist wird dort der HTTPS-Standard Port “443“ verwendet. Falls das nicht geht, sollte versucht werden, eine TCP-Verbindung aufzubauen. Dort wird dann mithilfe von HTTP/1.1 oder HTTP/2 eine alternative Lösung verwendet, und zwar wird ein *response header field* namens *Alt-Svc* versendet. Dieser beinhaltet z.B. den Wert

h3=":50781", wobei dann bei dem nächsten *request* eine HTTP/3 Verbindung mit dem angegebenen Port aufgebaut werden kann.

HTTP/3 erstellt mehrere *streams*: Einen für jede Kommunikation und einen für die Kontrollinformationen, die für die gesamte Verbindung gelten. Außerdem kann ein *push stream* nach der Spezifikation erstellt werden, das soll aber im Weiteren nicht genauer beleuchtet werden, da die Funktion nicht verwendet wird [13, sec. 7]. Innerhalb dieser *streams* wird mit *frames* kommuniziert, die innerhalb der *payload* eines QUIC-*streams* versendet werden. Dabei ist zwischen den *frames* von HTTP/3 und den von QUIC zu differenzieren. Jedes Anfrage-Antwort-Paar benutzt einen separaten QUIC-*stream*, diese sind unabhängig voneinander und blocken sich deshalb nicht gegenseitig. [13, sec. 7.1]

Die Spezifikation nutzt genau wie HTTP/2 den Mechanismus *pseudo header*. Weil die Kommunikation semantisch sehr ähnlich ist, wird sie im Folgenden weggelassen. Folgend sollen nun der *frame* in HTTP/3 vorgestellt werden. Die Struktur ist in Tabelle 2.9 zu sehen.

Type (i)
Length (i)
Frame Payload(...)

Tabelle 2.9: Beispielhafter HTTP/3-*frame* (nach RFC9114)

Wie bei QUIC-*frames* wird hier das Prinzip *variable-length Integer* verwendet. Diese sind in der Abbildung mit i gekennzeichnet. Der Type definiert den *frame*-Type. Die *Length* ist die Länge des *payloads* und der *payload* ist der spezifische Inhalt des *frames*. Wir haben hier also mindestens 2 Byte an *header*-Informationen.

Um nun spezifischer zu werden, soll sich auch der Data-*frame* angeschaut werden, genau wie bei HTTP/2. Dieser ist in der Tabelle 2.10 zu sehen. Er entspricht 1 zu 1 dem Standard-*frame* von HTTP/3 mit der Konkretisierung Type 0x00. Folglich ist dieser *header* auch mindestens 2 Byte groß.

Type (i) = 0x00
Length (i)
Data (..)

Tabelle 2.10: Beispielhafter HTTP/3-Data-*frame* (nach RFC9114)

Wie bereits in HTTP/2 sind die *fields* für *request* und *response* (*header* und *trail*) für den Transport komprimiert. Dafür musste eine Änderung in HTTP/3 im Vergleich zu HTTP/2 durchgeführt werden. Da das HTTP/2 mit seinem *hpack* auf *in-order deliver* angewiesen ist, was QUIC nicht zur Verfügung stellt, musste hier eine Anpassung durchgeführt werden. Deshalb wurde *qpack* [102] eingeführt.

QPACK

qpack [102] dient zur Komprimierung von *header* und *trailer section* eines HTTP *requests*. Dabei verwendet *qpack* das Kernkonzept von *hpack*, akzeptiert aber auch *out-of-order-delivery*. Dabei wurde versucht, einen Kompromiss zwischen wenig HoL Blocking und optimaler Kompressionsrate zu entwickeln. Das Designziel lautete: Es sollte eine Kompressionsrate wie *hpack* mit weniger HoL Blocking erreicht werden.

2.2 Betriebssysteme

Die meiste Hardware benötigt ein Betriebssystem, um effizient zu arbeiten. Dabei kümmert sich das System darum, wie unterschiedliche Software asynchron auf einem System laufen oder Hardware-Komponenenten miteinander interagieren können, sowie weitere *low-level*-Interaktionen. Ein weit verbreitetes Betriebssystem ist Linux [108] – bzw. die jeweilige Distribution mit einen Softwarestack [104], da Linux selbst nur den Kernel darstellt. Im Weiteren wird Linux der Einfachheit halber als Synonym für das Gesamtsystem verwendet.

2.2.1 Linux

Ein moderner Kernel wie Linux stellt sogenannte Kernel-Funktionen bereit. Diese Kernel-Funktionen werden in einer isolierten bzw. sauberen Umgebung ausgeführt [111]. Das be-

deutet, dass ein Kontextwechsel durchgeführt wird, um so eine Funktionalität auszuführen. Dafür müssen Register und ähnliche temporäre Information vom aktuell laufenden Programm ausgelagert bzw. gesäubert werden, um dann die Kernel-Operation durchführen zu können. Das führt dazu, dass möglichst viele Operationen innerhalb eines dieser *spaces* ausgeführt werden sollte, da der Wechsel Leistung kostet. Viele der ersten Implementierungen der *microkernel*-Architekturen haben dieses Problem verdeutlicht [105].

Die Kosten dieser Operationen sind auch Central processing unit (CPU)- und Implementierungsabhängig. Erhöhte Kontextwechsel aber können zu einem Einfluss auf die Performance führen [105].

2.3 Verteilte Systeme

Bei Systemen ab einer bestimmten Größe oder mit einer bestimmten Anforderung ist es unumgänglich, mit verteilten Systemen zu arbeiten. Verteilte Systeme sollten nicht verwendet werden, wenn sie nicht notwendig sind, da die Komplexität signifikant steigt [221, kap. 1.2.5]. Dennoch gibt es einige Anwendungen, die versuchen, die Handhabung von verteilten Systemen zu vereinfachen. Eines davon ist das Orchestrations-Tool Kubernetes [192], das hier zusammen mit seinen Subsystemen vorgestellt werden soll.

2.3.1 Kubernetes

Kubernetes ist ein Orchestrationssystem, das ursprünglich von Borg abstammt [185, 223]. Es wurde von Google LLC [75] entwickelt und der CNCF [194] zur Verwaltung übergeben. Es ist ein Open-Source-Tool mit einer großen Gemeinde [188]. Es wird außerdem als Basis für sogenannte Distributionen wie Openshift [157] oder Rancher [176] verwendet. Das Modell hier ähnelt dem Distributionssystem, wie es bei Linux-Systemen üblich ist.

Kubernetes ist in erster Linie ein einfach erweiterbares Orchestrationssystem. Es besteht aus einer Application Programming Interface (API), die mit Konfigurationen befüllt wird. In der Regel sind diese Informationen im Format Yet Another Markup Language (YAML) oder JavaScript Object Notation (JSON) und beschreiben eine Funktionalität, die ausgeführt werden soll. Die API-Elemente werden Ressourcen genannt. Nun zu einem einfachen Beispiel: In Abbildung 2.10 ist eine Ressource mit dem Namen Deployment zu

sehen. Sie besteht – wie die meisten Ressourcen – aus einer *apiVersion*, *kind*, *metadata* und *spec*. Dabei sind die ersten zwei erwähnten Elemente zur Type-Definition und Versionserkennung notwendig. In den Metadaten werden die Information zur Wiedererkennung definiert. In *spec* finden sich die eigentlichen Information wieder. Im Beispiel dieses *deployments* haben wir einen Nginx *container*, der in drei *Pods* gestartet werden und immer drei lauffähige Instanzen bereit stellen soll. Ein *Pod* entspricht einem isolierten Prozesszweig – dazu später mehr. Das *deployment* stellt sicher, dass die Anzahl der *replicas* immer vorhanden sind, wenn es möglich ist. Um das zu ermöglichen wird ein *selector* erstellt, der die Elemente mit den Metadaten *app: nginx* überwacht und ein *template* hinzugefügt, das für die Neuerstellung der *Nginx Pods* benutzt wird.

Der Aufbau von Kubernetes ist ein klassischer *master-slave*-Aufbau oder *control-plane*- und *worker*-Aufbau. Im weiteren wird die zweite Bezeichnung benutzt, die erste soll nur erwähnt sein, weil dieser Name in der Literatur häufige Verwendung findet [12]. Dabei wird vom Kontrollsystem – hier *control*-Node genannten – System aus die Verwaltung der jeweiligen *worker* oder *worker*-Nodes durchgeführt. Möglich sind auch *multi-control*-Pläne-Instanzen, aber dieses soll hier nicht genauer betrachtet werden [197].

Es gibt zwei Arten von Ressourcen: *Namespace*-basierte und nicht-*namespace*-basierte. Das bedeutet, dass sie entweder nur in einem *namespace* gültig sind oder global für den *cluster* gelten. Die Isolation wird in Kubernetes meist auf *namespace*-Ebene durchgeführt. So sind Pods, Deployment und Service Definition innerhalb eines *namespaces* enthalten, wobei hingegen *cluster-roles* nicht zu einem *namespace* gehören.

Eine weitere wichtige Ressource ist die Service-Ressource, die in Abbildung 2.11 zu sehen ist. Sie erstellt eine Art Proxy, um mit unterschiedlichen Pods kommunizieren zu können. Dabei agiert sie wie ein *load balancer* und schickt die Anfragen zu einem der Pods weiter. Das funktioniert, indem in der *spec* Sektion ein *selector* der Pods vorhanden ist und die Ports, die *exposed* werden sollen, definiert sind.

Bei Ressourcen, die innerhalb von Kubernetes laufen, wird häufig zwischen *cuttles* oder *pets* unterschieden. Dabei werden *cuttle* als zustandslose Anwendung und *pets* als zustandbehaftete Anwendungen bezeichnet.

In der Abbildung 2.12 sind die zwei Haupteinheiten von Kubernetes zu sehen: Die *control plane* und die *worker*-Nodes. Innerhalb dieser Einheiten gibt es verschiedene An-

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Abbildung 2.10: Beispielhafte Kubernetes Deployment YAML

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Abbildung 2.11: Beispielhafte Kubernetes Service YAML

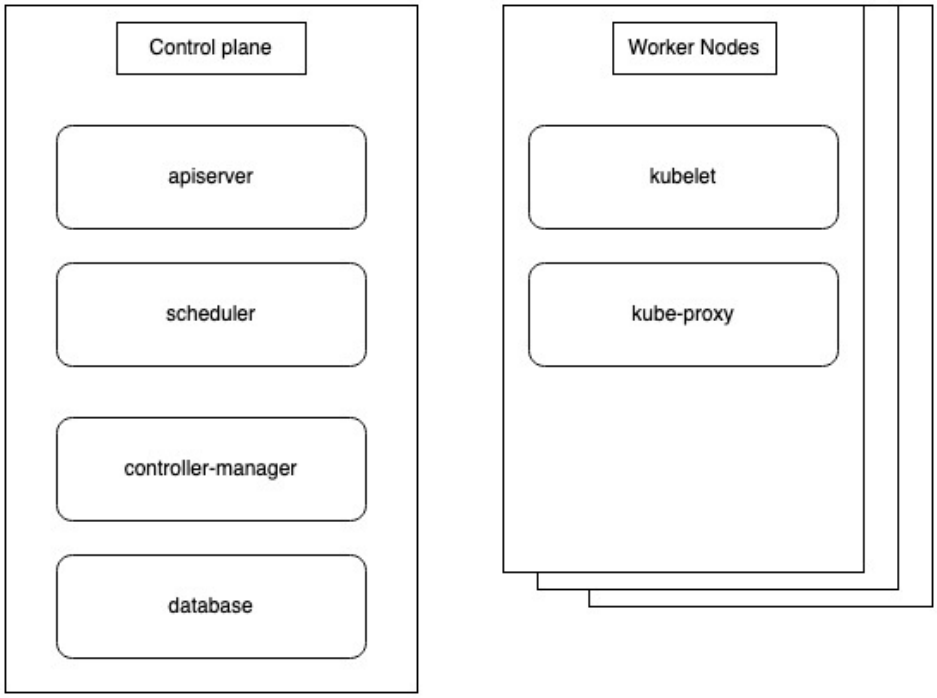

Abbildung 2.12: Kubernetes Komponenten

wendungen, die den Service bereitstellen. Der *control plane* zum Beispiel hat folgende Komponenten [203].

API-Server
: Der API-Server ist der Dienst, der die Kubernetes-API bereitstellt und damit die Kommunikation erlaubt.

Scheduler
: Die Komponente, die neu erstellten Pods Nodes zuweist, auf denen sie laufen. Dabei werden *scheduling*-Entscheidungen anhand von individuellen Regeln, Hardware- und Software-Anforderungen durchgeführt.

Controller-Manager
: Der Controller-Manager ist die *control plane* Komponenente, in dem die *controller*-Prozesse laufen. Jeder *controller* ist ein separater Prozess, aber um die Komplexizität zu reduzieren, wurden sie in einzelnen Anwendungen zusammengeführt und laufen als ein Prozess.

Um einige von diesen Controllern zu nennen:

- Node Controller: Er ist verantwortlich das Informieren und Reagieren, wenn ein Node Offline geht.
- Job Controller: Überwacht die *job resources*, die einen Task repräsentieren, und erstellt die Pods, um diese Tasks durchzuführen.
- ServiceAccount Controller: Erstellt *default serviceaccounts* für neue *namespaces*.

Database
: Meist ein etcd [58], kann aber bei anderen Distribution auch eine Mysql [134] Datenbank sein. Sie wird verwendet, um alle Daten/Zustände zu persistieren. Wichtig ist, dass diese Datenbank häufiger gesichert wird, da sie den kompletten Zustand des *clusters* beinhaltet.

Ein *worker*-Node hingegen besteht aus folgenden Komponenten:

Kubelet
: Der Service, der die Pods startet und sie am Leben hält.

Kube-proxy
: Ist ein Proxy, der auf jeden Node im *cluster* läuft und das Kubernetes-Service-Konzept implementiert.

Ein Kubernetes-System benötigt noch weitere Elemente, wie zum Beispiel einen DNS-Dienst [122] für die interne Kommunikation. Diese sind aber Strukturen, die selbst meist innerhalb des *clusters* verwaltet werden.

Eine Komponente, die genauer beleuchtet werden muss, ist der kube-proxy. Dieser stellt eine Abstraktion der Pods dar und agiert wie ein Proxy. Das wird mithilfe von Iptables [128] in der Standard-Konfiguration gelöst. Es gibt auch andere Lösungen wie IP Virtual Server (IPVS) [106] als Option, aber auch andere externe Lösungen [180].

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
```

Abbildung 2.13: Beispielhafte Kubernetes Secret YAML

In der Iptables-Variante wird eine IP benutzt, die für den Service zuständig ist, außerdem wird ein DNS-Eintrag für den Service erstellt. Wird nun eine Anfrage zum Service durchgeführt, wird die IP durch die Iptables-Regeln zu einer der zur Verfügung stehenden Pod-IP ausgetauscht. Dabei handelt es sich im Grunde um NAT-Regeln.

Eine weitere wichtige Ressource ist das *secret*, wie in der Abbildung 2.13 zu sehen ist. Die Werte, die hier unter *data* stehen, sind nur base64-encoded [95] und bieten damit nicht die Sicherheit, die der Name suggeriert. Ein *secret* ist eher ein Informationshalte-Objekt, das z.B. Zertifikate speichert, die anschließend als eine *environment variable* oder Datei im Dateisystem eingehängt werden kann.

Die in Kubernetes genannten Ressourcen können Speicher, Rechenleistung oder weitere Funktion wie *application layer routing* durchführen. Außerdem ist es Anwendungen möglich, diese API mit *custom resourcen* zu erweitern. Software, die dieses Verhalten nutzt, wird meist als Operator bezeichnet, da er die Operation von der jeweiligen Anwendung innerhalb des *clusters* durchführt.

2.3.2 Operator

Als Operatoren werden Software-Systeme innerhalb des Kubernetes-*clusters* genannt, welche Anwendungen operieren können [42]. Damit gemeint ist, dass diese Anwendung verwaltet werden. Um das zu verdeutlichen, nehmen wir als Beispiel den Dienst eines *webservers*. Ein Operator für diesen würde in der Regel eine sogenannte Custom Resource Definition (CRD) bereitstellen, mit deren Hilfe ein neuer *webserver* angelegt würde. Dabei kümmert sich der Operator darum, dass alles erstellt wird, was für dessen Betrieb benötigt wird. Im Falle eines einfachen *webservers* wäre es eine Deployment Ressource, ein *service* und gegebenfalls ein *ingress* [201]. Außerdem würde er die *configmap* [200] mit

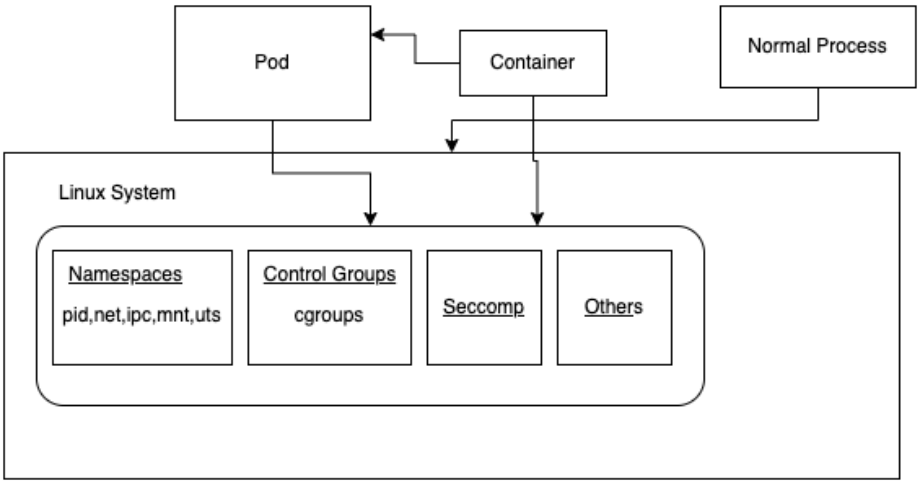

Abbildung 2.14: Beispiel Prozess Isolation

den Einstellungen einrichten und diese direkt ins Deployment zum *webserver* hinzufügen. Desweiteren würde er sicherstellen, dass diese Anwendung immer im konfigurierten Zustand verweilt.

2.3.3 Container

Um Kubernetes betreiben zu können, wird eine sogenannte Container Runtime Interface (CRI) kompatible Software benötigt. Alle Services innerhalb Kubernetes werden mit diesen *interface* gestartet und verwaltet. Dieses *interface* wird von einer Vielzahl von Lösungen implementiert, die auf unterschiedlichsten Wege versuchen, die zu laufende Anwendung zu isolieren. Dabei können die Level der Isolation sehr weit auseinander gehen, von Isolation über Linux-Kernel-Funktionalität bis hin zu Virtualisierung [125, 7, 133]. In der Regel wird ein System verwendet, das mithilfe von Linux-Kernel-Funktionalitäten eine Isolation herstellt. Meist ist es cri-o [24] in Kombination mit runc [132] oder docker [43]. Im folgenden werden wir uns dafür exemplarisch cri-o ein wenig genauer ansehen. In cri-o [24] werden mithilfe der *container engine* runc [132] Prozesse erstellt und isoliert.

In der Abbildung 2.14 sind unterschiedliche Quadrate zu sehen, einerseits ein *Normal Process*, welcher direkt auf das Linux-System und die globalen Komponenten *namespace*, *cgroups*, *seccomp* und *others* zugreift, und die *container* und Pods, welche darauf separat

zugreifen, dieses Verhalten ist hier mit Pfeilverbindungen gekennzeichnet. Diese separaten Elemente sollen im folgenden kurz erörtert werden:

Zuerst die *namespaces*. Diese isolieren komplette Räume von den Normalbereichen [107, 222]:

PID Der *container* erhält einen individuellen *process namespace* und kann andere Prozesse außerhalb des *namespaces* nicht sehen. Das führt auch dazu, dass der Prozess innerhalb eines *containers* die ID 1 hat, was normalerweise der *init process* besitzt. Dieser managed auch das Sterben von Prozessen, weshalb es zu dem Problem *pid 1 zombie reaping* kommen kann [103], aber auch das Handling von Kill-Signalen muss aufgrund des fehlenden *init processes* beachtet werden. Dieser *namespace* kann auch als *Process ID Namespace* bezeichnet werden.

net Der *container* erhält seinen eigenen Netzwerk-Stack. Wird auch Network genannt.

mnt Der *container* bekommt einen isolierten *mount table*. Wird mitunter auch *mount* genannt.

IPC IPC steht für *interprocess communication*. Die Möglichkeit, über das System-Level zu kommunizieren, kann durch IPC auf einen *namespace* reduziert werden.

uts Der *container* kann einen eigenen *hostname* und *domain name* besitzen. Wird auch als UNIX-Time-Sharing-*namespace* bezeichnet.

user Der User-*namespace* ermöglicht es, Userberechtigungen zu isolieren.

time Erlaubt es *containern*, unterschiedliche Zeiten zu benutzen.

cgroup Ermöglicht es, die Sicht auf den cgroups zu separieren.

Neben der Isolation innerhalb eines Systems durch *namespaces* gibt es noch die Isolation mithilfe von *control groups*. Cgroups erlaubt es, Prozesse, Prozesszweige, *resource usage* und Gerätezugriffe zu regulieren. Beispiele sind CPU oder Speicherzugriffe [85].

Außerdem gibt es noch seccomp [54], welches eine Art Sandboxing-Mechanismus im Linux-Kernel ist. Es erlaubt Beschränkungen von Systemaufrufen.

Außerdem wird *others* aufgeführt. Hiermit soll gezeigt werden, dass weitere Isolationsmöglichkeiten über seccomp hinaus möglich sind, wie z.B. SELinux [158] oder AppArmor

```
FROM node:18-alpine
WORKDIR /app
RUN yarn install --production
CMD ["node", "src/index.js"]
```

Abbildung 2.15: Beispielhaftes *dockerfile*

[9], möglich wären, die hier nicht aufgeführt wurden. Das würde aber zu weit führen, da hier nur ein kleiner Einblick gegeben werden soll.

Prozess-Isolation ist ein komplexes Thema und hier sollte nur ein Eindruck der Standard-Mechanismen vermittelt werden.

In Kubernetes werden die Isolationen Pods genannt. Diese werden von einem Kubelet gestartet. Das geschieht mithilfe des CRI. Innerhalb dieses Pods können mehrere *container* laufen, welche sich einen IPC-, NET- und PID-*namespace* teilen und die selbe cgroup verwenden [222].

Um einen *container* zu beschreiben, gibt es mehrere Möglichkeiten [219, 44]. Die am weitesten verbreitete Möglichkeit ist die unter Zuhilfenahme eines *dockerfiles* [44].

In der Abbildung 2.15 sehen wir ein Beispiel für ein *dockerfile*. Die *keywords* werden dort in Versalien geschrieben, wie z.B. das *FROM*, welches das Ursprungsimage definiert. *Keywords* wie *WORKDIR* und *RUN* sind selbsterklärend, mit *WORKDIR* wird in ein Verzeichnis gewechselt, mit *RUN* ein Befehl ausgeführt. *CMD* ist in diesem Beispiel der Befehl, der beim Ausführen des *containers* gestartet werden soll.

2.3.4 Container Registry

Eine *container registry* ist ein Service, der erstellte *container* speichert und verteilt. Dabei sehen die Adressen etwa so aus: *quay.io/goharbor/harbor-core:v2.2.1*. Sie bestehen aus einem *host*, z.B. *quay.io*, und einem Projekt *goharbor*, gefolgt von einem Namen mit einer Version, z.B. *harbor-core:v2.2.1*. Der erste Teil ist schlicht eine URL, der zweite Teil die Version. Auch hier gibt es eine Vielzahl an Implementierungen wie Harbor [189] oder Quay [155], die solch einen Dienst bereitstellen. Kubernetes benötigt so einen Dienst, um seine Pods zu starten, da diese *container*-Definition aus einer *registry* kommen muss.

2.3.5 Container Networks

Container Network Interface (CNI) [187] ermöglicht die Kommunikation unter den *containern* und ist damit ein signifikanter Bestandteil von Kubernetes. Es werden *overlay networks* oder BGP verwendet, aber es sind auch andere Optionen möglich, je nach Implementierung [181, 78].

Als *overlay networks* wird ein Netzwerk bezeichnet, das eine tieferliegende Netzwerkschicht noch einmal abbildet. Es stellt durch seine Abstraktion eine weitere Adressierungsebene bereit.

Das CNI stellt einen signifikanten Teil des *clusters* dar. Es wird zwar nicht mit Kubernetes mitgeliefert, wird aber benötigt, um ein lauffähiges System zu haben. Es gibt eine Menge Anbieter von *overlay networks*, um die Kommunikation der internen Strukturen von Kubernetes zu ermöglichen. Als Beispiel seien hier die größeren und bekannteren genannt: Calico [216] und Weave Net [227]. Wir werden uns im Folgenden genauer mit Projekt Calico auseinandersetzen.

Calico

Es wird sich Calico angeschaut, da es relativ lange auf dem Markt ist und in der Praxis recht viel verwendet wird, wenn man sich Kubernetes-*cluster*-Systeme von einigen größeren Anbietern wie IONOS SE [91] ansieht.

Calico bietet mehre Möglichkeiten zum Installieren an, wie die meisten Erweiterungen von Kubernetes z.B. durch einen HELM-Chart [191] oder mithilfe eines Operators. Desweiteren bietet es mehrere Einstellungsoptionen für den operativen Betrieb. In einer dieser Einstellungsmöglichkeiten wird ein *overlay* erstellt, in dem mithilfe von Virtual Extensible LAN (VXLAN) kommuniziert wird [213].

VXLAN ist eine im RFC 7348 [112] beschriebene Netzwerk-Virtualisierungsmethode, welche es ermöglicht *overlay*-Netzwerke einfach zu betreiben. Es fügt dabei weitere Elemente innerhalb der Nutzdaten hinzu und besitzt einen *header*, wie er in Tabelle 2.11 dargestellt ist.

0										1										2										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Flags (8)										VNI (24)																					
Reserved A (24)																												Reserved B (8)			

Tabelle 2.11: *header* eines VXLAN-Pakets (nach RFC7348)

Dieser *header* wird im *payload* eines UDP-Datagramms gesendet. Zur Funktionsweise von VXLAN: Es gibt sogenannte VXLAN Tunnel Endpoint (VTEP), welche virtuell oder physikalisch sein können, mit welchen kommuniziert wird. In unseren Beispiel wird Calico für jeden *server* ein virtuelles VTEP-*interface* mit den Namen "vxlan.calico" erstellen. Calico übernimmt die komplette Logik, um die VTEP-*devices* untereinander bekannt zu machen [147]. Über diese VTEP-*devices* wird dann das eigentliche *overlay*-Netzwerk ausgeführt.

Dabei werden die ersten 8 Bits des *headers* für *flags* verwendet, wobei alle Bits bis auf das erste reserviert sind. Dieses erste Bit muss immer gesetzt sein, um eine erlaubte VXLAN Netzwerk ID zu sein. Darauf folgen 24 Bits, welche verschiedene VXLANs verwenden können. Nur VXLANs mit der gleichen ID können untereinander kommunizieren. In dem CNI Calico wird hier einfach eine Standard- oder konfigurierte ID [212] gesetzt.

2.4 Verwandte Arbeiten

Die Forschung in den oben angeschnittenen Bereichen ist sehr aktiv. Dabei muss aber stark sondiert werden in jeweiligen Teilbereichen. Angefangen bei QUIC, dieses besitzt mehrere unterschiedliche Versionen. Einmal die ursprünglich von Google erstellte Version [14, 20, 101] und desweiteren die Versionen, die von einer Vielzahl von *draft*-Spezifikation erstellt wurden [142]. Dasselbe gilt natürlich auch für HTTP/3, welches auch in vielen Vorab-Versionen evaluiert worden ist [113, 99]. Daraus resultierend gibt es eine Vielzahl an Veröffentlichungen im Bereich QUIC, welche für diese Untersuchung nicht mehr die höchste Relevanz haben. Erst die Veröffentlichung ab Mai 2021 oder kurz davor sind für diese Untersuchung sehr interessant, da dort die Standardisierung des Protokolls festgelegt worden ist [92]. Ähnliches gilt auch für HTTP/3, welches erst 2022 standardisiert wurde. In den Veröffentlichungen wird QUIC im Titel von Evaluation häufig mit HTTP/3 gleichgesetzt, weshalb im Folgenden auch hier diesem Muster gefolgt wird. Wenn von QUIC gesprochen wird, kann es sich um Arbeiten handeln, die das Protokoll direkt

getestet oder über HTTP/3 dieses Protokoll evaluiert haben.

Aber auch ab diesen Zeitpunkt gibt es eine Menge Untersuchungen von QUIC in unterschiedlichsten Bereichen. Ein häufig anzutreffender Vergleich ist z.B. der zwischen TCP und QUIC, z.B. *Dissecting Performance of Production QUIC* von Yu und Benson [232]. Meist sind die Ergebnisse dieser Vergleiche sehr positiv für QUIC, auch in diesem vorgestellten Falle wird festgestellt, das QUIC – bzw. in dieser Untersuchung HTTP/3 – aufgrund des reduzierten *handshakes* besser mit kleinen Nutzdatenmengen performt, und dass bei größeren Nutzdaten die gleiche Performance wie bei HTTP/2 auftritt. Dabei bezieht sich diese Aussage auf Messung ohne hinzugefügte *loss rate* und ohne extra Latenz. Ein weiterer wichtiger Aspekt, der in dieser Arbeit erwähnt wird, ist, dass die Messwerte, die es bislang gab, unterschiedliche Ergebnisse erbrachten, so dass QUIC TCP in einigen Veröffentlichungen outperformt und in anderen wiederum nicht [21, 30, 96, 126, 233]. Das wird auf die unterschiedliche Konfiguration zurückgeführt. Diese Begründung ist auch in anderen Veröffentlichungen als ein Aspekt wiederzufinden [57, 123]. Ein wichtiger Punkt, der hier ebenfalls erwähnt werden sollte, ist, dass viele von diesen Arbeiten weit vor der Finalisierung von QUIC durchgeführt wurden und einige auch mit eine gQUIC Variante, was den Vergleich schwierig macht. Dieser Test wurde hier auf einem Produktionssystem durchgeführt, was natürlich viele netzwerkbasierte Problematiken wie Switching, Routing und Middleboxes mit in die Messungen einfließen lässt.

Ein weiterer wichtiger Aspekt, der dort erwähnt wird, ist, dass die Implementierung der Spezifikation einen signifikanten Einfluss auf die Performance hat. In der obengenannten Veröffentlichung von Yu und Benson wurde der *congestion-control*-Algorithmus als Beispiel genommen: Obwohl der gleiche Algorithmus implementiert wurde, sind durch die unterschiedliche Implementierung auch unterschiedliche Performance-Ergebnisse entstanden.

Andere Ergebnisse der Untersuchungen in diesem Feld sind auch gerne beim *handshake*, da dieser durch die direkte Integration von TLS 1.3 Vorteile gegenüber TCP bietet, selbst mit TFO, was vor allem bei kurzen Kommunikationen zum Tragen kommt [232, 167]. Außerdem wird gerne über die Sicherheit dieses verkürzten *handshakes* und weitere Faktoren diskutiert [94, 92]. Interessant sind auch die Untersuchungen, die in den Bereich *kernel space* gehen, diese zeigen signifikante Performanceverbesserungen in allen Bereichen, was auch durch die eben erwähnte Veröffentlichung untermalt wird [225, 220]. Diese Verbesserungen lassen darauf schließen, dass der *user-space*- und *kernel-space*-

Kontextwechsel einen Einfluss auf die Performance haben könnte. Desweiteren werden übliche Transportprotokoll-Untersuchungen durchgeführt, wie Wechseln der *congestion control* oder Parameterstudien [123, 226].

Zusammenfassend haben wir in dem Bereich von QUIC und HTTP/3 ein sehr heterogenes Feld an Messergebnissen, was durch die unterschiedlichen Implementierungen des Standards und deren Komponenten mit unterschiedlichen Parametern entsteht.

Wenn sich Kubernetes angeschaut wird, sollte sich vor allem auf den Bereich CNI konzentriert werden. In diesem Bereich gibt es wiederum viele Untersuchungen [137, 149, 234, 130]. In der Veröffentlichung *Overview of kubernetes cni plugins performance* von Kapočius [98] werden 9 populare CNI Plugins evaluiert. Die Umgebungen sind dabei virtuelle und physikalische Datencenter, welche jeweils 10km voneinander entfernt sind und mit einer separaten Glasfaserleitung verbunden sind. Das Ergebnis dieser Evaluation ist, dass eine *baremetal*, also eine CNI unabhängige Verbindung einen messbaren Unterschied zu einen CNI im Datendurchsatz hat. Die einzelnen CNI haben dabei größtenteils untereinander keine großen Unterschiede. In dieser Ausarbeitung wurden nur TCP und auf TCP aufbauende Protokolle als Datenverkehr gemessen. Da in dieser Untersuchung QUIC bzw HTTP/3 evaluiert werden, sind diese Messergebnisse nur als Orientierung hilfreich, um Unterschiede zwischen verschiedenen CNI zu analysieren.

Ein weiterer spannender Aspekt ist dabei, dass es auch einige Untersuchungen gibt, die unterschiedliche Ergebnisse haben, wie z.B. *Measurement and evaluation for docker container networking* von Zeng u. a. [234] und *Benchmark results of Kubernetes Network Plugins (CNI) over 10gbit/S network* von Ducastel [50, 49]. Sie beschreiben beide ein unterschiedliches Verhalten innerhalb von UDP Datenverkehr in einem CNI. Aus einer weiteren Veröffentlichung von Novianti und Basuki geht hervor, dass ein *overlay*-Netzwerk nur einen kleinen Geschwindigkeitsnachteil bringt [130]. Dabei ist diese Aussage mit Vorsicht zu genießen, da keine Informationen zu der konkreten Konfiguration in der Veröffentlichung enthalten war. Ein wichtiger Aspekt oben genannter Untersuchung von Novianti und Basuki ist, dass der UDP Datenverkehr nur einen Bruchteil der *interface*-Bandbreite nutzen konnte und eine hohe CPU Auslastung im Gegensatz zu dem TCP Datenverkehr erzeugte.

Die hier vorgestellte Forschungsarbeit soll viele dieser Bereiche verbinden, indem explizit der Bereich der Kommunikation von Diensten innerhalb eines Kubernetes-*clusters*

untersucht werden soll. Dabei soll das Gesamtzusammenspiel von Kubernetes und verschiedenen HTTP/3 Implementierungen evaluiert werden. Dieses soll dann zeigen, ob HTTP/3 in diesem Kontext gleichwertig mit den Anforderung der bisherigen HTTP-Standards ist oder sogar Vorteile bietet. Außerdem ist aufgrund des Alters der letzten Veröffentlichung eine Revisite der Implementierung angemessen: Einerseits wurde QUIC Version 2 in einigen Implementierungen hinzugefügt, andererseits sind durchschnittliche Releasezyklen zwischen 3-6 Monaten [198, 182, 1] realistisch. Dadurch haben Veränderungen stattgefunden, welche es sinnvoll machen können, sich die Implementierungen noch einmal genauer anzuschauen. Dabei zu beachten ist, dass hier explizit eine Komposition evaluiert werden soll, welche eine realistische *cloud environment* darstellt und damit keine isolierte Betrachtung der jeweiligen Komponenten durchgeführt werden soll. Damit verbunden ist, dass es hier keine spezielle Kubernetes-Untersuchung eines CNI oder von kube-proxy optionen geben wird. Es ist auch keine isolierte Untersuchung von speziellen QUIC oder HTTP/3 Funktionalitäten. Als letztes sei erwähnt, dass hier öffentliche und damit für eine Vielzahl an Entwicklern zugängliche Versionen von Protokollimplementierungen evaluiert werden sollen.

3 Evaluationsvorbereitungen

Im folgenden Abschnitt soll zuerst die Entscheidung zu den jeweiligen Komponenten der Evaluation diskutiert werden. Dann wird die Kommunikation innerhalb des Netzwerks noch einmal genauer beleuchtet. Danach werden die vorgestellten Protokolle zusammengeführt und der theoretische Protokoll-*overhead* definiert. Als nächstes wird die Versuchsplattform vorgestellt und erörtert. Dann werden die verwendeten Implementierungen der Webkommunikationsprotokolle vorgestellt.

3.1 Umfeldanalyse

Zuerst muss eine Entscheidung über die jeweiligen verwendeten Elemente getroffen werden, dafür werden wir vom Großen ins Kleine gehen.

In heutigen *cloud*-Systemen wird meist die Hardware durch Virtualisierung getrennt und nach Einsatzzeit verkauft [86]. Deshalb soll auch in dieser Evaluation eine Virtualisierung verwendet werden. Es wird als Virtualisierungslösung Virtualbox [135] eingesetzt. Es wurde sich für diese Virtualisierungslösung entschieden, weil es trivial ist, diese zu verwenden und keine größeren Anpassungen am System vorgenommen werden muss. Außerdem ist es sehr verbreitet, weshalb eine gute Dokumentation vorhanden ist.

Um eine Evaluation von Webkommunikationsprotokollen durchführen zu können, muss zuerst eine Plattform geschaffen werden, die dem aktuellen Standard entspricht. Dafür gibt es die Option, die Software direkt auf einem Linux- oder Windows-System zu betreiben. In diesem Aufbau wurde sich dagegen entschieden, da die horizontale Skalierung auf einem Direktsystem aufwändig ist. Das *scheduling* muss selbst gemanagt werden, ebenso die Verwaltung der zur Verfügung stehenden Ressourcen. Das gleiche gilt für die Kommunikation der einzelnen Systeme untereinander, was eine extreme Komplexität mit sich bringen würde. Eine Alternative dazu ist es, ein Orchestrationstool zu verwenden, wie z.B.

Nomad [84], Mesos/Marathon [115] oder Kubernetes [192]. Vergleicht man die aktuellen Aktivitäten in Github [71] von *contributer* und *issuer*, zeigen Nomad und Mesos/Marathon im Vergleich zu Kubernetes eine signifikant geringere Aktivität auf [179, 190, 83]. Der Funktionsumfang der Lösungen ähnelt sich sehr, dadurch bieten sich für diese Evaluation alle an. Aufgrund der aktuellen Github-Aktivität und der Verwaltung durch die CNCF [194] wurde sich in dieser Evaluaton für Kubernetes als Orchestrationssystem entschieden.

Kubernetes ist aber nicht gleich Kubernetes. Es wird auch gerne als *cloud os* bezeichnet [29]. Ähnlich wie bei Betriebssystemen gibt es auch hier eine Reihe von Anbietern, die Kubernetes-Distribution anbieten – als Beispiel seien hier Konvoy [33], Openshift [157] und Rancher [176] erwähnt. Jedes System hat hierbei seine eigene Komposition von z.B. *container engine*, *container network*, Datenbanken und mehr. Aufgrund der extremen Vielfalt und der Grundannahme, eine Evaluation für Webkommunikationsoprotokolle durchzuführen, was keine erhöhten Sicherheitsanspruche benötigt, wurde sich für ein Vanilla Kubernetes entschieden, welches mithilfe von kubeadm [196] installiert wird, um die komplexe Initialisierung aller Zertifikate zu umgehen [100].

Die Plattform Kubernetes benötigt aber immer noch ein Betriebsystem, auf dem diese API bereitgestellt wird. Es werden aktuell zwei Systeme offiziell unterstützt, Windows Server [118] und viele Linux-Distributionen [202, 205]. Da Windows auf dem *server*-Markt keine große Rolle spielt und viele Softwarelösungen im *server*-Bereich ursprünglich für Linuxsysteme bzw. Unix-artige Systeme entwickelt wurden, fiel die Wahl auf eine Linux-Distribution [67]. Dort gibt es wiederum eine große Auswahl, meist wird dort zwischen den jeweiligen Package-Systemen entschieden. Am populärsten sind dabei RPM von Red Hat, Inc [156] und DEB von Debian [35, 10]. Aufgrund der besseren Erfahrung mit Debian-basierten Paketen wurde sich für Debian 11 als System entschieden.

Um Kubernetes zu betreiben, benötigt es noch zwei weitere Komponenten: Das *container engine* und das *container network*.

Für das CRI, wie die Schnittstelle in Kubernetes für das *container engine* genannt wird, wurde sich für eine Kombination aus runc und cri-o entschieden. Diese beiden Komponenten sind Open Source und Teil der CNCF [228, 186]. Außerdem besitzen sie eine gute Dokumentation und die Isolation basiert wie bei Docker auf den oben vorgestellten *namespace*-Mechanismen, weshalb die Wahl auf dieses Setup fiel.

Das CNI von Kubernetes benötigt eine konkrete Implementierung. In den Performancetests von Novianti und Basuki [130] ist zu erkennen, dass die Performance zwischen den unterschiedlichen Anbietern von *overlay*-Netzwerken nicht signifikant ist. In den hier geplanten Versuchsaufbauten soll ein *encapsuled* Netzwerkverkehr, sprich ein *overlay*-Netzwerk, evaluiert werden. Der Grund dafür ist, dass es häufig als Standard-Installation vorhanden ist und keine weiteren Anforderungen an die Netzwerk-Infrastruktur stellt [215]. Dafür gibt es eine gute Auswahl an Anbietern von CNI-Software. Flannel ist aufgrund seiner Einfachheit sehr interessant, bietet aber nicht an, erweiterte Netzwerk-Richtlinien durchzuführen [63]. Calico ermöglicht es, zwischen unterschiedlichsten Modi für das *container network* zu wechseln. Diese Flexibilität und die sehr umfangreiche Dokumentation [214] hat zu der Entscheidung geführt, dieses System zu verwenden.

Um *container* innerhalb von Kubernetes benutzen zu können, benötigt es eine *container registry*, wo diese erreichbar sind. Hier gibt es ein großes Spektrum an Fähigkeiten und Anbietern. Im Folgenden wurde sich einfach für Harbor [189] entschieden, weil es ein CNCF-Projekt ist [186] und die Anforderung einer einfachen Installation und Verwendung erfüllt. Außerdem spielt es in der Evaluation selbst keine signifikante Rolle, da die Images auf den *worker nodes* gecached werden.

Es gibt eine Vielzahl an Webkommunikationsprotokollen. In dieser Arbeit beschränken wir uns auf die im Web am meisten verwendete Protokoll-Familie HTTP. Von dieser Familie existieren eine Vielzahl an Implementierungen [146, 59, 153]. Durch diese Popularität wird sich in dieser Untersuchung ausschließlich auf diese Protokoll-Familie reduziert, da der Umfang bei weiteren Protokollen zu groß wird. Außerdem ist die Vergleichbarkeit schwierig, wenn schon die Semantik zwischen den Protokollen unterschiedlich ist. Aus diesen Gründen werden andere Protokolle in dieser Untersuchung nicht beachtet.

Für die Anwendungsevaluation wurden bereits veröffentliche und bei der IETF QUIC gelistete Implementierung sondiert [154]. Dabei sind alle Implementierungen für Forschungszwecke und Versionen ohne HTTP/3 herausgefallen. Die anderen Implementierungen wurden betrachtet und ein Auswahl anhand der zur Entwicklung benutzten Programmiersprachen für die Evaluierung vorbereitet.

Als Auswahl für die HTTP/1.1- und HTTP/2-Varianten wurden Apache2 [178] und Nginx [59] ausgewählt, das sind aktuell die meist verbreiteten *webserver* und damit sinn-

volle Kandidaten, um sie mit den neuen HTTP/3 Implementierungen zu vergleichen [127].

Für die jeweiligen Implementierungen wurde jeweils ein *dockerfile* erstellt, in dem die Anwendung mit ihren Subsystem und System-Anforderungen beschrieben wurde [44]. Diese Art der Beschreibung für *container engine* ist weit verbreitet, wie anhand der vielen Integrationen in den Repositories zu sehen ist [27, 152]. Aus diesem Grund wurde sich auch auch für diesen Weg entschieden, anstelle alternative Ansätze zu verwenden [219].

Als nächstes musste die Kommunikation zwischen den Diensten definiert werden. Dafür gibt es einige Optionen, wie z.B. REST [114], JSON:API [231], Simple Object Access Protocol (SOAP) [12, Kap. 3.4.4.1] oder Remote Procedure Call (RPC) [12, Kap. 3.2]. Hier wurde sich für ein simples JSON entschieden, was in verschiedenen Größen verschiedene Ressourcen darstellen soll. Es wird keine Dokumentation für die Struktur benötigt, wie bei SOAP oder keine Referenzierungen wie bei JSON:API. Auch RPC kommt nicht in Frage, da es eine weiteren Ebene in die Anwendung ziehen würde und damit weniger die eigentliche HTTP-Implementierung testen würde, sondern auch die jeweilige RPC-Implementierung.

Es wird außerdem auf Komprimierung der Daten verzichtet, da dieses hinsichtlich CPU-Zeit und Größe das Endergebnis beeinflussen könnte.

Zusammenfassend wird eine Virtualisierung der Maschinen über Virtualbox durchgeführt, auf denen Debian 11 läuft, mit einen Kubernetes-*cluster*, der mit kubeadm aufgesetzt wird. Als Komponenten für Kubernetes wird das CNI Calico eingesetzt, das über VXLAN ein *overlay*-Netzwerk für die interne Kubernetes-Kommunikation benutzt. Als CRI wird cri-o mit runc verwendet, um *container* auszuführen. Außerdem wird ein *Harbor server* als *container registry* erstellt, von welcher Kubernetes dann die in *dockerfiles* beschriebenen *container* beziehen kann. Die *container* beinhalten die jeweilige HTTP-Implementierung, die als Webkommunikationsprotokoll evaluiert werden soll. Diese werden von *server*-Seite mit einem JSON bestückt, um eine Ressource zu simulieren.

Dieser Sachverhalt ähnelt sehr dem Setup in modernen *cloud-environments* [86, 91]. Damit ist das Setup ideal geeignet, um zu beobachten, wie sich die unterschiedlichen HTTP-Implementierungen verhalten.

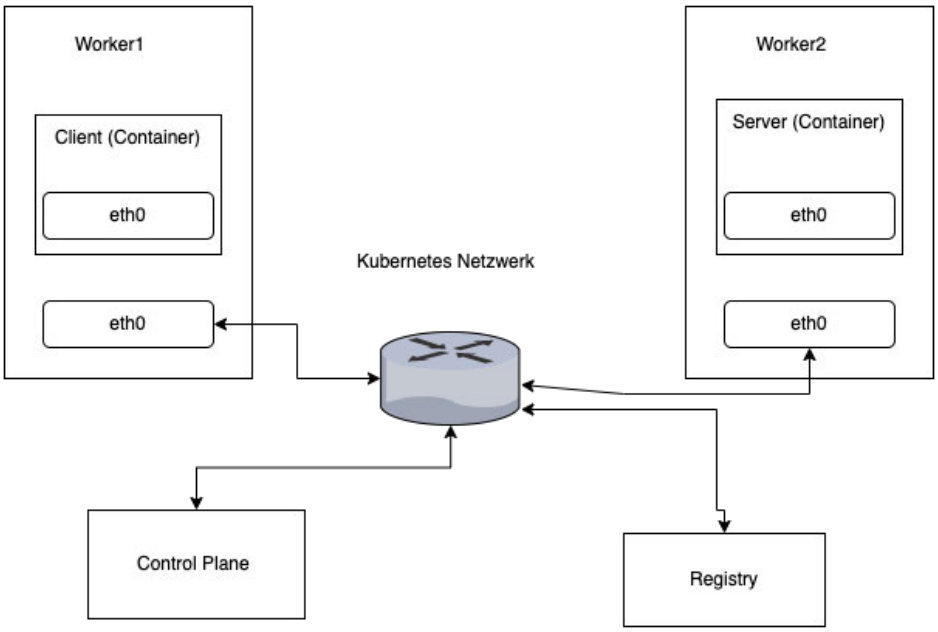

Abbildung 3.1: Aufbau Kubernetes-Cluster

3.2 Aufbau Kubernetes

Im Folgenden wird die Evaluationsumgebung definiert. Als Grundlage wird ein mit Virtualbox virtualisiertes Netzwerk verwendet. Innerhalb dieses Netzwerks befinden sich ein Kubernetes-*control-plane* und zwei Kubernetes-*worker-nodes*, außerdem ein *registry server*. Die Kommunikation zwischen den jeweiligen Anwendungen im *cluster* wird über ein *overlay*-Netzwerk mit Calico über VXLAN durchgeführt. Dieser Sachverhalt ist in Abbildung 3.1 noch einmal verdeutlicht.

In der Abbildung sind Quadrate zu sehen, die jeweils separate virtuelle Maschinen symbolisieren. Diese sind gekennzeichnet mit *worker1*, *worker2*, *control plane* und *registry*. Die abgerundeten Vierecke innerhalb der *worker* stellen die Netzwerk-Schnittstellen dar. Die Pfeile stellen die Netzwerkverbindungen zwischen den virtuellen Maschinen dar. Bei *control plane* und *registry* wurde darauf verzichtet, da dort nur eine relevante Netzwerk-Schnittstelle nach außen vorhanden ist. Innerhalb der *worker1*- und *worker2*-Maschinen ist ein weiteres Quadrat, gekennzeichnet mit *client* und *server*. Diese Elemente stellen den jeweiligen HTTP-*client* und -*server* dar, welcher innerhalb von einem *container* isoliert ausgeführt wird. Das ist eine abstrakte Darstellung, die die Struktur vereinfacht darstellen soll. Innerhalb der *container* werden auch weitere Services ausgeführt, genau

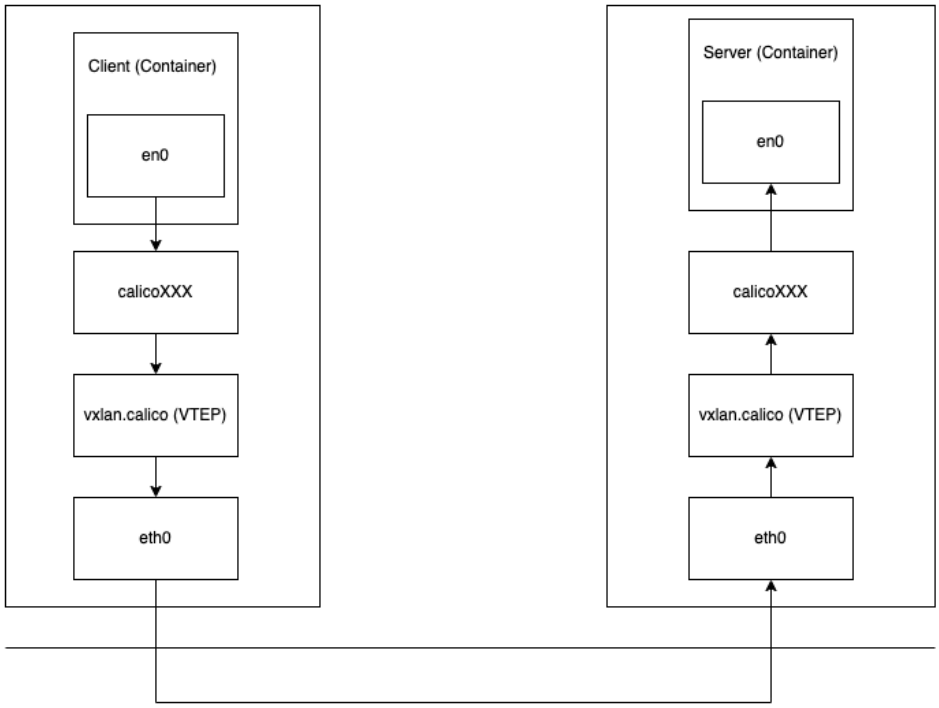

Abbildung 3.2: Netzwerkaufbau

wie auf den *worker nodes*, außerdem sind auch weitere Netzwerk-Schnittstellen bei einer Kommunikation von *client* zu *server* beteiligt – diese Aspekte wurden der Einfachheit halber nicht mit dargestellt.

Die Entscheidung für zwei *worker*-Nodes entstand, um evaluieren zu können, ob eine gewisse Trennung auf der Virtualisierungsebene Einfluss auf die jeweiligen Protokolle hat.

Bei Betrachtung dieser Umgebung darf die Konstruktion der einzelnen Anwendung nicht außer acht gelassen werden. Durch die Isolation, die Kubernetes mit Hilfe des CRI, also cri-o und dem CNI – in diesem Falle Calico – durchgeführt wird, entstehen viele neue virtuelle Netzwerkgeräte, was die Kommunikation zwischen den *client* und *server* deutlich beeinflusst. Hier folgt ein Beispiel für eine Kommunikation in dem hier entworfenen Szenario.

In der Abbildung 3.2 ist zu erkennen, dass zwei große Vierecke mit einem Pfeil verbunden sind, dabei sollen die Vierecke die zwei virtuellen Maschinen darstellen. Im linken Viereck, wo ein Viereck mit der Aufschrift *client* enthalten ist, sehen wir unterschiedli-

che Vierecke, die miteinander verbunden sind. Dabei stellt jedes Verbund-Mitglied eine Netzwerkschnittstelle dar. Das bedeutet: Um eine Nachricht an das rechte Viereck mit der Aufschrift *server container* zu senden, durchläuft die Nachricht vier Netzwerkschnittstellen.

Das Interessante daran ist, dass der *server* hinter einer NAT-IP-Adresse versteckt ist. In Kubernetes werden die Pods meist über einen Service adressiert. Hierfür wird eine virtuelle IP erstellt, die der Service erhält. Anschließend werden in unseren Fall über den kube-proxy-Dienst einige Iptables-Einträge hinzugefügt, so dass der *server* nach dieser IP aufgelöst werden kann. Das bedeutet effektiv, dass innerhalb des *containers* nicht bekannt ist, mit welchem *server* kommuniziert wird. Das hat auch auf die Messergebnisse im tcpdumps innerhalb eines *containers* Einfluss.

3.3 Aufbau der Pakete

In den Grundlagen wurden die signifikant involvierten Protokolle vorgestellt. Diese Protokolle sollen hier nun in der Reihenfolge zusammengesetzt werden, wie sie im System auch auftauchen.

Generell ist hier zu differenzieren zwischen dem Protokoll-*header* und den eigentlichen Nutzdaten. Der *overhead* variiert zwischen den zu evaluierenden Anwendungen immer ein wenig, es gibt aber einen gemeinsamen Nenner. In der Abbildung 3.3 sieht man zwei Quadrate mit Protokollen und deren Größe in Bytes. Dabei wird unterschieden zwischen *overlay overhead*, welcher 50 Bytes groß ist, und aus den Elementen Ethernet II mit 14 Bytes, IPv4 mit 20 Bytes, UDP mit 8 Bytes und VXLAN ebenfalls mit 8 Bytes besteht, und dem *default overhead*, welcher 34 Bytes entspricht und aus den Elementen Ethernet II mit 14 Bytes und IPv4 mit 20 Bytes besteht. Das erstgenannte ist die extra *header*-Last, die durch das *overlay*-Netzwerk entsteht, und das *default overhead* ist benötigt, um eine generelle Adressierung möglich zu machen. Damit werden generell schon mal 84 Bytes benötigt, ohne dass Transportprotokoll-*overhead* und das Anwendungsprotokoll-*overhead* einbezogen worden sind.

Aufbauend auf der Basis des *header overhead* kommt jetzt noch das eigentliche Protokoll-*overhead*. Dabei muss zwischen den einzelnen HTTP-Versionen unterschieden werden, wie in Abbildung 3.4 zu sehen. Die Darstellung besteht aus unterschiedlichen Vierecken,

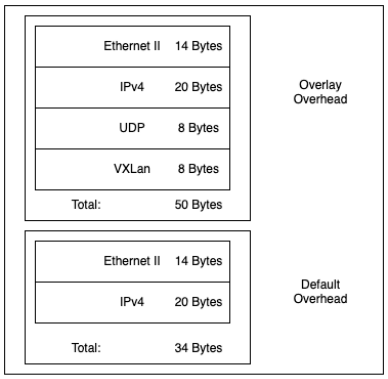

Abbildung 3.3: Vergleich der Pakete

wobei die äußeren mit einer Überschrift gekennzeichnet sind, die die jeweiligen Protokolle definieren. In diesen Vierecken sind weitere Vierecke, welche die jeweiligen *overhead* Elemente mit Namen und Größe definieren. Unter diesen Vierecken ist noch einmal ein "*Total*" für das jeweilige Anwendungsprotokoll definiert.

Bei der Version 1 von HTTP misst der TCP-*overhead* mindestens 20 Byte.

In Version 2 steigt durch den Einbau weiterer *stream*-Logik die Komplexität schon ein wenig im Vergleich zur ersten Version. Es läuft weiterhin über TCP, also bleibt dort der *overhead* von mindestens 20 Byte bestehen, aber es wird eine weitere *overhead*-Schicht hinzugefügt – Erstens für das TLS von 6 Bytes, zweitens für die *stream*-Logik von 9 Bytes. Das TLS wird hier explizit erwähnt, da jede *stream*-Einheit per TLS gesichert wird. Bei HTTP/1.1 wird der gesamte *stream* in ein TLS-Paket gepackt, weshalb der *overhead* nicht kontinuierlich gesendet wird.

Version 3 von HTTP benutzt die QUIC-Mechanismen. Das führt dazu, dass wir eine *header-overhead*-Basis von weiteren 8 Byte für UDP haben, außerdem einen *overhead* von 7 Bytes für den QUIC-*header* und durch die *stream*-Abstraktion noch einmal 6 Bytes. Da viele Mechanismen von QUIC über *frames* innerhalb des Protokolls gelöst werden, nicht wie bei TCP über den *header*, ist der Vergleich ein wenig schwierig.

Es sollten nach diesem Modell theoretisch über HTTP/3 mehr Nutzdaten transferiert werden können.

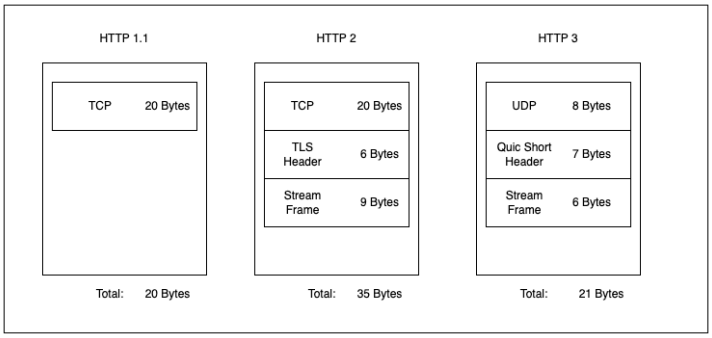

Abbildung 3.4: Vergleich der HTTP Pakete

Zu beachten ist immer das unterschiedliche Protokoll-Design, welches bei TCP eher *header*-orientiert ist, bei QUIC hingegen eher *frame*-orientiert. Das bedeutet, dass TCP bei jeder Nachricht mehr Kontrollinformationen im *header* mitversendet, wobei QUIC hingegen situativ seiner Nachricht einen weiteren *frame* hinzufügt. Das kann dazu führen, dass sich, bedingt durch die unterschiedlichen Vorgehen, die Anzahl an benötigten Nachrichten ausgleichen könnten.

3.4 Aufbau Testsetup

Um die Evaluation durchführen zu können, wurde eine Umgebung aufgesetzt, die es ermöglicht, automatisch Versuchsdurchläufe zu starten. Dafür wurde zuerst ein Entwurf entwickelt und anschließend eine Implementierung erstellt, welche anschließend verifiziert wurde.

3.4.1 Anforderung

Zuerst wurde definiert, welche Werte gesammelt werden sollen und können. Dafür wurden sich relevante Werte für ein Webkommunikationsprotokoll überlegt, die messbar sind.

Erstens ist die Dauer der gesamten Kommunikation über das jeweilige HTTP-Protokoll ein zu messender Faktor. Das ist interessant, da die Dauer auf Anwendungsebene gemessen wird und damit abhängig von der entwickelten Sprache oder der Laufzeitumgebung ist. Da QUIC über UDP läuft und damit im *user space* implementiert ist, könnte dieser Wert in HTTP/3 einen Einfluss haben.

Zweitens soll der Netzverkehr innerhalb des Pods gemessen werden, um die Welt aus dem Standpunkt des Senders zu sehen. Durch die virtuelle Netzwerkstruktur ist die Ansicht von dort ein wenig anders als vom *interface* aus, welches die eigentliche Netzwerk-Kommunikation durchführt. An dieser Stelle sind noch keine Einflüsse von Iptables oder des *overlay*-Netzwerks zu sehen.

Die wirkliche Netzwerk-Kommunikation über das Haupt-Netzwerk-*interface* muss beobachtet werden, um Aussagen über den realen Netzwerkverkehr treffen zu können.

Außerdem sollte die Benutzung von *system_ calls* beleuchtet werden. Da QUIC im *user space* implementiert ist, ist es interessant, wie die Netzwerk-Kommunikation dort im Gegensatz zu den TCP basierten Implementierung unterschiedlich realisiert ist.

Damit sind die zu monitorenden Werte definiert. Nun muss eine Werkzeuganalyse durchgeführt werden, um herauszufinden, wie diese Werte gemessen werden können.

Um Netzwerkverkehr zu messen, ist tcpdump [207] das Standardwerkzeug und sollte hier auch benutzt werden, um die zwei Kommunikationskanäle mitzuschneiden. Das Messen der *system_ calls* soll über die Softwarelösung strace [175] durchgeführt werden. Das ist sehr weit verbreitet und ausgereift und damit ideal für diesen Versuch geeignet. Um die Zeit eines *request* innerhalb einer Anwendung zu messen, wird die *client*-Anwendung so manipuliert, dass sie die Zeit in eine Hashmap oder eine ähnliche Struktur in Millisekunden schreibt und am Ende des Prozesses in eine Datei speichert. Dieses Vorgehen bringt den Vorteil, dass alles sehr genau gemessen werden kann, da das Vorgehen anwendungsspezifisch entwickelt wurde. Das setzt voraus, dass die *client*-Seite manuell erstellt wurde.

3.4.2 Implementierung/Entwurf

Um diesen Prozess, der in den Anforderungen definiert ist, möglichst effizient und reproduzierbar zu machen, muss dieser Prozess automatisiert werden.

Dafür wurde ein Entwurf in Abbildung 3.5 entwickelt. Dort zu sehen sind die zwei zu testenden Elemente, dargestellt als Quadrate mit der Aufschrift *client* und *server*. Da-

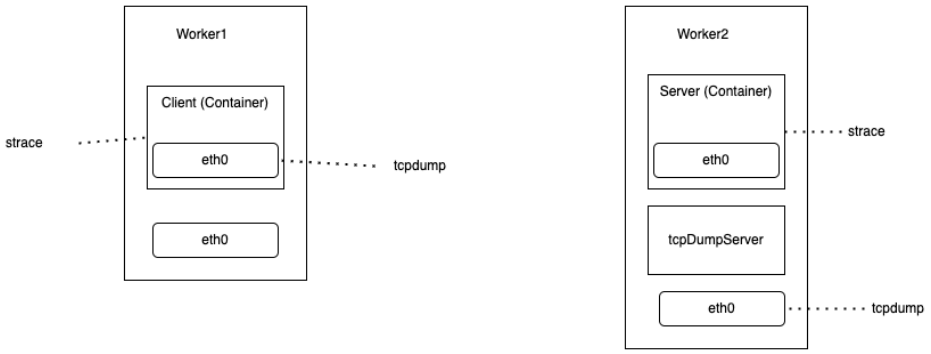

Abbildung 3.5: Aufbau des Testsetups

hinter sind Vierecke zu sehen mit der Aufschrift *worker1* und *worker2*, was die jeweiligen Virtuelle Maschine (VM)s darstellen sollen. Außerdem sind die Netzwerk-Schnittstellen als abgerundete Elemente zu sehen. Desweiteren gibt es die gestrichelten Linien, welche die Messpunkte definieren. Einerseits die Netzwerk-Schnittstellen, die mit tcpdump überwacht werden, andererseits die Anwendung, welche mit strace überwacht werden.

Um diesen Entwurf zu automatisieren, wurde entschieden, eine separate API zu entwickeln, die es erlaubt, per *request* einen tcpdump auf das *interface* "eth0" der *VM* zu starten und zu beenden – dies ist in der Abbildung als Viereck mit der Aufschrift *tcpDumpServer* zu sehen. Damit ist es möglich, das Messen auf den *worker*-Nodes automatisch durchführen zu können.

Kommen wir jetzt zu der Messung von strace am *server*: Dafür wird einfach ein *entrypoint shellskript* zum *dockerfile* hinzugefügt, das den strace startet und auch speichert. Ein ähnliches Vorgehen wurde beim *client* durchgeführt, dort wird auch ein *entrypoint shellskript* erstellt, welches strace und tcpdump managed. Außerdem wird die oben erstellte API benutzt, um den tcpdump für den *worker*-Node zu starten, bevor die Anwendung durchgeführt und nachdem sie beendet wurde. Sobald die Operation beendet ist, wird es in eine Datei geschrieben und der Prozess mit dem Befehl "sleep" angehalten. Vorher wurden natürlich alle benutzten Prozesse beendet. Die erwähnten Dateien werden dabei in ein geteiltes Verzeichnis mit dem *host system* geschrieben.

Um diese ganze Logik nun miteinander zu verschmelzen, wurde ein *shellskript* auf der *control*-Node erstellt, welches in die jeweiligen Verzeichnisse wechselt und dort den Testaufbau mit Hilfe von Kubernetes-Ressourcen aufsetzt. Dann werden mithilfe des *shell-*

skript je nach Testdurchlauf die *environment variables* der Pods angepasst, damit die Ergebnisse des Versuchs auch in dem richtigen Verzeichnis abgelegt werden und die richtigen Informationen vorhanden sind.

3.4.3 Verifikation

Um sicherzustellen, dass die entwickelte Softwarelösung korrekt läuft, wurden keine separaten Tests entwickelt. Es wurde aber ein manueller Test mit zehn Testdurchläufen durchgeführt, um sicherzustellen, dass die korrekten Daten erstellt werden und die enthaltenen Werte in den Dumps plausibel sind. Das bedeutet: Wenn 10 Anfragen von 1 MegaByte durchgeführt worden sind, muss der tcpdump mindestens 10 Anfragen enthalten und auch mindestens 10 MegaByte groß sein.

3.5 Auswahl der für die Evaluation geeigneten Systeme

Im Folgenden sollen alle zur Evaluation ausgewählten Implementierungen kurz vorgestellt werden. Anschließend werden diese Implementierungen in einer Tabelle aufgelistet, um einen einfachen Überblick zu erhalten. Jede Implementierung hat ein Basis-*dockerfile* erhalten, sowie *client* und *server*, die auf dem Basis-*container* aufbauen. Das ist notwendig, um Anpassungen in der jeweiligen Realisierung einfach durchführen zu können. Dabei basiert jedes Image auf Debian 11 und es werden die jeweiligen Abhängigkeiten hinzugefügt, um die *container* zu bauen.

3.5.1 HTTP/3-Implementierungen

Im Folgenden nun die Liste, in der die verwendeten HTTP/3-Implementierungen vorgestellt werden.

msquic

msquic ist die QUIC-Bibliothek, die von Microsoft Corporation [116] entwickelt wird. Sie kann im .NET-Framework [119] mit dem dort vorhandenen Kestrel-Service und den HTTP-*client* [120] verwendet werden. Das ermöglicht es, dieses Setup sehr schnell zu realisieren. Es unterstützt Linux und ist in C [68] geschrieben. Das Setup bei dieser

Bibliothek stellte eine Herausforderung dar, da es bisher keine Implementierung für SSLKEYLOGFILE [32] gab. Aber es gab einen *Merge Request* auf Github, welcher dieses Feature bereitstellt [48], weshalb die .NET-Runtime komplett gebaut werden musste. Nach dieser Änderung waren alle Funktionalitäten zum Testen bereit.

aioquic

Die Implementierung aioquic [3] ist eine in Python [148] geschriebene Implementierung für QUIC und HTTP/3. Die Realisierung einer Testanwendung war durch die recht einfache Dokumentation und die Bereitstellung von Standards wie SSLKEYLOGFILE sehr trivial.

quic-go

Das Projekt quic-go [153] ist, wie der Name suggeriert, eine Implementierung in der Sprache go-lang [74], es wird von Projekten wie caddy [172] und traefik [218] verwendet. Diese Implementierung hat in der hier getesteten Version einige Performance-Probleme in der UDP-Implementierung [150, 151]. Da sie dennoch in populären *proxys/webserver* vorhanden ist, soll sie hier auch untersucht werden.

quiche

Quiche [26] ist eine von Cloudflare [25] entwickelte QUIC und HTTP/3-Bibliothek, die sogar als *module* für Nginx bereitsteht [28]. Quiche ist in Rust [165] programmiert, was dazu führt, dass sie sehr gut mit *low-level*-Operationen optimiert werden kann.

kwik

Kwik ist eine in Java [136] geschriebene Implementierung von QUIC, wobei die Implementierung von QUIC in kwik [47] stattfindet und die HTTP/3-Implementierung in flupke [45]. Zu beachten ist bei dieser Implementierung, dass es sich um eine Test-Implementierung handelt und TLS auch selbst implementiert worden ist, was dazu führen kann, dass diese mehr Sicherheitsproblematiken enthält als eine der häufiger genutzten

Bibliotheken wie z.B. boringssl [73]. Dieses ist eine Ausnahme in der Auswahl, da es die einzige Java-Implementierung war, weshalb sie trotz ihres Status evaluiert werden soll.

lsquic

Mit lsquic [109] wurde eine Implementierung von QUIC und HTTP/3 ausgewählt, die schon den Versionsstandard QUIC V2 mit implementiert hat [224].

picoquic

Die Implementierung von picoquic [146] wird von Software-Architekten gewartet, die sich auch an der Spezifikation signifikant beteiligen, wie z.B. Christian Huitema [87, 110]. Die Implementierung wurde in C entwickelt und ist relativ modular aufgebaut.

Weitere Implementierungen

Es gibt weitere Implementierungen, die QUIC bereitstellen, aber keine HTTP/3-Implementierung haben, wie z.B. aktuell aws-s2n [6]. Außerdem gibt es reine *server*-Implementierungen wie z.B. h2o [40]. Weitere Implementierungen, die aufgrund des Umfangs dieser Arbeit nicht in diese Evaluation aufgenommen wurden, sind z.B. neqo [124], ngtcp [129] und xquic [5]. Die Implementierungen, die hier ausgewählt wurden, sind eine Mischung aus aktuellen, gepflegten und auf unterschiedlichsten Sprachen aufbauenden Implementierungen, um ein möglichst breites Spektrum an Implementierungen evaluieren zu können.

3.5.2 HTTP/2- und HTTP/1.1-Implementierungen

Im Folgenden nun eine Auflistung der ausgewählten HTTP/2- und HTTP/1.1-Implementierungen.

Nginx

Nginx ist ein von F5, Inc verwalteter HTTP/2- und HTTP/1.1-*server* [59]. Er bietet unterschiedliche Betriebsmodi an: als *cache* oder *static deliviery server*, aber auch als *server* für Skriptsprachen wie PHP [206]. Durch seine langanhaltende und weite Verbreitung ist dieser *server* eine gute Wahl für diese Evaluation.

Apache2

Ähnliches gilt für Apache2. Es ist lange auf den Markt und weit verbreitet. Es stellt die meisten Funktionalitäten von Nginx ebenfalls bereit und ist ähnlich weit verbreitet. Es wird von der Apache Software Foundation entwickelt und gepflegt [178].

Weitere

Es gibt auch weitere Implementierungen als Bibliothek oder als direkte Anwendung von HTTP/1.1 und HTTP/2 [172, 218, 88]. Im Falle einer Bibliothek müsste der *server* aber selbst implementiert werden, was Implementierungsabweichungen oder sprachspezifische Eigenheiten mit sich führen kann. Deshalb wurde sich für den Weg entschieden, als Referenz Nginx und Apache2 zu verwenden, da diese weit verbreitet, gut gepflegt und lange auf den Markt sind. [173]. Andere *webserver* können sicherlich das Gleiche abbilden, aber durch die weite Verbreitung sind diese zwei *server* gute Kandidaten für die Evaluation.

Zusammenfassung

In der Tabelle 3.1 ist eine Zusammenfassung aller zu evaluierender Implementierungen der HTTP-Protokolle zu sehen. Dabei werden noch einmal die Schlüsselinformationen zu den jeweiligen Implementierungen gezeigt, wie Projektname, die Programmiersprache, die unterstützten HTTP-Version, die aktiven Entwickler gemäß Github und die Initiatoren des Projekts.

Projektname	Programmiersprache	Unterstützte HTTP Versionen	Aktive Entwickler	Initiator
msquic	C (.NET Benutzung)	H3	71	Microsoft Corporation
aioquic	Python3	H3	23	Jeremy Lainé
quiche	Rust	H3	75	Cloudflare, Inc
quic-go	Go	H3	85	Marten Seemann & Lucas Clemente
kwik	Java	H3	3	Peter Doornbosch
lsquic	C	H3	48	LiteSpeed Technologies Inc
picoquic	C	H3	37	Christian Huitema
Nginx	C	H1.1,H2	64 (benutzen kein Github)	Igor Sysoev (F5, Inc.)
Apache2	C	H1.1,H2	47 (benutzen kein Github)	Robert McCool (The Apache Software Foundation)

Tabelle 3.1: Liste der zu evaluierenden Implementierungen (besucht am 01.07.2023)

4 Evaluation

Im folgenden Abschnitt wird die Evaluation besprochen, dabei werden Vergleichskriterien für die Versuche definiert. Danach werden die Versuchsaufbauten beschrieben, gefolgt von den Erwartungen. Anschließend werden die Experimente durchgeführt. Danach werden die Ergebnisse dargestellt und anschließend interpretiert.

4.1 Vergleichskriterien

Um eine Anwendung vergleichen zu können, müssen Vergleichskriterien festgelegt werden. Im Folgenden sollen mehrere Webkommunikationsprotokolle innerhalb von Kubernetes evaluiert werden, um herauszufinden, wie die neueren Protokolle sich im Verhältnis zu den etwas älteren Protokollen verhalten.

Ein interessanter Aspekt dabei ist die gesamte Kommunikationsdauer zwischen Sender und Empfänger. Aufgrund der vielen Schritte, die ein Paket von A nach B in dem definierten System benötigt, ist es interessant zu beobachten, ob ein Protokoll wie TCP aufgrund seiner langen Entwicklungszeit und damit verbundenen Optimierungszeit besser performt als ein neueres, auf UDP aufbauendes Protokoll.

Außerdem ist ein Vergleich der jeweiligen *system_ calls* bei der gleichen Anzahl an Anfragen spannend. Aufgrund des Aufbaus von HTTP/3, welches QUIC verwendet, was wiederum in allen ausgewählten Implementierungen im *user space* implementiert ist, stellt sich die Frage, ob es einen Einfluss zwischen diesen unterschiedlichen Implementierungen gibt.

Noch ein Vergleichskriterium in dieser Arbeit soll die Anzahl der versendeten Pakete sein. Aufgrund der unterschiedlichen Strukturen der Protokolle könnte es interessant sein, diese zu beobachten.

Desweiteren sollte der Faktor der Größe der jeweiligen Dateien mit beachtet werden. Die Fragestellung hier lautet, ob die Größe der zu transferierenden Datei irgendwelche Einflüsse auf das System hat.

4.2 Versuchsaufbauten

Der Kubernetes-*cluster* wird nach jedem Versuch in die Ausgangssituation zurückgebracht. Diese Ausgangssituation besteht aus zwei Deployments mit der *server*- und *client*-Anwendung, dabei sind beim *client* einige Zähler hinzugefügt, um die Dauer der jeweiligen Sende-Abschnitte aufzuzeichnen – erstens für die einzelne Anfrage, zweitens für den gesamten Anfrage-Zyklus. Außerdem wird die gesamte Laufzeit mithilfe dieser Zähler gemessen.

Beide Anwendungen, *server* und *client*, werden mit strace gestartet, um die *system_ calls* während des Versuchsaufbaus zu protokollieren.

Der *client* schreibt außerdem mithilfe des SSLKEYLOGFILE seine TLS-1.3-Schlüsselinformation heraus, um das Analysieren innerhalb der tcpdumps einfacher zu machen. Durch das Tool tcpdumps wird der Netzwerkverkehr an zwei Stellen mitgeschnitten. Einmal innerhalb des *client containers* und einmal auf der *server*-Seite auf dem externen Netzwerk-*interface*. Die Kommunikation zwischen *client* und *server* lauft über einen Service (eine Kubernetes-Ressource), der die Aufrufe über Iptables-Regeln weiterleitet.

Bei jedem Versuch wird nur die Kommunikation zwischen den jeweiligen implementierten Bibliotheken *client* und *server* evaluiert und damit keine Interoperabilität zwischen den einzelnen Implementierungen getestet. Als Ausnahme von der Regel wird Apache2 und Nginx benutzt, da diese keinen eigenen *client* bereitstellen. Hier wird einfach eine eigene Rust-Implementierung [165] mit der hyper 1.0.0-rc.3 [88] und rustls 0.20 [166] mit libc 0.2.141 [164] für extra *socket options* verwendet. Außerdem wurde *early data* von TLS 1.3 hinzugefügt – in dieser Arbeit auch 0-RTT genannt.

Pro Versuchsdurchlauf werden Evaluationen mit folgenden Werten durchgeführt. Einer 100KB, 1MB und einer 10MB großen JSON-Datei.

Dabei sind 100KB und 10MB die Ausreißer nach oben und unten, der mittlere Wert stellt den Normalwert dar. Diese Werte wurden gewählt, da eine durchschnittliche Seite etwas 2MB groß ist [138], während größere Seiten etwa 4MB umfassen, was die Evaluation von 1 - 10 MB als realistische API-Kommunikation durchgehen lässt. Dabei wird jeder Versuch 10 Mal durchlaufen und 10 Mal durchgeführt, damit sollten genug Daten vorhanden sein, um ein gutes Spektrum an Messdaten für den jeweiligen Versuch zu erhalten.

Das Experiment-System ist ein iMac 2017 mit einer Intel Core i5-7500 3400 MHz (4 Kerne) mit Mac OS Ventura 13.2.1, 64GB 2400MHz DDR4 Random Access Memory (RAM) und Virtualbox in Version 7.0.r154219. Jede Node (VM) hat 6GB RAM und 2 Prozessoren konfiguriert bekommen und eine 1GB Netzwerkschnittstelle in das NAT-Netzwerk.

Die Maximum Transmission Unit (MTU) wurde nicht verändert. Diese entspricht dem Standard 1500 auf allen vorhandenen Systemen. Die *congestion-control*-Algorithmen wurden auf Cubic gesetzt, mit Ausnahme von kwik und aioquic, diese benutzten NewReno [81].

Die Erweiterung für TFO wurde aktiviert für die auf TCP basierenden Implementierungen. Alle weiteren Einstellungen entsprechen der Standard-Installation von Debian 11. Es wurden ansonsten auch keine Veränderungen an den jeweiligen Implementierungskonfigurationen durchgeführt.

4.3 Experimente

Es sollen drei Experimentszenarien mit dem eben definierten Setup evaluiert werden:

4.3.1 Versuch 1

In diesem Versuch geht es darum, die Kommunikation von einem *client* zu einem *server* zu evaluieren. Dieses entspricht der trivialsten möglichen Situation in diesem Versuchsszenario. Dabei stellt der *client* A die Anfragen zum *server* B über eine Kubernetes-Service-Definition.

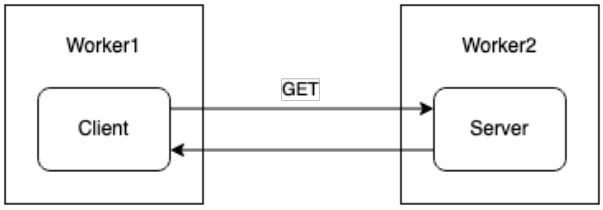

Abbildung 4.1: Versuch 1

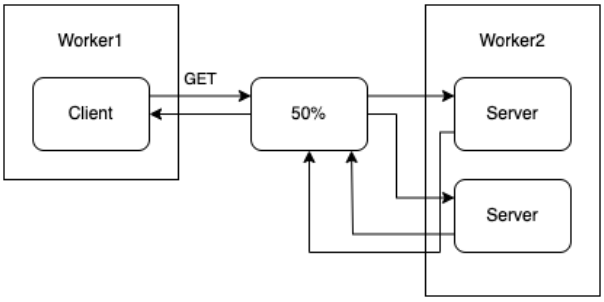

Abbildung 4.2: Versuch 2

Dieses Verhalten ist auch in der Abbildung 4.1 dargestellt. Ein Viereck entspricht dem jeweiligen virtuellen *server* und die Vierecke mit abgerundeten Ecken entsprechen der jeweiligen Anwendung innerhalb des *servers*. Die Pfeile sind die abstrakten Kommunikationswege. Außerdem ist der Anfrage-Pfeil mit der HTTP-Methode GET gekennzeichnet, welche als Operation ausgeführt wird. Der Kubernetes-Service ist hier nicht zu sehen, da er eine NAT-Operation durchführt und kein *load balancing* und damit keine Richtungsänderung in der abstrakten Darstellung durchführt.

4.3.2 Versuch 2

In dem zweiten Versuch soll von einem *client* zu zwei *server* auf den gleichen Node, also auf der gleichen virtuellen Maschine kommuniziert werden. Dadurch kann evaluiert werden, ob es einen Unterschied macht, mit zwei isolierten Anwendungen zu kommunizieren. Außerdem muss in diesem Szenario immer mindestens eine zweite Verbindung aufgebaut werden.

Dieser Sachverhalt ist in Abbildung 4.2 noch einmal zu sehen. Die Konstruktion ist identisch mit der von Versuch 1, mit der Ausnahme, dass nun das Zwischenelement – der Service – als Element zwischen Worker1 und Worker2 vorhanden ist, außerdem

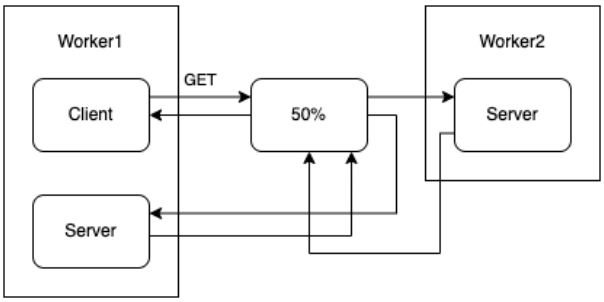

Abbildung 4.3: Versuch 3

existieren nun zwei *server*. Der Service ist hier mit 50% gekennzeichnet, da er ein *load balancing* mit einer Verteilungschance von 50% durchführt. Dieses Verhalten liegt an der Iptables-Konfiguration von Kubernetes, das kein *roundrobin* nutzt, sondern nur eine Wahrscheinlichkeit [170]. Das kann dazu führen, dass z.B. in drei Fällen der eine und in den sieben anderen Fällen der andere *server* angesprochen wird. Aufgrund der zehn Durchläufe sollte sich das aber in der Gesamtauswertung ausgleichen.

4.3.3 Versuch 3

Im dritten Versuch wird wie beim zweiten Versuch die Kommunikation von einem *client* zu zwei *servers* durchgeführt. Hier wird aber die Lokalität eines *server* verändert, indem er auf die Node des *client* gelegt wird. Das führt dazu, dass beide *server* auf unterschiedlichen VM laufen, jedoch ist die Lokalität zwischen einem der *server* und dem *client* näher, da keine Kommunikation außerhalb einer VM stattfinden muss.

Dieser Sachverhalt ist noch einmal in der Abbildung 4.3 zu sehen. Es entspricht genau der Abbildung 2, mit dem Unterschied, dass einer der *server* von Worker2 auf Worker1 versetzt wurde.

4.4 Erwartung

Um das zu erwartende Verhalten zu analysieren, müssen mehrere Schichten des OSI-Modells [236] und der Gesamtarchitektur betrachtet werden. Angefangen bei Kubernetes: Dieses nutzt, wie bereits in Aufbau Kubernetes (3.2) erörtert, in dieser Konfiguration ein

overlay-Netzwerk. Für diese Untersuchung interessiert uns der Aspekt, dass die eigentlichen Nutzdaten durch dieses reduziert werden. In dieser Konfiguration mit VXLAN mithilfe von Calico entspricht es einer Reduzierung der Nutzdaten von 50 Bytes.

Die Problematik von physikalischer Hardware und damit verbunden gegebenenfalls unterschiedlichem Routing der Pakete kann hier ignoriert werden, da es sich um eine Virtualisierung handelt und damit maximal Einflüsse vom Betriebssystem wie *scheduling* dort ein Faktor sein kann, was aber zu vernachlässigen sein sollte.

Ein weiterer Aspekt in Kubernetes wären die Iptables, die in Abschnitt Aufbau Kubernetes (3.2) erwähnt werden, um das Weiterleiten zum eigentlichen Dienst zu ermöglichen. Da dieser unabhängig von der eigentlichen Anwendung ist, sollte auch das hier keinen größeren erwartbaren Einfluss haben, mit der Ausnahme einer geringfügigen Verschlechterung durch die Abarbeitung der Regeln.

Außerdem sind weitere signifikante *delays* jeglicher Art aufgrund der *cluster*-Architektur nicht zu erwarten, da es alles Virtualisierungen sind, die folglich in diesem Aufbau eine hohe Lokalität zueinander haben.

Einen Schritt weiter hoch im OSI-Modell [236] kommen wir zum Transport. Im Transport sind die Protokolle HTTP/1.1 und HTTP/2 im Vergleich zu HTTP/3 grundverschieden. Während die ersten beiden Versionen auf TCP aufbauen, benutzt HTTP/3 hier QUIC, welches UDP nutzt. Durch die Paket-Architekturen, die in Kapitel Aufbau der Pakete (3.3) gezeigt wurden, sollte QUIC mehr Nutzdaten transferieren können als die TCP-Implementierung.

Der Aspekt von mehreren *streams* sollte hier nicht zum Tragen kommen, da die definierten Nutzdaten hier nur ein in sich geschlossenes Dokument sind und folglich in den Implementierungen nur als ein *stream* gehandhabt werden.

Ein Nachteil des aktuellen Vorgehens ist, dass QUIC im *user space* implementiert ist, daraus folgend muss für jede Operation an den Kontrollinformationen ein Kontextwechsel im System durchgeführt werden. Dieser Kontextwechsel kostet bei den meisten *system_ calls* ein wenig Leistung. Daher könnte es zu einer messbaren Verschlechterung der Geschwindigkeit beitragen. Dazu kommen die implementierungsspezifischen Unterschiede von QUIC. Durch die Divergenz an Sprachen können die Handhabungen signifikant

unterschiedlich sein, von Speichermanagement bis hin zum *scheduling*. Weitere Aspekte der Transportprotokolle sollten in den definierten Szenarien keinen signifikanten Einfluss haben.

Nun zur Anwendungsschicht. Die Protokolle HTTP/1.1 und HTTP/2 unterscheiden sich signifikant voneinander in dem Aspekt, dass wir mit HTTP/1.1 ein zeichenbasiertes Protokoll haben und mit HTTP/2 ein bytebasiertes. Das sollte dazu führen, dass HTTP/1.1 theoretisch einen Nachteil hat, da hier aber die Elemente wie *header* und *trail* Information sehr gering sind, sollte es, wie später erörtert wird, eher zu einen Nachteil für HTTP/2 führen. Die TLS-Implementierung wird bei beiden einfach nur vorgesetzt. Im Gegensatz dazu hat HTTP/3, das auch bytebasiert ist, aber TLS direkt mit integriert. Das bedeutet, dass TLS + HTTP/1.1 oder HTTP/2 zwei *handshake*-Phasen durchlaufen müssen, einerseits die von TCP, andererseits die von TLS. Auch wenn die TLS mithilfe von TLS 1.3 verkürzt werden und TCP mithilfe von TFO seinen *handshake* verkürzen kann (aber nur bei einem zweiten *request*), sind es dennoch mindestens 1 RTT mehr als bei HTTP/3 bzw. QUIC. Hier ist zu erwarten, dass durch die Reduzierung eine neue Verbindung schneller aufgebaut werden kann und auch bei bekannten Verbindungen ein schnellerer erneuter Verbindungsaufbau möglich ist. Wie bei den anderen Aspekten bereits erwähnt, muss hier vieles von HTTP/2 nicht neu auf das Transportprotokoll hinzugefügt werden. Viele Aspekte sind bereits vorhanden, wie z.B. *streams*, was dazu führt, dass die Nutzdaten allein dadurch ein wenig höher sein sollten, wie in Abschnitt Aufbau der Pakete (3.3) zusammengefasst.

Zusammenfassend zur generellen Erwartung: Es ist zu erwarten, dass HTTP/3 seine Vorteile vor allem im verkürzten Verbindungsaufbau und der reduzierten Kontrollinformation hat. Im Zusammenhang mit der erhöhten Kontrollinformation, die durch das *overlay*-Netzwerk entsteht, sollten Vorteile beim Testen mit größeren Datenmengen entstehen. Bei kleineren Datenmengen könnte der Kontextwechsel seinen Einfluss zeigen, ansonsten sollte durch den optimierten Verbindungsaufbau QUIC bzw. HTTP/3 einen kleinen Vorteil mit sich bringen, der durch die geringe Anzahl an *requests* wahrscheinlich keinen signifikanten Einfluss haben sollte.

Hier folgt eine detaillierte Analyse der Erwartungen für die gewählten Szenarien.

Szenario 1: Es sollte bei steigender Größe der Nutzdaten ein Vorteil der HTTP/3-Implementierung im Vergleich zu den anderen HTTP-Implementierungen geben. Das

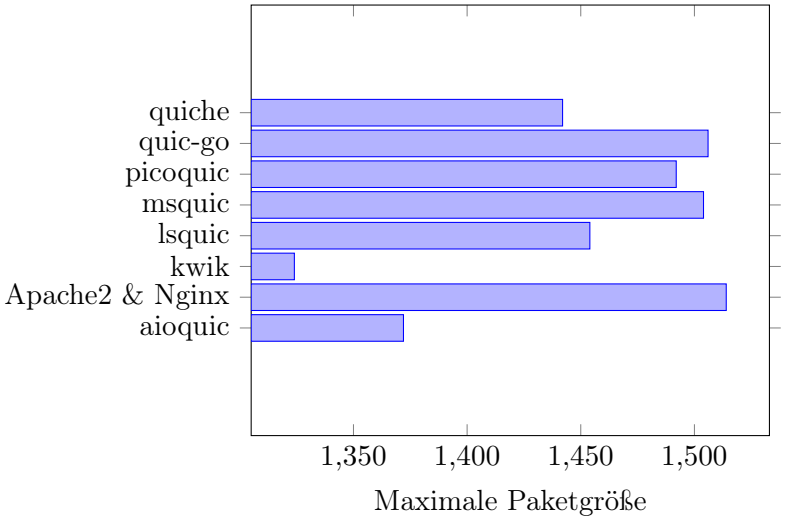

Abbildung 4.4: Ergebnisse Maximale Paketgröße

resultiert aus der Akkumulation des reduzierten Verbindungsaufbaus und der erhöhten Nutzdaten.

Szenario 2: Hier ist dasselbe Verhalten wie in Szenario 1 zu erwarten. Nur, dass durch den Wechsel zwischen den zwei Diensten der Verbindungsaufbau mindestens doppelt stattfinden muss, was einen kleinen Vorteil für QUIC bedeuten sollte. Außerdem wird nach jeder Anfrage eine neue Verbindung aufgebaut, was bei vielen kleinen *request* auch einen messbaren Unterschied zeigen sollte.

Szenario 3: Auch hier ist dasselbe Verhalten wie in Szenario 1 und 2 zu erwarten. Hier wird nur bei dem einen *server* der Netzwerkverkehr lokal ausgeführt, was dazu führen kann, dass TCP einen Vorteil bringt, da die Bearbeitung durch die geringere Anzahl an Kontextwechseln effizienter sein sollte.

4.5 Ergebnisse

Als Ergebnisse sind Messtabellen entstanden, von diesen sollen folgend kurz die signifikanten Messungen grafisch dargestellt werden.

Das erste interessante Messergebnis der Untersuchung ist in Abbildung 4.4 zu sehen. Dort ist auf der X-Achse die Größe in Bytes zu sehen, auf der Y-Achse einige unterschiedliche

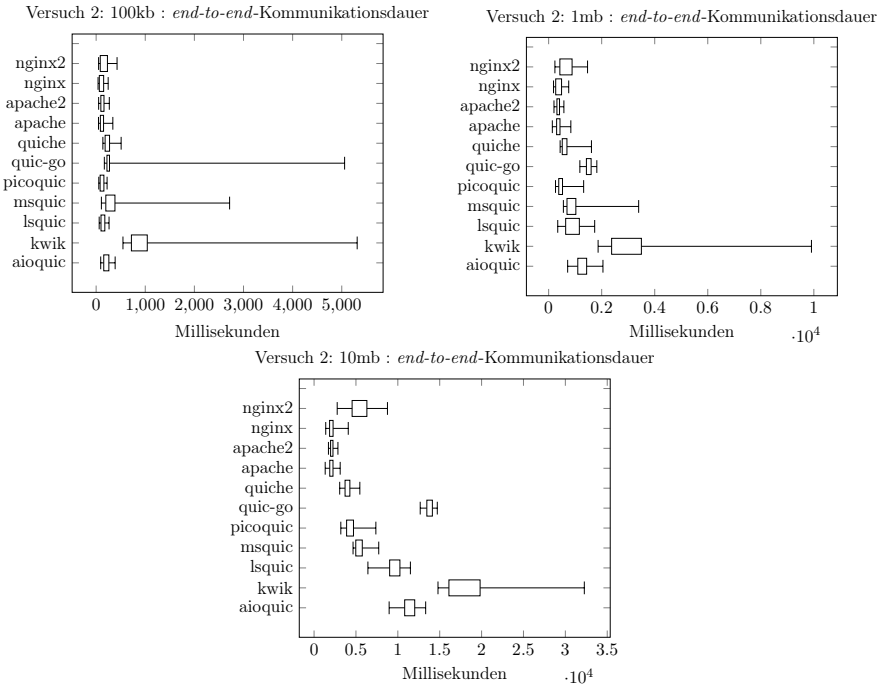

Abbildung 4.5: Ergebnisse zu Versuch 2: *end-to-end*-Kommunikationsdauer

Implementierungen. Zu erkennen ist dort die maximale Ausnutzung der MTU. Dabei sind einige HTTP/3-Implementierung nicht an das Limit von 1500 gekommen.

In den folgenden Tabellen werden alle getesteten Implementierungen aufgeführt, dabei sind die Implementierungen von Nginx und Apache2 einmal ohne die HTTP-Versionsangabe aufgeführt, was für die Version HTTP/1.1 steht. Version 2 ist hingegen angegeben. Desweiteren sind einige Implementierungen mit Generic segmentation offload (GSO) gekennzeichnet, was bedeutet, das dieses dort aktiv war.

Nun zu den Szenarien 1 bis 3: Diese verhielten sich ähnlich in der Dauer der *end-to-end*-Kommunikation, weshalb hier nur Szenario 2 gezeigt wird. Alle weiteren sind im Anhang zu finden unter A.1. In der Abbildung 4.5 sind die drei Graphen zu sehen, und zwar für 100KB, 1MB und 10MB. Dabei ist jeweils die X-Achse die Zeit in Millisekunden und die Y-Achse die Version. Zu erkennen ist, dass einige HTTP/3-Varianten bei 100KB mithalten können oder sogar besser sind als HTTP/2 oder HTTP/1.1. Bei der 1MB-Variante ist schon zu erkennen, dass HTTP/3 schlechter dasteht als die anderen Varianten und bei 10MB sehen wir den Sachverhalt genau so. In dieser Abbildung ist im

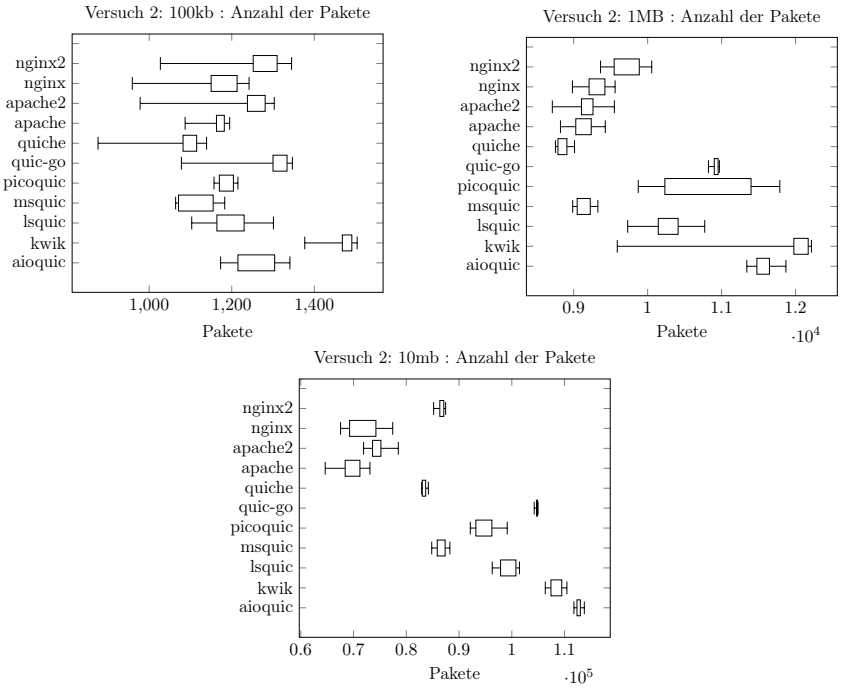

Abbildung 4.6: Ergebnisse zu Versuch 2: Versendete Pakete

Gegensatz zu Szenario 1 keine GSO-Variante von HTTP/2 und HTTP/1.1 zu sehen, da sie keinen signifikanten Einfluss auf die Geschwindigkeit hatten.

Ähnlich wie eben verhält sich das Ganze auf der Paketebene. Dort sind Szenario 1 bis 3 sehr ähnlich, weshalb im Folgenden wieder nur Szenario 2 betrachtet werden soll. In der Abbildung 4.6 sind die drei Graphen zu sehen für 100KB, 1MB und 10MB, dabei steht die X-Achse für die Pakete, die während der Kommunikation ausgetauscht worden sind, und die Y-Achse für die Version. Hier zu erkennen ist, dass einige HTTP/3-Implementierungen einen Vorteil in der Anzahl der Pakete bis hin zu 10MB haben, dort übernehmen wieder die HTTP/2- und HTTP/1.1-Implementierungen die Führung.

Als letztes zu den Messung der *sytem_ calls*. Als relevant wurden hier die Gesamtzahl der *sytem_ calls* herausgezogen. Genau wie die anderen zwei Messwerte sind auch hier nicht viele Änderungen zwischen den unterschiedlichen Szenarien zu sehen, weshalb hier wieder die Ergebnisse von Szenario 2 dargestellt werden und der Rest im Anhang zu finden ist (A.1). In der Abbildung 4.7 sind die drei Graphen zu sehen für 100KB, 1MB und 10MB, dabei ist jeweils die X-Achse die Anzahl der *sytem_ calls* und die Y-Achse die Version. Zu erkennen ist, dass die HTTP/3-Implementierung mehr *sytem_ calls* benutzt

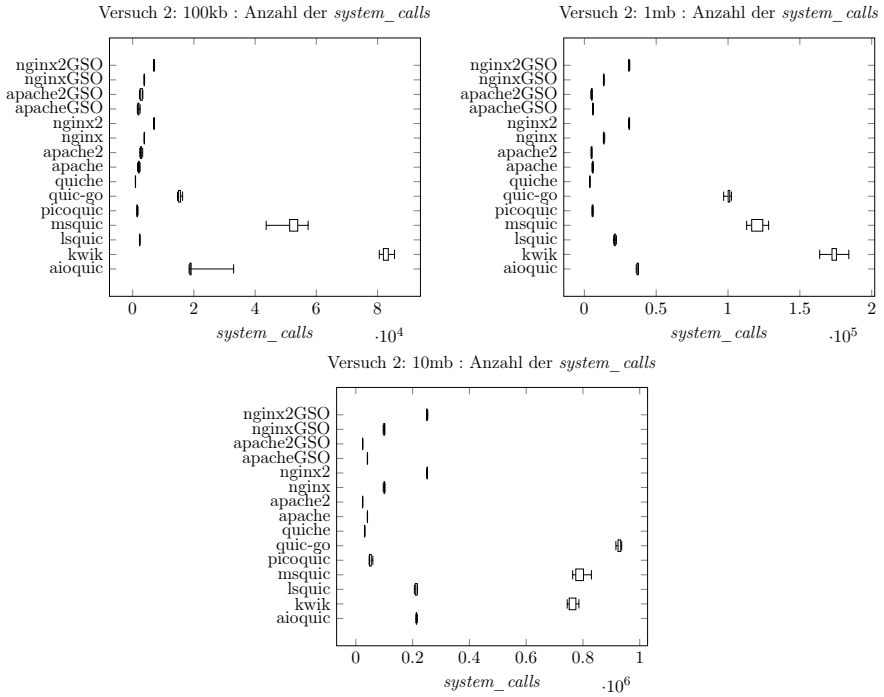

Abbildung 4.7: Ergebnisse zu Versuch 2: Anzahl der *system_ calls*

als die HTTP/2- und HTTP/1.1-Implementierungen.

Bei der Analyse der tcpdump ist außerdem ein Implementierungsdetail in der Paketgröße aufgefallen. Im Abschnitt Aufbau der Pakete (3.3) wurden für TCP-*overhead* die Werte 20 Byte angenommen. In dem Versuch waren es 32 Bytes, was bedeutet, dass einige Optionen gesetzt worden sind. Ansonsten ist ein erweiterter *overhead* von 16 Bytes [210, sec 5.3] für die Verschlüsselung dazugekommen. Die Authenticated Encryption with Associated Data (AEAD), die bei den meisten *cipher suits* benutzt wird, fügt dieses hinzu. Das führt dazu, dass Protokolle wie HTTP/1.1, die den ganzen *stream* nur einmal verschlüsseln und dann wieder zusammenbauen, diesen *overhead* nur einmal in der Kommunikation haben. Bei HTTP/2 sieht es schon ein wenig schlechter aus, dieser benutzt *streams*, welche aber auch über mehrere Pakete verschlüsselt werden können, da die Verschlüsselung aufgesetzt und nicht integriert ist. Bei dem Beispiel mit Nginx sind es beispielhaft acht *stream frames* pro TLS-*stream*. Ganz schlecht sieht dieses Verhalten bei QUIC aus, da es für jedes Paket ein *overhead* von diesen 16 Bytes benötigt, weil jedes Paket individuell verschlüsselt werden muss. Außerdem existieren durch die variable Implementierung von der *destination id* unterschiedlich große Anforderung. Bei der quiche

z.B. werden 20 Bytes benutzt, für die *destination id* und innerhalb von quic-go nur 6 Bytes.

Ein weiterer Aspekt, der festgestellt werden konnte, ist, dass die Implementierungen unterschiedlichste *system_calls* verwenden, um mit Sockets zu interagieren. Dieses ist nur eine Feststellung, hatte aber keine direkten zuordenbaren Einflüsse auf die Implementierungen.

Abseits der eigentlichen Evaluation ist aufgefallen, dass die *namespaces* der aktuellen sysctl keine Kopie von allen Werten anlegen, sondern bei einigen Konfigurationen die Standardwerte verwendet werden, was dazu führt, dass diese Konfigurationen in Kubernetes freigegeben werden müssen [204], damit die jeweilige Konfiguration innerhalb des *containers* gilt. In dieser Evaluation war das die TFO-Konfiguration.

4.6 Interpretation

In den Ergebnissen ist zu erkennen, dass einige Implementierungen wie quiche und msquic im Bereich von 1KB ähnlich bis schneller als HTTP/1.1 und HTTP/2 performen, wobei die Unterschiede so marginal sind, dass man sie als identisch bezeichnen könnte. Sobald die übertragene Datenmenge größer wird, zeigt sich, dass TCP mit diesem Szenario besser zurechtkommt. Hier ist zu erkennen, dass HTTP/2 sogar schlechter performt als HTTP/1.1. Das mag zu einem kleinen Teil daran liegen, dass *early data* mit HTTP/2 nicht gut arbeitet, aber auch das generelle Design von HTTP/2 keinen Vorteil in diesem Szenario bieten kann. Da nur ein *stream* verwendet wird und wie oben erwähnt die Verschlüsselung blockweise durchgeführt wird, entsteht ein *overhead* an mehr Daten. Außerdem ist das Parsing leicht aufwendiger durch die in *hpack* durchgeführte Kompression.

Ein weiterer Grund für die schlechtere Performance von QUIC könnte das *overlay*-Netzwerk sein, welches hier zum Tragen kommt. In Versuchen von Novianti und Basuki [130] war zu sehen, dass *overlay*-Netzwerke häufig schlechter mit UDP performen. Leider ist dort nicht geklärt, ob das durch Hardware entstanden oder ob es ein generelles Kernel-Problem ist.

Ein auffälliger Aspekt ist, dass die Werte der *system_calls* relativ gut mit der Geschwindigkeit einer Implementierung korrelieren. Zu sehen ist das in Abbildung 4.8. Diese

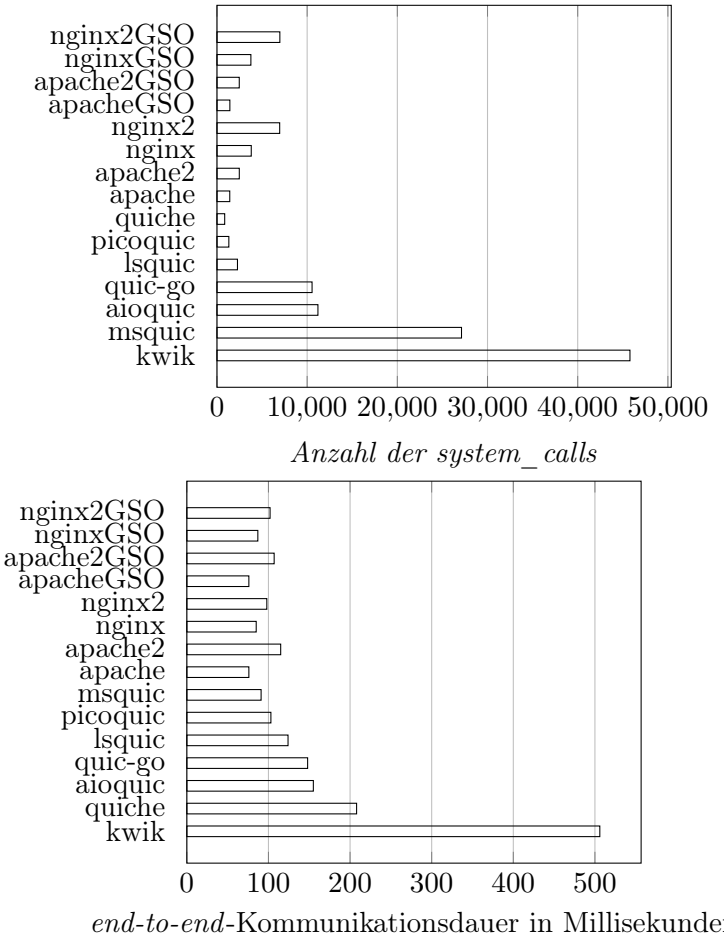

Abbildung 4.8: Versuch 2: 100kb: Vergleich von Anzahl der *system_ calls* und *end-to-end*-Kommunikationsdauer (Durchschnitt)

Abbildung zeigt, dass bei der Implementierung von kiwik, quic-go, picoquic und lsquic die *system_ calls* im engen Verhältnis zur Dauer der *end-to-end*-Kommunikation stehen. Gleichzeitig haben wir aber auch einige Messungen, die nicht gut mit dieser Aussage harmonieren, wie msquic oder quiche. Es handelt sich hier also um kein 1-zu 1-Mapping, sondern eher um eine Tendenz. Selbst bei Nginx und Apache2 sind Tendenzen zu erkennen.

Ein Faktor, der hier sicher auch sehr zum Tragen kommt, ist die Art der Implementierung. Bei den Implementierungen von kiwik, golang und aioquic sehen wir, dass die Performance eher schlecht ist. Dieses hat sicher mit der hohen Abstraktion der Sprachen zu tun, was die Performance signifikant beeinflussen wird und definitiv als ein Problem von Implementierungen im *user space* angesehen werden kann. Wenn sich dagegen die *low-level*-Implementierungen in Rust oder C angeschaut werden, ist zu erkennen, dass diese im Verhältnis zu diesen Sprachen deutlich besser performen. Die Implementierung von msquic hat den Weg gewählt, die Implementierung von QUIC in C zu schreiben und als eine Bibliothek in .NET verfügbar zu machen. Dieses Vorgehen zeigt in den Untersuchungen einen Performancevorteil, weshalb dieser Ansatz bei den eben aufgeführten Sprachen sicherlich ebenfalls einen Vorteil bringen könnte. Dennoch bleiben immer kleine Performanceverluste gegenüber anderen Sprachen, aber weniger durch die Nutzung von HTTP/3, sondern durch das generell höhere Abstraktionslevel der Sprache.

Zu finden ist auch, dass die MTU innerhalb der QUIC-Implementierung nicht maximal ausgenutzt werden kann, wie in der Abbildung 4.4 zu sehen. Dieses Verhalten kann an der relativ komplexen Struktur liegen oder es ist ein Implementierungsdetail, auffällig ist aber, dass die meisten hier evaluierten Implementierungen dieses Verhalten aufwiesen. Aber durch dieses Verhalten werden selbst bei den trivialen Übertragungen wie z.B. bei Szenario 1 mit 100KB mehr Pakete versendet als bei HTTP/1.1- oder HTTP/2-Implementierungen. Quic-go nutzt dabei schon das RFC 8899 [60], um die MTU zu bestimmen. Im RFC von QUIC [92, sec. 14.2] steht, dass entweder Datagram Packetization Layer Path MTU Discovery (DPLPMTUD) oder Path MTU Discovery (PMTUD) [38] oder die kleinste erlaubte MTU verwendet werden sollte. Implementierungen wie kiwik oder aioquic benutzen einfach einen konstanten Wert [46, 4]. Interessant ist auch, dass es Implementierungen wie lsquic gibt, welche damit werben, dass sie DPLPMTUD implementieren, aber nicht an die maximale MTU herankommen [217]. Dieses kann aber auch durch den *client* mit der max_udp_payload_size option [92, sec. 18.2] im QUIC-*handshake* limitiert werden. Daraus resultierend ist es auch hier durch die vielfältigen

Möglichkeiten, QUIC zu implementieren, durch drei oben genannten unterschiedlichen Wege dazu gekommen, dass QUIC unterschiedlich implementiert wurde.

Zu beachten ist, dass in den Versuchen auch die Optimierung nicht in Betracht gezogen wurde. Das bedeutet, es wurden bei keiner Implementierung Parameter verändert, um das Verhalten zu optimieren.

Zusammenfassend ist die Hauptfeststellung, dass TCP und die damit verbundenen Protokolle von HTTP/1.1 und HTTP/2 in diesen Versuchen in der Regel am besten abschnitten. Einzig bei kleinen Dateien ist QUIC mit der HTTP/3-Implementierung gleichauf gewesen. Das ganze Verhalten sollte bei einem ausgelasteten System mit echter Netzwerk-Hardware noch drastischer sein, da TCP segmentation offload (TSO) dort einen signifikanten Vorteil bieten kann. Dieses ist sogar durch VXLAN hindurch möglich [131]. Die GSO hatte in diesen Versuchen dagegen gar keinen Einfluss. Durch die gewählte Verschlüsselung hatte HTTP/3 in diesem Versuchsaufbau effektiv sogar einen Nachteil gegenüber HTTP/1.1 und HTTP/2.

Ein zweiter Punkt, der hier abgeleitet werden kann, ist die Benutzung von *low-level*-Sprachen. Diese bieten bei der Implementierung von *user-spaces*-Transportprotokollen ein signifikanten Vorteil.

Und als Drittes ist das Vergleichen von *system_ calls* eine gute Orientierung, um die Performance einer Protokollimplementierung abzuschätzen, ohne sie modifizieren zu müssen. Aber wie in diesen Versuchen zu erkennen ist, gibt es Ausnahmen und sollte maximal als Einstieg für weitere Untersuchungen dienen.

Ein Aspekt, der bei dieser Evaluierung zumindest erwähnt werden muss, sind die Risiken, die mit TFO und 0-RTT mit TLS verbunden sind. In den RFC sind die Problematiken von vor allem *replay*-Attacken sehr gut aufgeführt [209], [160, appendix-E.5] [23, sec. 5]. Das führt auch dazu, dass, wenn diese Mechanismen möglich sind, sie meistens begrenzt sind auf GET-Methoden, da diese per Definition idempotent sind [90] und dadurch theoretisch sicher vor *replay*-Attacken. Aber die meisten *webserver* erlauben auch, eine nicht idempotente Operation mit GET durchzuführen, was dazu führt, dass diese Methode nicht als wirklich sicher angesehen werden kann.

Noch einmal rückblickend auf die Erwartungshaltung. Zuerst ist die Annahme, die auf-

grund der Paketgröße getroffen worden ist, nicht zutreffend. Einerseits durch die Erkenntnis, dass die Pakete in der Realität eine andere Größe besitzen und durch zweitens die Erkenntnis, dass die Implementierungen der Erkennung der MTU ziemlich unterschiedlich sind. Wobei zumindest bei der Evaluation der Implementierung von quiche immer ein sehr annäherndes Verhalten zu HTTP/1.1 und HTTP/2 zu sehen war, was folglich bei einer besseren MTU-Ausnutzung in dieser Evaluation sicherlich auch einige Vorteile gebracht hätte. Die Erwartung, dass die reduzierte Anzahl an Paketen einen messbaren Vorteil bringt, konnte in diesem Versuchsaufbau nicht bestätigt werden. Diese Erwartung wurde durch die Messwerte widerlegt.

5 Zusammenfassung

In dieser Arbeit wurde sich mit modernen Webkommunikationsprotokollen beschäftigt, welche in einem Kubernetes-*cluster* evaluiert wurden.

Gestartet wurde mit einer Einleitung in die Thematik und einen Überblick über den Inhalt. Dafür wurde ein Überblick über die relevanten Protokolle jeder Schicht gegeben, von Vermittlungsschicht bis zur Anwendungsschicht. Anschließend folgte ein kurzer Überblick über betriebssystemspezifische Funktionalitäten des Linux-Kernels, gefolgt von einer Einführung in verteilte Systeme, vor allem im Bereich Kubernetes und seine benötigten Dienste. Als letztes in diesem Abschnitt wurde ein Überblick über die verwandten Arbeiten gegeben, welche eine Abgrenzung dieser Untersuchung zu anderen Untersuchungen aufzeigt, und einen Eindruck über den aktuellen Stand der Forschung geben möchte. Als nächstes wurde die Evaluationsvorbereitung durchgeführt, dabei wurde definiert, wie das zu evaluierende Umfeld aussehen soll und es wurden Alternativen aufgezeigt.

Anschließend wurde der konkrete Aufbau des Kubernetes-*clusters* besprochen und dargestellt. Direkt folgend wurde der Aufbau des Netzwerks besprochen, dabei wurde der Fluss eines Pakets durch das System beschrieben. Als nächstes wurden die unterschiedlichen Pakete analysiert und der *overhead* in der Kommunikation für die unterschiedlichen Webkommunikationsprotokolle definiert.

Nach diesem Abschnitt wurde das Test-Setup definiert, das benötigt wurde, um die Protokolle effizient zu evaluieren. Im letzten Teil der Evaluationsvorbereitung wurde eine Auswahl an Systemen von Webkommunikationsprotokollen getroffen und vorgestellt.

Der Abschnitt Evaluation hat zuerst die Vergleichskriterien für diese Evaluation aufgezeigt, gefolgt von der Definition der Versuchsaufbauten. Anschließend wurden die unterschiedlichen Versuche vorgestellt. Darauf aufbauend wurde eine Erwartung für die

Evaluationen aufgestellt. Diese wurden durchgeführt und innerhalb der Ergebnisse niedergeschrieben. Anschließend wurden die Messwerte interpretiert und eingeordnet.

6 Ausblick

In dieser Arbeit wurde betrachtet, wie sich die aktuellen Versionen von HTTP Implementierungen innerhalb eines Kubernetes *clusters* mit einem *overlay* Netzwerk in einer virtualisierten Umgebung verhalten. Das Ergebnis zeigt, dass viele Implementierungen noch nicht optimal implementiert sind und durch die Varianten an Implementierungen unterschiedlichste Werte ergeben können.

In folgenden Untersuchungen sollte ein Fokus darauf gelegt werden, wie das Verhalten mit echter Hardware aussieht. Durch Optimierungen wie TSO sollte unter Last ein noch signifikant besseres Ergebnis für TCP basierte Protokolle entstehen. Auch das muss in einer weiteren Untersuchung evaluiert werden.

Wie eben schon erwähnt, wurde in dieser Evaluation nie eine komplette Auslastung des Systems untersucht, weshalb auch das ein untersuchungswürdiger Faktor sein könnte.

Ein weiterer Punkt sind Parameter-Studien. Durch die bessere Konfigurierbarkeit und Optimierbarkeit von QUIC im *user space* sollte es möglich sein, dort Gewinne zu erzielen, was im Rahmen einer Forschung untersucht werden sollte.

In diesen hier durchgeführten Versuchen wurde der Idealzustand eines *cluster* getestet. Was ist aber, wenn zwischen zwei Nodes eine höhere Distanz liegt und die Latenz erhöht wird – ist das Verhalten dann immer noch dasselbe?

Ein wichtiger Aspekt, der in einer weiteren Analyse angeschaut werden sollte, ist der Ursprung der aktuell schlechteren Performance von HTTP/3 in dieser Evaluation. Das kann aus unterschiedlichsten Regionen stammen, z.B. aus der *congestion control*, *flow control*, *scheduling* oder anderen Faktoren der Implementierungen sowie des CNI.

In Rahmen dieser Arbeit konnte nur ein Ausschnitt der offenen Fragen angeschaut und

beleuchtet werden. Aufbauend auf diesen Erkenntnis kann nun eine vertiefende Untersuchung stattfinden.

Literaturverzeichnis

[1] KDE e.V. : *Schedules/Plasma 5 - KDE Community Wiki*. https://community.kde.org/Schedules/Plasma_5. – (besucht am 09.05.2023)

[2] Adam Miller, Maxim S.: *RPM Packaging Guide*. https://rpm-packaging-guide.github.io/. – (besucht am 01.05.2023)

[3] Aiortc: *aioquic documentation*. https://aioquic.readthedocs.io/en/latest/. – (besucht am 17.05.2023)

[4] Aiortc: *Aioquic/src/aioquic/quic/packet_builder.py at main AIORTC/AIOQUIC*. https://github.com/aiortc/aioquic/blob/main/src/aioquic/quic/packet_builder.py. – (besucht am 15.05.2023)

[5] Alibaba: *Alibaba/xquic: XQUIC Library released by Alibaba is a cross-platform implementation of QUIC and HTTP/3 protocol*. https://github.com/alibaba/xquic. – (besucht am 20.04.2023)

[6] Amazon Web Services, Inc: *AWS/S2N-TLS: An implementation of the TLS/SSL protocols*. https://github.com/aws/s2n-tls. – (besucht am 20.04.2023)

[7] Amazon Web Services, Inc.: *Firecracker*. https://firecracker-microvm.github.io/. – (besucht am 17.05.2023)

[8] Amazon.com, Inc.: *Verwalteter Kubernetes-Service – häufig gestellte Fragen zu Amazon EKS – Amazon Web Services*. https://aws.amazon.com/de/eks/faqs/. – (besucht am 20.04.2023)

[9] AppArmor: *AppArmor*. https://apparmor.net/. – (besucht am 20.04.2023)

[10] Atea Ataroa Limited.: *Distrowatch.com: Put the fun back into computing. use linux, BSD*. https://distrowatch.com/. – (besucht am 17.05.2023)

[11] Balasubramanian, Praveen ; Huang, Yi ; Olson, Matt: HyStart++: Modified Slow Start for TCP / Internet Engineering Task Force. Internet Engineering Task Force, Februar 2023 (draft-ietf-tcpm-hystartplusplus-14). – Internet-Draft. – URL https://datatracker.ietf.org/doc/draft-ietf-tcpm-hystartplusplus/14/. Work in Progress

[12] Bengel, Günther: *Grundkurs verteilte Systeme: Grundlagen und Praxis des Client-Server und distributed computing*. Springer-Verlag, 2015

[13] Bishop, Mike: *HTTP/3*. RFC 9114. Juni 2022. – URL https://www.rfc-editor.org/info/rfc9114

[14] Biswal, Prasenjeet ; Gnawali, Omprakash: Does quic make the web faster? In: *2016 IEEE Global Communications Conference (GLOBECOM)* IEEE (Veranst.), 2016, S. 1–6

[15] Bittau, Andrea ; Giffin, Daniel B. ; Handley, Mark J. ; Mazieres, David ; Slack, Quinn ; Smith, Eric W.: *Cryptographic Protection of TCP Streams (tcpcrypt)*. RFC 8548. Mai 2019. – URL https://www.rfc-editor.org/info/rfc8548

[16] Blanton, Ethan ; Paxson, Dr. V. ; Allman, Mark: *TCP Congestion Control*. RFC 5681. September 2009. – URL https://www.rfc-editor.org/info/rfc5681

[17] Borman, David ; Braden, Robert T. ; Jacobson, Van ; Scheffenegger, Richard: *TCP Extensions for High Performance*. RFC 7323. September 2014. – URL https://www.rfc-editor.org/info/rfc7323

[18] Burns, Brendan ; Grant, Brian ; Oppenheimer, David ; Brewer, Eric ; Wilkes, John: Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade. In: *Queue* 14 (2016), jan, Nr. 1, S. 70–93. – URL https://doi.org/10.1145/2898442.2898444. – ISSN 1542-7730

[19] Cardwell, Neal ; Cheng, Yuchung ; Yeganeh, Soheil H. ; Swett, Ian ; Jacobson, Van: BBR Congestion Control / Internet Engineering Task Force. Internet Engineering Task Force, März 2022 (draft-cardwell-iccrg-bbr-congestion-control-02). – Internet-Draft. – URL https://datatracker.ietf.org/doc/draft-cardwell-iccrg-bbr-congestion-control/02/. Work in Progress

[20] Carlucci, Gaetano ; De Cicco, Luca ; Mascolo, Saverio: HTTP over UDP: an Experimental Investigation of QUIC. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, 2015, S. 609–614

[21] Carlucci, Gaetano ; De Cicco, Luca ; Mascolo, Saverio: HTTP over UDP: An Experimental Investigation of QUIC. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. New York, NY, USA : Association for Computing Machinery, 2015 (SAC '15), S. 609–614. – URL https://doi.org/10.1145/2695664.2695706. – ISBN 9781450331968

[22] Chen, Jiajia ; Cheng, Weiqing: Analysis of web traffic based on HTTP protocol. In: *2016 24th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2016, S. 1–5

[23] Cheng, Yuchung ; Chu, Jerry ; Radhakrishnan, Sivasankar ; Jain, Arvind: *TCP Fast Open*. RFC 7413. Dezember 2014. – URL https://www.rfc-editor.org/info/rfc7413

[24] Cloud Native Computing Foundation: *cri-o — cri-o.io*. https://cri-o.io/. – (besucht am 03.05.2023)

[25] Cloudflare, Inc: *Cloudflare: Sicheres hosting in Der cloud | cloudflare*. https://www.cloudflare.com/de-de/. – (besucht am 01.05.2023)

[26] Cloudflare, Inc: *Cloudflare/Quiche: implementation of the QUIC transport protocol and HTTP/3*. https://github.com/cloudflare/quiche. – (besucht am 13.05.2023)

[27] Cloudflare, Inc: *Quiche/fuzz at master · cloudflare/quiche*. https://github.com/cloudflare/quiche/tree/master/fuzz. – (besucht am 15.05.2023)

[28] Cloudflare, Inc: *Quiche/nginx at master · cloudflare/quiche*. https://github.com/cloudflare/quiche/tree/master/nginx. – (besucht am 01.05.2023)

[29] Cognixia: *How kubernetes has risen to be the operating system of the cloud?* https://www.cognixia.com/blog/how-kubernetes-has-risen-to-be-the-operating-system-of-the-cloud/. May 2022. – (besucht am 20.04.2023)

[30] Cook, Sarah ; Mathieu, Bertrand ; Truong, Patrick ; Hamchaoui, Isabelle: QUIC: Better for what and for whom? In: *2017 IEEE International Conference on Communications (ICC)*, 2017, S. 1–6

[31] Corbet, Jonathan: *Checksum offloads and protocol ossification*. https://lwn.net/Articles/667059/. – (besucht am 12.04.2023)

[32] curl: *SSLKEYLOGFILE - Everything curl*. https://everything.curl.dev/usingcurl/tls/sslkeylogfile. – (besucht am 08.05.2023)

[33] D2iQ, Inc: *D2iQ Konvoy: Kubernetes Distributions for the Enterprise*. https://d2iq.com/products/konvoy. – (besucht am 20.04.2023)

[34] Datadog: *9 Insights on Real-World Container Use*. https://www.datadoghq.com/container-report/. Nov 2022. – (besucht am 03.05.2023)

[35] Debian: *Debian – News – Debian 11 "bullseye"released — debian.org*. https://www.debian.org/News/2021/20210814. – (besucht am 10.04.2023)

[36] Debian: *Debian New Maintainers' Guide*. https://www.debian.org/doc/manuals/maint-guide/. Oct 2022. – (besucht am 01.05.2023)

[37] Deering, Dr. Steve E. ; Hinden, Bob: *Internet Protocol, Version 6 (IPv6) Specification*. RFC 8200. Juli 2017. – URL https://www.rfc-editor.org/info/rfc8200

[38] Deering, Dr. Steve E. ; Mogul, Jeffrey: *Path MTU discovery*. RFC 1191. November 1990. – URL https://www.rfc-editor.org/info/rfc1191

[39] Dehghani, Zhamak: *How to break a monolith into microservices*. https://martinfowler.com/articles/break-monolith-into-microservices.html. – (besucht am 01.05.2023)

[40] DeNA Co., Ltd.: https://h2o.example.net/. – (besucht am 20.04.2023)

[41] Dierks, Tim ; Rescorla, Eric: *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346. April 2006. – URL https://www.rfc-editor.org/info/rfc4346

[42] Dobies, Jason ; Wood, Joshua: *Kubernetes operators: Automating the container orchestration platform*. O'Reilly Media, 2020

[43] Docker, Inc: *Home — docker.com*. https://www.docker.com/. – (besucht am 03.05.2023)

[44] Docker, Inc: *Dockerfile reference*. https://docs.docker.com/engine/reference/builder/. May 2023. – (besucht am 09.05.2023)

[45] Doornbosch, Peter: *Flupke is a Java HTTP3 implementation that runs on top of Kwik*. https://bitbucket.org/pjtr/flupke/src/master/. – (besucht am 02.05.2023)

[46] Doornbosch, Peter: *Kwik/src/main/java/net/luminis/quic/quicconnectionimpl.java at 06FF824EB04509DE280A548FB192CE19CE5D1D0A PTRD/Kwik*. https://github.com/ptrd/kwik/blob/06ff824eb04509de280a548fb192ce19ce5d1d0a/src/main/java/net/luminis/quic/QuicConnectionImpl.java#L739. – (besucht am 15.05.2023)

[47] Doornbosch, Peter: *PTRD/Kwik: A Quic client, client library and server implementation in Java*. https://github.com/ptrd/kwik. – (besucht am 04.05.2023)

[48] Dotnet: *Add option to decrypt QUIC traffic in debug builds by WFURT · pull request #83001 · dotnet/runtime*. https://github.com/dotnet/runtime/pull/83001. – (besucht am 08.05.2023)

[49] Ducastel, Alexis: *Benchmark results of Kubernetes Network Plugins (CNI) over 10Gbit/S Network (updated: April 2019)*. https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-updated-april-2019-4a9886efe9c4. April 2019. – (besucht am 15.05.2023)

[50] Ducastel, Alexis: *Benchmark results of Kubernetes Network Plugins (CNI) over 10gbit/S network*. https://itnext.io/benchmark-results-of-kubernetes-network-plugins-cni-over-10gbit-s-network-36475925a560. Sep 2020. – (besucht am 15.05.2023)

[51] Duke, Martin: QUIC Version 2 / Internet Engineering Task Force. Internet Engineering Task Force, Dezember 2022 (draft-ietf-quic-v2-10). – Internet-Draft. – URL https://datatracker.ietf.org/doc/draft-ietf-quic-v2/10/. Work in Progress

[52] Duke, Martin: QUIC Version 2 / Internet Engineering Task Force. Internet Engineering Task Force, 2023 (draft-duke-quic-v2-00). – Internet-Draft. – URL https://datatracker.ietf.org/doc/draft-duke-quic-v2/00/. Work in Progress

[53] Eddy, Wesley: *Transmission Control Protocol (TCP)*. RFC 9293. August 2022. – URL https://www.rfc-editor.org/info/rfc9293

[54] Edge, Jake: *A seccomp overview*. https://lwn.net/Articles/656307/. – (besucht am 08.05.2023)

[55] Editor, RFC: *Assigned Numbers: RFC 1700 is Replaced by an On-line Database*. RFC 3232. Januar 2002. – URL https://www.rfc-editor.org/info/rfc3232

[56] eMarketer: *Retail e-commerce sales worldwide from 2014 to 2026 (in billion U.S. dollars) [Graph]*. https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/. July 2022. – (besucht am 03.05.2023)

[57] Endres, Sebastian ; Deutschmann, Jörg ; Hielscher, Kai-Steffen ; German, Reinhard: *Performance of QUIC Implementations Over Geostationary Satellite Links*. 2022

[58] etcd Authors: *etcd*. https://etcd.io/. – (besucht am 17.05.2023)

[59] F5, Inc.: *NGINX*. https://www.nginx.com/. May 2023. – (besucht am 17.05.2023)

[60] Fairhurst, Gorry ; Jones, Tom ; Tüxen, Michael ; Ruengeler, Irene ; Völker, Timo: *Packetization Layer Path MTU Discovery for Datagram Transports*. RFC 8899. September 2020. – URL https://www.rfc-editor.org/info/rfc8899

[61] Fielding, Roy T. ; Nottingham, Mark ; Reschke, Julian: *HTTP Semantics*. RFC 9110. Juni 2022. – URL https://www.rfc-editor.org/info/rfc9110

[62] Fielding, Roy T. ; Nottingham, Mark ; Reschke, Julian: *HTTP/1.1*. RFC 9112. Juni 2022. – URL https://www.rfc-editor.org/info/rfc9112

[63] Flannel-Io: *Flannel/readme.md at master · flannel-IO/flannel*. https://github.com/flannel-io/flannel/blob/master/README.md#flannel. – (besucht am 12.05.2023)

[64] Floyd, Sally: *Limited Slow-Start for TCP with Large Congestion Windows*. RFC 3742. März 2004. – URL https://www.rfc-editor.org/info/rfc3742

[65] Floyd, Sally ; Ramakrishnan, Dr. K. K. ; Black, David L.: *The Addition of Explicit Congestion Notification (ECN) to IP*. RFC 3168. September 2001. – URL https://www.rfc-editor.org/info/rfc3168

[66] Ford, Alan ; Raiciu, Costin ; Handley, Mark J. ; Bonaventure, Olivier ; Paasch, Christoph: *TCP Extensions for Multipath Operation with Multiple Addresses*. RFC 8684. März 2020. – URL https://www.rfc-editor.org/info/rfc8684

[67] Fortune Business Insights: *Server Operating System Market Volume, Share | Analysis, 2030*. https://www.fortunebusinessinsights.com/server-operating-system-market-106601. – (besucht am 17.05.2023)

[68] Free Software Foundation, Inc: *The GNU C Reference Manual*. https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html. – (besucht am 15.05.2023)

[69] Fuller, Vince ; Li, Tony ; Varadhan, Kannan ; Yu, Jessica: *Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy*. RFC 1519. September 1993. – URL https://www.rfc-editor.org/info/rfc1519

[70] Geist, Moritz ; Jaeger, Benedikt: Overview of TCP congestion control algorithms. In: *Network* 11 (2019)

[71] GitHub, Inc.: *About | GitHub*. https://github.com/about. – (besucht am 15.05.2023)

[72] Google LLC: *About google, our culture and company news*. https://about.google/. – (besucht am 17.05.2023)

[73] Google LLC: *BoringSSL is a fork of OpenSSL*. https://boringssl.googlesource.com/boringssl/. – (besucht am 02.05.2023)

[74] Google LLC: *The Go Project*. https://go.dev/project. – (besucht am 15.05.2023)

[75] Google LLC: *Google - About Google, Our Culture & Company News — about.google*. https://about.google/. – (besucht am 25.04.2023)

[76] Google LLC: *Google Kubernetes Engine (GKE) | Google Cloud*. https://cloud.google.com/kubernetes-engine. – (besucht am 03.05.2023)

[77] Google LLC: *Quic, a multiplexed transport over UDP*. https://www.chromium.org/quic/. – (besucht am 01.05.2023)

[78] Gravitational: *Gravitational/Wormhole: Wireguard based Overlay Network CNI plugin for Kubernetes*. https://github.com/gravitational/wormhole/. – (besucht am 03.05.2023)

[79] Gray, Jim: Notes on Data Base Operating Systems. In: *Operating Systems, An Advanced Course*. Berlin, Heidelberg : Springer-Verlag, 1978, S. 393–481. – ISBN 3540087559

[80] Grigorik, Ilya: *High Performance Browser Networking: What every web developer should know about networking and web performance*. Ö'Reilly Media, Inc.", 2013

[81] Gurtov, Andrei ; Henderson, Tom ; Floyd, Sally ; Nishida, Yoshifumi: *The NewReno Modification to TCP's Fast Recovery Algorithm*. RFC 6582. April 2012. – URL https://www.rfc-editor.org/info/rfc6582

[82] Ha, Sangtae ; Rhee, Injong: Taming the elephants: New TCP slow start. In: *Computer Networks* 55 (2011), 06, S. 2092–2110

[83] Hashicorp: *Hashicorp/Nomad: Nomad is an easy-to-use, flexible, and performant workload orchestrator that can deploy a mix of microservice, batch, containerized, and non-containerized applications. Nomad is easy to operate and scale and has native consul and vault integrations*. https://github.com/hashicorp/nomad. – (besucht am 12.04.2023)

[84] HashiCorp: *Nomad by HashiCorp*. https://www.nomadproject.io/. Apr 2023. – (besucht am 20.04.2023)

[85] Heo, Tejun: *Control Group v2*. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html#introduction. – (besucht am 08.05.2023)

[86] Hetzner Online GmbH: *FAQ - Hetzner Docs*. https://docs.hetzner.com/de/cloud/technical-details/faq. – (besucht am 17.05.2023)

[87] Huitema, Christian ; Dickinson, Sara ; Mankin, Allison: *DNS over Dedicated QUIC Connections*. RFC 9250. Mai 2022. – URL https://www.rfc-editor.org/info/rfc9250

[88] hyper: *fast and safe HTTP for the Rust language*. https://hyper.rs/. – (besucht am 09.05.2023)

[89] IETF LLC: *Introduction to the IETF — ietf.org*. https://www.ietf.org/about/introduction/. – (besucht am 12.04.2023)

[90] Internet Corporation for Assigned Names and Numbers: *Internet Assigned Numbers Authority*. https://www.iana.org/assignments/http-methods/http-methods.xhtml. – (besucht am 17.05.2023)

[91] Ionos SE: *Kubernetes FAQ - Products*. https://docs.ionos.com/cloud/managed-services/managed-kubernetes/kubernetes-faq. – (besucht am 25.04.2023)

[92] Iyengar, Jana ; Thomson, Martin: *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. Mai 2021. – URL https://www.rfc-editor.org/info/rfc9000

[93] Jacobs, Ross: *TLS encrypted*. https://tshark.dev/export/export_tls/. – (besucht am 09.05.2023)

[94] Joarder, YA ; Fung, Carol: A Survey on the Security Issues of QUIC. In: *2022 6th Cyber Security in Networking Conference (CSNet)* IEEE (Veranst.), 2022, S. 1–8

[95] Josefsson, Simon: *The Base16, Base32, and Base64 Data Encodings*. RFC 4648. Oktober 2006. – URL https://www.rfc-editor.org/info/rfc4648

[96] Kakhki, Arash M. ; Jero, Samuel ; Choffnes, David ; Nita-Rotaru, Cristina ; Mislove, Alan: Taking a Long Look at QUIC: An Approach for Rigorous Evaluation of Rapidly Evolving Transport Protocols. In: *Proceedings of the 2017 Internet Measurement Conference*. New York, NY, USA : Association for Computing Machinery, 2017 (IMC '17), S. 290–303. – URL https://doi.org/10.1145/3131365.3131368. – ISBN 9781450351188

[97] Kalske, Miika ; Mäkitalo, Niko ; Mikkonen, Tommi: Challenges when moving from monolith to microservice architecture. In: *Current Trends in Web Engineering: ICWE 2017 International Workshops, Liquid Multi-Device Software and EnWoT, practi-O-web, NLPIT, SoWeMine, Rome, Italy, June 5-8, 2017, Revised Selected Papers 17* Springer (Veranst.), 2018, S. 32–47

[98] Kapočius, Narūnas: Overview of kubernetes cni plugins performance. In: *Mokslas–Lietuvos ateitis/Science–Future of Lithuania* 12 (2020)

[99] Kawai, Shintaro ; Miyazawa, Kouto ; Yamaguchi, Saneyasu: Performance evaluation of HTTP/3 QUIC on a network with high latency and high packet loss ratio. In: *IEICE Proceedings Series* 63 (2020), Nr. N2-5

[100] Kelseyhightower: *Kelseyhightower/Kubernetes-the-hard-way: Bootstrap kubernetes the hard way on google cloud platform. no scripts.* https://github.com/kelseyhightower/kubernetes-the-hard-way. – (besucht am 12.04.2023)

[101] Kharat, Prashant K. ; Rege, Aniket ; Goel, Aneesh ; Kulkarni, Muralidhar: QUIC protocol performance in wireless networks. In: *2018 International Conference on Communication and Signal Processing (ICCSP)* IEEE (Veranst.), 2018, S. 0472–0476

[102] Krasic, Charles '. ; Bishop, Mike ; Frindell, Alan: *QPACK: Field Compression for HTTP/3.* RFC 9204. Juni 2022. – URL https://www.rfc-editor.org/info/rfc9204

[103] Lai, Hongli: *Docker and the PID 1 zombie reaping problem.* https://blog.phusion.nl/2015/01/20/docker-and-the-pid-1-zombie-reaping-problem/. Dec 2017. – (besucht am 17.05.2023)

[104] Legay, Damien ; Decan, Alexandre ; Mens, Tom: On Package Freshness in Linux Distributions. In: *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2020, S. 682–686

[105] Liedtke, Jochen: Improving IPC by Kernel Design. In: *SIGOPS Oper. Syst. Rev.* 27 (1993), dec, Nr. 5, S. 175–188. – URL https://doi.org/10.1145/173668.168633. – ISSN 0163-5980

[106] The Linux Foundation: *IPVS Software - Advanced Layer-4 Switching — linuxvirtualserver.org.* http://www.linuxvirtualserver.org/software/ipvs.html. – (besucht am 09.05.2023)

[107] The Linux Foundation: *Namespaces - linux manual page*. https://man7.org/linux/man-pages/man7/namespaces.7.html. – (besucht am 20.04.2023)

[108] The Linux Foundation: *The Linux Kernel Archives — kernel.org*. https://www.kernel.org/. – (besucht am 25.04.2023)

[109] LiteSpeed Technologies Inc: *Litespeedtech/LSQUIC: Litespeed Quic and HTTP/3 library*. https://github.com/litespeedtech/lsquic. – (besucht am 15.05.2023)

[110] Liu, Yanmei ; Ma, Yunfei ; Coninck, Quentin D. ; Bonaventure, Olivier ; Huitema, Christian ; Kühlewind, Mirja: Multipath Extension for QUIC / Internet Engineering Task Force. Internet Engineering Task Force, März 2023 (draft-ietf-quic-multipath-04). – Internet-Draft. – URL https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/04/. Work in Progress

[111] Love, Robert: *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. – ISBN 0672329468

[112] Mahalingam, Mallik ; Dutt, Dinesh ; Duda, Kenneth ; Agarwal, Puneet ; Kreeger, Larry ; Sridhar, T. ; Bursell, Mike ; Wright, Chris: *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. RFC 7348. August 2014. – URL https://www.rfc-editor.org/info/rfc7348

[113] Marx, Robin ; De Decker, Tom ; Quax, Peter ; Lamotte, Wim: Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC. In: *WEBIST*, 2019, S. 130–143

[114] Masse, Mark: *REST API design rulebook: designing consistent RESTful web service interfaces*. Ö'Reilly Media, Inc.", 2011

[115] Mesosphere, Inc.: *Marathon: A container orchestration platform for Mesos and DC/OS*. https://mesosphere.github.io/marathon/. – (besucht am 09.05.2023)

[116] Microsoft Corporation: *About Microsoft | Mission and Vision | Microsoft*. https://www.microsoft.com/en-us/about. – (besucht am 15.05.2023)

[117] Microsoft Corporation: *Managed kubernetes service (AKS): Microsoft azure*. https://azure.microsoft.com/en-us/products/kubernetes-service. – (besucht am 03.05.2023)

[118] Microsoft Corporation: *Microsoft Server*. https://www.microsoft.com/de-de/windows-server. – (besucht am 17.05.2023)

[119] Microsoft Corporation: *.NET | Build. Test. Deploy.* https://dotnet.microsoft.com/en-us/. – (besucht am 15.05.2023)

[120] Microsoft Corporation: *Verwenden von HTTP/3 mit dem asp.net core-kestrel-webserver*. https://learn.microsoft.com/de-de/aspnet/core/fundamentals/servers/kestrel/http3?view=aspnetcore-8.0. – (besucht am 08.05.2023)

[121] Mikac, M ; Horvatić, M ; Mikac, V: NETWORKING CASE STUDY IN STEM EDUCATION-IP FRAGMENTATION. In: *INTED2020 Proceedings* IATED (Veranst.), 2020, S. 1068–1077

[122] Mockapetris, P.: *Domain names - implementation and specification*. RFC 1035. November 1987. – URL https://www.rfc-editor.org/info/rfc1035

[123] Moulay, Mohamed ; Muñoz, Fernando D. ; Mancuso, Vincenzo: On the Experimental Assessment of QUIC and Congestion Control Schemes in Cellular Networks. In: *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*, 2021, S. 1–8

[124] Mozilla: *Mozilla/neqo*. https://github.com/mozilla/neqo. – (besucht am 20.04.2023)

[125] Nabla Containers: *Nabla Containers*. https://nabla-containers.github.io/. – (besucht am 03.05.2023)

[126] Nepomuceno, Késsia ; Oliveira, Igor Nogueira d. ; Aschoff, Rafael R. ; Bezerra, Daniel ; Ito, Maria S. ; Melo, Wesley ; Sadok, Djamel ; Szabó, Géza: QUIC and TCP: A Performance Evaluation. In: *2018 IEEE Symposium on Computers and Communications (ISCC)*, 2018, S. 00045–00051

[127] Netcraft Ltd: *Web server survey | netcraft news*. https://news.netcraft.com/archives/category/web-server-survey. – (besucht am 17.05.2023)

[128] NETFILTER'S: *The netfilter.org iptables project.* https://www.netfilter.org/projects/iptables/index.html. – (besucht am 08.05.2023)

[129] NGTCP2: *NGTCP2/NGTCP2: NGTCP2 project is an effort to implement IETF QUIC protocol.* https://github.com/ngtcp2/ngtcp2. – (besucht am 20.04.2023)

[130] NOVIANTI, Siska ; BASUKI, Achmad: The Performance Analysis of Container Networking Interface Plugins in Kubernetes. In: *Proceedings of the 6th International Conference on Sustainable Information Engineering and Technology.* New York, NY, USA : Association for Computing Machinery, 2021 (SIET '21), S. 231–234. – URL https://doi.org/10.1145/3479645.3479700. – ISBN 9781450384070

[131] NVIDIA CORPORATION: *VXLAN Hardware Stateless Offloads - NVIDIA Networking Docs.* https://docs.nvidia.com/networking/display/MLNXOFEDv5432723/VXLAN%20Hardware%20Stateless%20Offloads. – (besucht am 09.05.2023)

[132] OPEN CONTAINER INITIATIVE: *GitHub - opencontainers/runc: CLI tool for spawning and running containers according to the OCI specification — github.com.* https://github.com/opencontainers/runc. – (besucht am 03.05.2023)

[133] OPENINFRA FOUNDATION: *Kata Containers - Open Source Container Runtime Software.* https://katacontainers.io/. – (besucht am 03.05.2023)

[134] ORACLE CORPORATION: *MySQL.* https://www.mysql.com/de/. – (besucht am 17.05.2023)

[135] ORACLE CORPORATION: *Oracle VM VirtualBox — virtualbox.org.* https://www.virtualbox.org/. – (besucht am 10.04.2023)

[136] ORACLE CORPORATION: *What is Java technology and why do I need it?* https://www.java.com/en/download/help/whatis_java.html. – (besucht am 15.05.2023)

[137] PARK, Youngki ; YANG, Hyunsik ; KIM, Younghan: Performance analysis of cni (container networking interface) based container network. In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)* IEEE (Veranst.), 2018, S. 248–250

[138] Pat Meenan, Paul Calvano und Barry P.: *HTTP Archive: Top 1,000,000: Page Weight*. https://httparchive.org/reports/page-weight?lens=top1m&start=2018_02_01&end=latest&view=list. May 2023. – (besucht am 19.05.2023)

[139] Pauly, Tommy ; Kinnear, Eric ; Schinazi, David: *An Unreliable Datagram Extension to QUIC*. RFC 9221. März 2022. – URL https://www.rfc-editor.org/info/rfc9221

[140] Peon, Roberto ; Ruellan, Herve: *HPACK: Header Compression for HTTP/2*. RFC 7541. Mai 2015. – URL https://www.rfc-editor.org/info/rfc7541

[141] Petit-Huguenin, Marc ; Salgueiro, Gonzalo: *Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)*. RFC 7983. September 2016. – URL https://www.rfc-editor.org/info/rfc7983

[142] Piraux, Maxime ; De Coninck, Quentin ; Bonaventure, Olivier: Observing the evolution of QUIC implementations. In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2018, S. 8–14

[143] Postel, J.: User Datagram Protocol / RFC Editor. RFC Editor, August 1980 (6). – STD. – URL http://www.rfc-editor.org/rfc/rfc768.txt. http://www.rfc-editor.org/rfc/rfc768.txt. – ISSN 2070-1721

[144] Postel, J.: *Assigned numbers*. RFC 790. September 1981. – URL https://www.rfc-editor.org/info/rfc790

[145] Postel, Jon: Internet Protocol / RFC Editor. RFC Editor, September 1981 (5). – STD. – URL http://www.rfc-editor.org/rfc/rfc791.txt. http://www.rfc-editor.org/rfc/rfc791.txt. – ISSN 2070-1721

[146] private-octopus: *Private-octopus/picoquic: Minimal implementation of the Quic Protocol*. https://github.com/private-octopus/picoquic. – (besucht am 15.05.2023)

[147] projectcalico: *vxlan_resolver.go at master calico*. https://github.com/projectcalico/calico/blob/master/felix/calc/vxlan_resolver.go#L31. – (besucht am 25.04.2023)

[148] Python Software Foundation: *About Python | Python.org*. https://www.python.org/about/. – (besucht am 15.05.2023)

[149] Qi, Shixiong ; Kulkarni, Sameer G. ; Ramakrishnan, KK: Understanding container network interface plugins: design considerations and performance. In: *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN* IEEE (Veranst.), 2020, S. 1–6

[150] Quic-Go: *HTTP/3 much slower than HTTP/1.1*. https://github.com/quic-go/quic-go/issues/3729. – (besucht am 08.05.2023)

[151] Quic-Go: *New API in go should offer faster UDP*. https://github.com/quic-go/quic-go/issues/3563. – (besucht am 08.05.2023)

[152] Quic-Go: *Quic-go/example at master · Quic-go/quic-go*. https://github.com/quic-go/quic-go/tree/master/example. – (besucht am 08.05.2023)

[153] Quic-Go: *Quic-go/quic-go: A quic implementation in pure go*. https://github.com/quic-go/quic-go. – (besucht am 15.05.2023)

[154] Quicwg: *Quic WG implementations listing*. https://github.com/quicwg/base-drafts/wiki/Implementations. – (besucht am 08.05.2023)

[155] Red Hat: *Quay*. https://quay.io/. – (besucht am 09.05.2023)

[156] Red Hat: *Das Red Hat Enterprise Linux Betriebssystem*. https://www.redhat.com/de/technologies/linux-platforms/enterprise-linux. – (besucht am 17.05.2023)

[157] Red Hat: *Red Hat OpenShift enterprise Kubernetes container platform — redhat.com*. https://www.redhat.com/en/technologies/cloud-computing/openshift. – (besucht am 03.05.2023)

[158] Red Hat: *SELinux/Understanding*. https://fedoraproject.org/wiki/SELinux/Understanding. – (besucht am 08.05.2023)

[159] Rekhter, Yakov ; Hares, Susan ; Li, Tony: *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. Januar 2006. – URL https://www.rfc-editor.org/info/rfc4271

[160] Rescorla, Eric: *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. August 2018. – URL https://www.rfc-editor.org/info/rfc8446

[161] Rescorla, Eric ; Dierks, Tim: *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. August 2008. – URL https://www.rfc-editor.org/info/rfc5246

[162] Rhee, Injong ; Xu, Lisong ; Ha, Sangtae ; Zimmermann, Alexander ; Eggert, Lars ; Scheffenegger, Richard: *CUBIC for Fast Long-Distance Networks*. RFC 8312. Februar 2018. – URL https://www.rfc-editor.org/info/rfc8312

[163] Ruamviboonsuk, Vaspol: *HTTP: 2022: The web almanac by HTTP archive*. https://almanac.httparchive.org/en/2022/http. Apr 2023. – (besucht am 03.05.2023)

[164] Rust-Lang: *Rust-lang/libc: Raw bindings to platform apis for rust*. https://github.com/rust-lang/libc. – (besucht am 09.05.2023)

[165] Rust Team: *Rust Programming Language*. https://www.rust-lang.org/. – (besucht am 01.05.2023)

[166] Rustls: *Rustls/rustls: A modern TLS library in rust*. https://github.com/rustls/rustls. – (besucht am 09.05.2023)

[167] Shreedhar, Tanya ; Panda, Rohit ; Podanev, Sergey ; Bajpai, Vaibhav: Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. In: *IEEE Transactions on Network and Service Management* 19 (2021), Nr. 2, S. 1366–1381

[168] Simpson, J: *20 impressive API economy statistics: Nordic apis*. https://nordicapis.com/20-impressive-api-economy-statistics/. May 2022. – (besucht am 01.05.2023)

[169] Simpson, William A.: *TCP Cookie Transactions (TCPCT)*. RFC 6013. Januar 2011. – URL https://www.rfc-editor.org/info/rfc6013

[170] Singh, Harinderjit: *Inspecting and understanding k8s service network*. https://itnext.io/inspecting-and-understanding-service-network-dfd8c16ff2c5. May 2023. – (besucht am 17.05.2023)

[171] Social, DataReportal Meltwater We A.: *Global internet penetration rate as of January 2023, by region [Graph]*. https://www.statista.com/statistics/269329/penetration-rate-of-the-internet-by-region/. January 2023. – (besucht am 03.05.2023)

[172] Stack Holdings GmbH: *Caddy 2 - the ultimate server with automatic https*. https://caddyserver.com/. – (besucht am 01.05.2023)

[173] Statista: *Webserver - marktanteile mai 2023*. https://de.statista.com/statistik/daten/studie/181588/umfrage/marktanteil-der-meistgenutzten-webserver/. Jun 2023. – (besucht am 15.06.2023)

[174] Stenberg, Daniel: *Ossification - HTTP/3 explained*. https://http3-explained.haxx.se/en/why-quic/why-ossification. – (besucht am 12.04.2023)

[175] strace Entwickler: *strace*. https://strace.io/. – URL https://strace.io/. – (besucht am 17.05.2023)

[176] SUSE Software Solutions Germany GmbH: *What is Rancher? | Rancher Manager — ranchermanager.docs.rancher.com*. https://ranchermanager.docs.rancher.com/. – (besucht am 03.05.2023)

[177] Tanenbaum, Andrew S. ; Wetherall, David J.: *Computer Networks*. 5th. USA : Prentice Hall Press, 2010. – ISBN 0132126958

[178] The Apache Software Foundation: *The Apache HTTP Server Project*. https://httpd.apache.org/. – (besucht am 12.04.2023)

[179] The Apache Software Foundation: *Apache/Mesos: Apache mesos*. https://github.com/apache/mesos. – (besucht am 12.04.2023)

[180] The Cilium Authors: *Kubernetes Without kube-proxy Cilium 1.10.20 documentation*. https://docs.cilium.io/en/v1.10/gettingstarted/kubeproxy-free/. – (besucht am 09.05.2023)

[181] The Cilium Authors: *Linux native, API-aware networking and security for containers*. https://cilium.io/. – (besucht am 09.05.2023)

[182] The GNOME Project: *ReleasePlanning - GNOME Wiki!* https://wiki.gnome.org/ReleasePlanning. – (besucht am 25.04.2023)

[183] The GraphQL Foundation: *GraphQL*. https://graphql.org. 2015. – URL https://graphql.org/. – (besucht am 01.05.2023)

[184] The Kubernetes Steering Committee: *Kubernetes Annual Report 2021*. https://www.cncf.io/reports/kubernetes-annual-report-2021/. Feb 2023. – (besucht am 09.05.2023)

[185] The Linux Foundation: *Borg: The Predecessor to Kubernetes — kubernetes.io*. https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/. – (besucht am 25.04.2023)

[186] The Linux Foundation: *Cloud Native Computing Foundation*. https://www.cncf.io/projects/. – (besucht am 17.05.2023)

[187] The Linux Foundation: *The Container Network Interface*. https://www.cni.dev/. – (besucht am 25.04.2023)

[188] The Linux Foundation: *GitHub - kubernetes/kubernetes: Production-Grade Container Scheduling and Management — github.com*. https://github.com/kubernetes/kubernetes. – (besucht am 09.05.2023)

[189] The Linux Foundation: *Harbor 2.7 documentation*. https://goharbor.io/docs/2.7.0/. – (besucht am 10.04.2023)

[190] The Linux Foundation: *Kubernetes/Kubernetes: Production-Grade Container Scheduling and management*

[191] The Linux Foundation: *The package manager for Kubernetes*. https://helm.sh/. – (besucht am 25.04.2023)

[192] The Linux Foundation: *Production-grade container orchestration*. https://kubernetes.io/. – (besucht am 08.05.2023)

[193] The Linux Foundation: *Sysctl(8) - linux man page*. https://linux.die.net/man/8/sysctl. – (besucht am 17.05.2023)

[194] The Linux Foundation: *Who we are*. https://www.cncf.io/about/who-we-are/. – (besucht am 08.05.2023)

[195] The Linux Foundation: *New slashdata report: 5.6 million developers use Kubernetes, an increase of 67% over one year*. https://www.cncf.io/blog/2021/12/20/new-slashdata-report-5-6-million-developers-use-kubernetes-an-increase-of-67-over-one-year/. 2021. – (besucht am 20.04.2023)

[196] The Linux Foundation: *Kubernetes create cluster with kubeadm*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. Nov 2022. – (besucht am 20.04.2023)

[197] The Linux Foundation: *Kubernetes production environment topologies*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/. Jan 2022. – (besucht am 17.05.2023)

[198] The Linux Foundation: *Kubernetes release cycle*. https://kubernetes.io/releases/release/. Jul 2022. – (besucht am 04.05.2023)

[199] The Linux Foundation: *CNCF Annual Survey 2022*. https://www.cncf.io/reports/cncf-annual-survey-2022/. Jan 2023. – (besucht am 09.05.2023)

[200] The Linux Foundation: *Kubernetes concepts configmap*. https://kubernetes.io/docs/concepts/configuration/configmap/. Apr 2023. – (besucht am 03.05.2023)

[201] The Linux Foundation: *Kubernetes concepts ingress*. https://kubernetes.io/docs/concepts/services-networking/ingress/. Feb 2023. – (besucht am 03.05.2023)

[202] The Linux Foundation: *Kubernetes install with kubeadm*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/. May 2023. – (besucht am 17.05.2023)

[203] The Linux Foundation: *Kubernetes overview components*. https://kubernetes.io/docs/concepts/overview/components. May 2023. – (besucht am 17.05.2023)

[204] The Linux Foundation: *Kubernetes using sysctls*. https://kubernetes.io/docs/tasks/administer-cluster/sysctl-cluster/. Apr 2023. – (besucht am 09.05.2023)

[205] The Linux Foundation: *Windows containers in Kubernetes*. https://kubernetes.io/docs/concepts/windows/intro/. Mar 2023. – (besucht am 17.05.2023)

[206] The PHP Group: *PHP: Was ist PHP? - Manual*. https://www.php.net/manual/de/intro-whatis.php. – (besucht am 01.05.2023)

[207] The Tcpdump Group: *Home | TCPDUMP AND LIBPCAP — tcpdump.org*. https://www.tcpdump.org/. – (besucht am 10.04.2023)

[208] Thomson, Martin ; Benfield, Cory: *HTTP/2*. RFC 9113. Juni 2022. – URL https://www.rfc-editor.org/info/rfc9113

[209] Thomson, Martin ; Nottingham, Mark ; Tarreau, Willy: *Using Early Data in HTTP*. RFC 8470. September 2018. – URL https://www.rfc-editor.org/info/rfc8470

[210] Thomson, Martin ; Turner, Sean: *Using TLS to Secure QUIC*. RFC 9001. Mai 2021. – URL https://www.rfc-editor.org/info/rfc9001

[211] Thönes, Johannes: Microservices. In: *IEEE Software* 32 (2015), Nr. 1, S. 116–116

[212] Tigera, Inc.: *Calico Documentation FelixConfig*. https://docs.tigera.io/calico/latest/reference/resources/felixconfig. – (besucht am 25.04.2023)

[213] Tigera, Inc.: *Calico Documentation VXLAN*. https://docs.tigera.io/calico/latest/networking/configuring/vxlan-ipip#configure-vxlan-encapsulation-for-all-inter-workload-traffic. – (besucht am 25.04.2023)

[214] Tigera, Inc.: *Kubernetes | Calico Documentation*. https://docs.tigera.io/calico/latest/getting-started/kubernetes/. – (besucht am 09.05.2023)

[215] Tigera, Inc.: *Overlay networking | Calico Documentation — docs.tigera.io*. https://docs.tigera.io/calico/latest/networking/configuring/vxlan-ipip. – (besucht am 12.04.2023)

[216] Tigera, Inc: *Tigera*. https://www.tigera.io/project-calico/. May 2023. – (besucht am 01.05.2023)

[217] Tikhonov, Dmitri: *Improve performance with DPLPMTUD litespeed blog*. https://blog.litespeedtech.com/2020/10/19/improve-performance-with-dplpmtud. Jan 2021. – (besucht am 15.05.2023)

[218] Traefik.io: *Traefik*. https://doc.traefik.io/traefik/. – (besucht am 01.05.2023)

[219] Tulka, Tomas: *Building container images without a dockerfile by Tomas Tulka*. https://blog.ttulka.com/building-container-images-without-dockerfile/. – (besucht am 15.05.2023)

[220] Tyunyayev, Nikita ; Piraux, Maxime ; Bonaventure, Olivier ; Barbette, Tom: A high-speed QUIC implementation. In: *Proceedings of the 3rd International CoNEXT Student Workshop*, 2022, S. 20–22

[221] Van Steen, Maarten ; Tanenbaum, Andrew S.: *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017

[222] Velichko, Ivan: *Containers vs. pods - taking a deeper look*. https://iximiuz.com/en/posts/containers-vs-pods/. Oct 2021. – (besucht am 20.04.2023)

[223] Verma, Abhishek ; Pedrosa, Luis ; Korupolu, Madhukar R. ; Oppenheimer, David ; Tune, Eric ; Wilkes, John: *Large-scale cluster management at Google with Borg*. 2015

[224] Wang, George: *Lsquic v4.0*. https://blog.litespeedtech.com/2023/03/08/lsquic-v4-0/. May 2023. – (besucht am 12.05.2023)

[225] Wang, Peng ; Bianco, Carmine ; Riihijärvi, Janne ; Petrova, Marina: Implementation and Performance Evaluation of the QUIC Protocol in Linux Kernel. In: *Implementation and Performance Evaluation of the QUIC Protocol in Linux Kernel*, 10 2018, S. 227–234

[226] Wang, Yue ; Zhao, Kanglian ; Li, Wenfeng ; Fraire, Juan ; Sun, Zhili ; Fang, Yuan: Performance Evaluation of QUIC with BBR in Satellite Internet. In: *2018 6th IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE)*, 2018, S. 195–199

[227] WEAVEWORKS: *Weave net*. https://www.weave.works/oss/net/. – (besucht am 25.04.2023)

[228] Woods, Natasha: *Cloud Native Computing Foundation*. https://www.cncf.io/announcements/2017/03/29/containerd-joins-cloud-native-computing-foundation/. Sep 2020. – (besucht am 17.05.2023)

[229] Xu, Changqiao ; Zhao, Jia ; Muntean, Gabriel-Miro: Congestion Control Design for Multipath Transport Protocols: A Survey. In: *IEEE Communications Surveys & Tutorials* 18 (2016), Nr. 4, S. 2948–2969

[230] Yang, Y.R. ; Kim, Nin S. ; Lam, S.S.: Transient behaviors of TCP-friendly congestion control protocols. In: *Proceedings IEEE INFOCOM 2001. Conference on*

Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213) Bd. 3, 2001, S. 1716–1725 vol.3

[231] Yehuda Katz, Gabe Sullice und Jeldrik H.: *JSON:API*. https://jsonapi.org/. – (besucht am 01.05.2023)

[232] Yu, Alexander ; Benson, Theophilus A.: Dissecting Performance of Production QUIC. In: *Proceedings of the Web Conference 2021*. New York, NY, USA : Association for Computing Machinery, 2021 (WWW '21), S. 1157–1168. – URL https://doi.org/10.1145/3442381.3450103. – ISBN 9781450383127

[233] Yu, Yajun ; Xu, Mingwei ; Yang, Yuan: When QUIC meets TCP: An experimental study. In: *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, 2017, S. 1–8

[234] Zeng, Hao ; Wang, Baosheng ; Deng, Wenping ; Zhang, Weiqi: Measurement and evaluation for docker container networking. In: *2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)* IEEE (Veranst.), 2017, S. 105–108

[235] Zhang, Min ; Dusi, Maurizio ; John, Wolfgang ; Chen, Changjia: Analysis of UDP Traffic Usage on Internet Backbone Links. In: *2009 Ninth Annual International Symposium on Applications and the Internet*, 2009, S. 280–281

[236] Zimmermann, H.: OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. In: *IEEE Transactions on Communications* 28 (1980), Nr. 4, S. 425–432

A Anhang

A.1 Weitere Messwerte

A.1.1 Versuch 1

Abbildung A.1

Abbildung A.2

Abbildung A.3

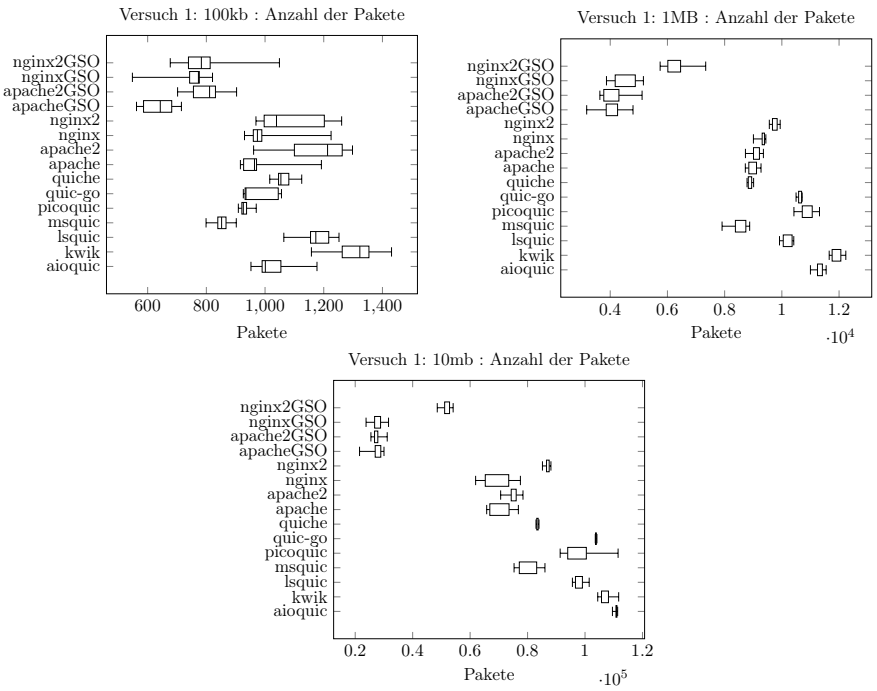

Abbildung A.1: Ergebnisse zu Versuch 1: Versendete Pakete

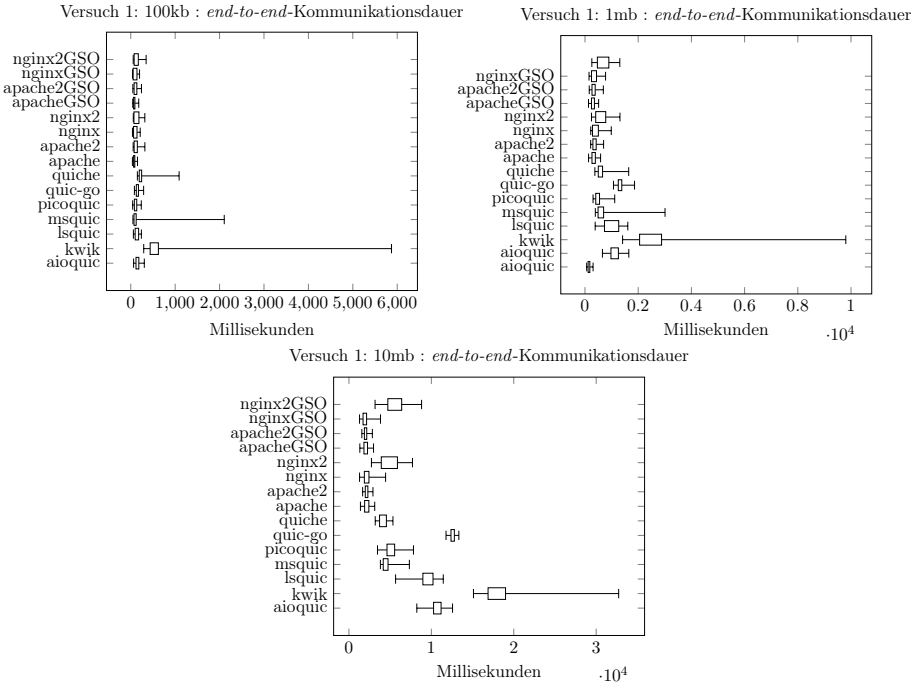

Abbildung A.2: Ergebnisse zu Versuch 1: *end-to-end*-Kommunikationsdauer

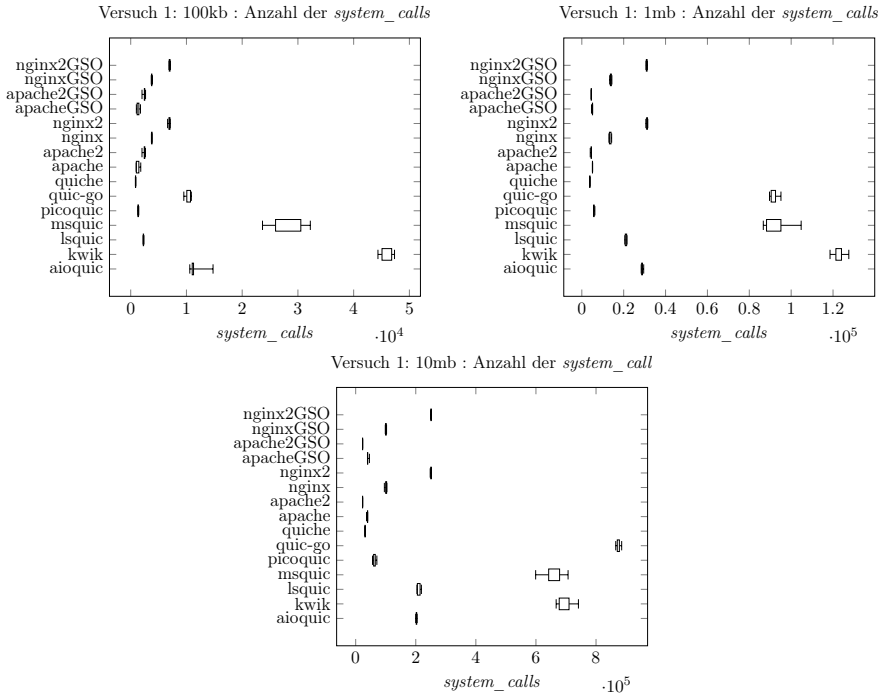

Abbildung A.3: Ergebnisse zu Versuch 1: Anzahl der *system_ calls*

A.1.2 Versuch 3

Abbildung A.4

Abbildung A.5

Abbildung A.6

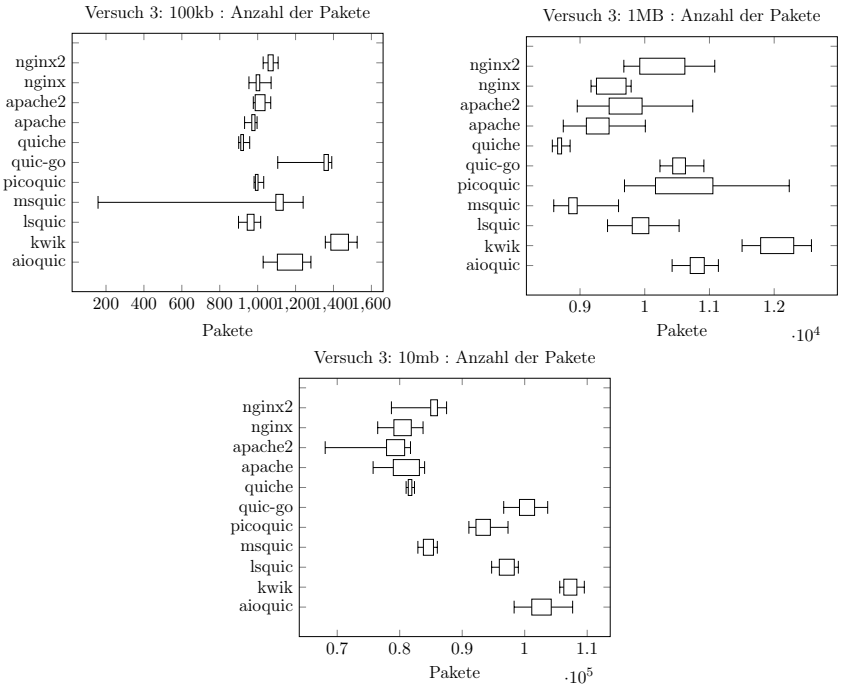

Abbildung A.4: Ergebnisse zu Versuch 3: Versendete Pakete

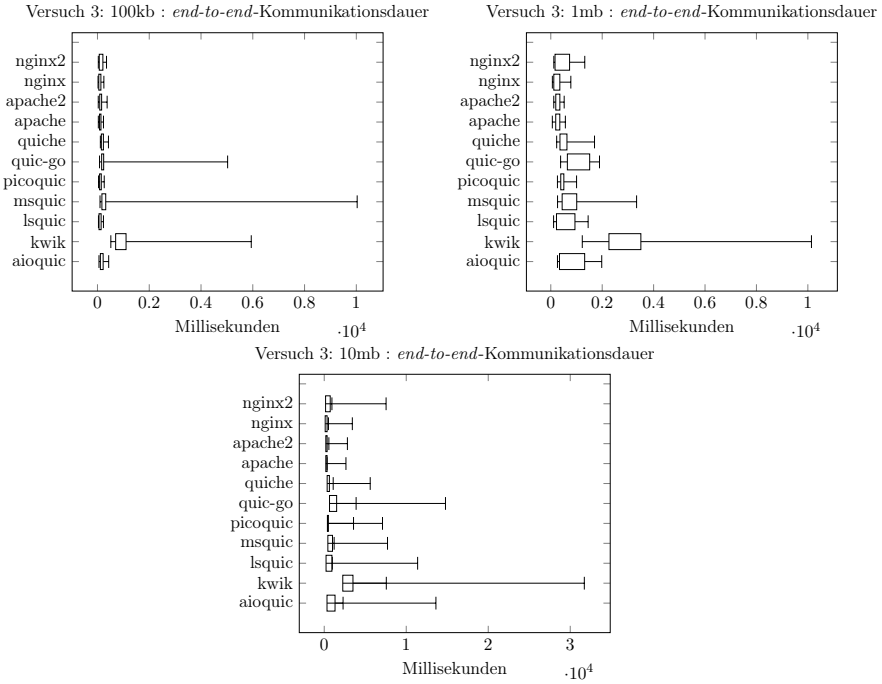

Abbildung A.5: Ergebnisse zu Versuch 3: *end-to-end*-Kommunikationsdauer

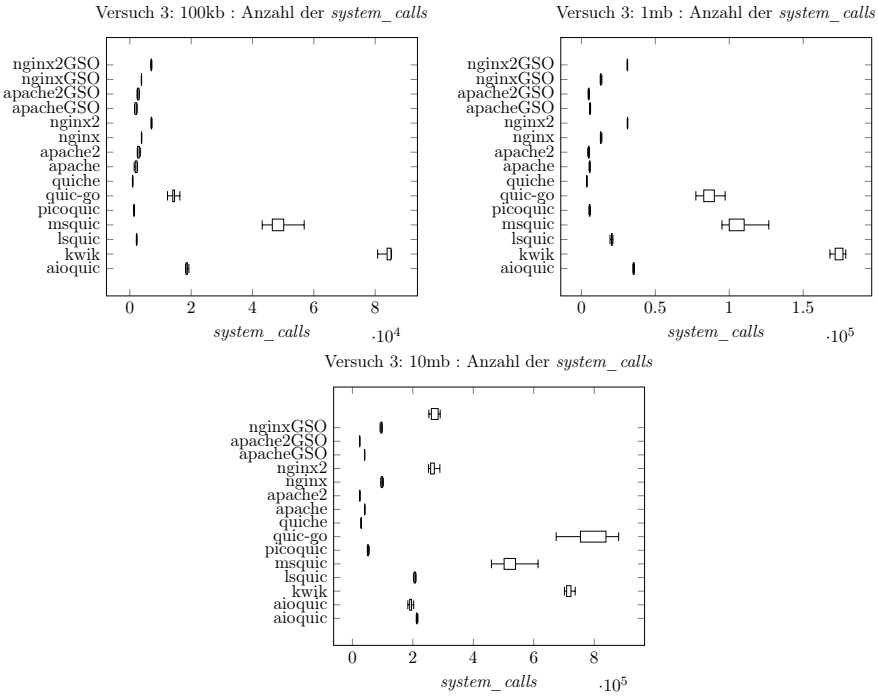

Abbildung A.6: Ergebnisse zu Versuch 3: Anzahl der *system_ calls*

Glossar

5-Tuples Dieses besteht aus *Source IP address*, *Destination IP address*, *Protocol*, *Source port*, *Destination port* und wird als einzigartige ID verwendet, um eine Verbindung zu einer anderen Verbindung abgrenzen zu können [177].

DEB ist wie RPM ein Paketformat, welches für Debian entwickelt worden ist. Es ist genau wie RPM eine Softwareauslieferungslösung [36].

Maximum Transmission Unit MTU steht für *Maximum Transmission Unit* und ist die maximale Größe eines Pakets, welches über eine Netzwerkschnittstelle versendet werden kann [177].

RPM RPM Package Manager, oder früher Red Hat Package Manager, ist ein Paketverwaltungssystem. Es bietet dabei ein Paketformat und alle dazugehörigen Programme an, um RPM-Pakete zu erstellen und zu verwalten. Generell ist es wie DEB auch eine Softwareauslieferungslösung [2].

SSLKEYLOGFILE Durch die Konfiguration mit einem SSLKEYLOGFILE ist es möglich, die SSL Keys in einer Datei zu speichern, um anschließend den aufgezeichneten Datenverkehr zu entschlüsseln [93].

sysctl Ist ein Tool, um Kernel-Parameter von Linux-Systemen zur Laufzeit bearbeiten und lesen zu können [193].

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original