Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marvin Butkereit

Entwicklungsprozess für eine diskrete Eventsimulation am Beispiel eines QUIC Verbindungsaufbaus

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marvin Butkereit

Entwicklungsprozess für eine diskrete Eventsimulation am Beispiel eines QUIC Verbindungsaufbaus

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

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

Eingereicht am: 08. Mai 2019

Marvin Butkereit

Thema der Arbeit
Entwicklungsprozess für eine diskrete Eventsimulation am Beispiel eines QUIC Verbindungsaufbaus

Stichworte
QUIC-Protokoll, OMNeT++, Simulation, INET Framework, Transportprotokoll

Kurzzusammenfassung
Internetdienste sind ein fester und alltäglicher Bestandteil der modernen Gesellschaft und nicht mehr wegzudenken. Beim Kauf von Bekleidung, elektronischen Geräten sowie vielen weiteren Produkten verdrängen sie den herkömmlichen Einzelhandel und sind marktbeherrschend. Ein großer Teil der Interaktionen zwischen Personen und zwischen Organisationen jeder Art sind vollständig abhängig von einem gut funktionierenden Netz. Ansprüche und Erwartungen von Nutzern und Konsumenten an die Leistung und Performanz von Internetauftritten steigen permanent - z.B. wartet ein Kunde nicht mehrere Augenblicke, bis eine Webseite geladen und benutzbar ist.

Eine Technologie, mit der diese gestiegenen, modernen Anforderungen an die Kommunikation erfüllt werden können, ist Quick UDP Internet Connections, kurz QUIC, ein neues Transportprotokoll, welches auf UDP (User Datagram Protocol) aufsetzt.

Diese Arbeit beschäftigt sich mit der Implementation des Verbindungsaufbaus von QUIC, mit dem Ziel, einen Vergleich der Delay-Sensitivität zwischen den Verbindungsaufbauten verschiedener verbindungsorientierter Protokolle herzustellen.

Als Basis wird das zeitdiskrete Eventsimulations-Framework OMNeT++ verwendet. Dieses ermöglicht es, reproduzierbare Versuche aufzubauen und durchzuführen. Basierend auf der Bibliothek INET Framework wird ein QUIC-Simulationsmodell erstellt, mit dem die Delay-Sensitivät des Verbindungsaufbaus untersucht werden kann.

Marvin Butkereit

Title of the paper
Development process for a discrete event simulation using the example of a QUIC connection setup

Keywords
QUIC protocol, OMNeT++, Simulation, INET Framework, transport protocol

Abstract
Internet services are an integral and everyday part of modern society and it is hard to imagine life without them. When it comes to buying clothing, electronic devices and many other products, the internet is displacing traditional retail stores and dominating the market. Much of the interaction between individuals and between organisations of all kinds is completely dependent on a well-functioning network. The demands and expectations of users and consumers on the performance of internet presences are constantly increasing - e.g. a customer does not wait several moments until a website is loaded and usable.

One technology with which these increased, modern communication requirements can be met is Quick UDP Internet Connections, or QUIC for short, a new transport protocol based on UDP (User Datagram Protocol).

This thesis deals with the implementation of the connection setup of QUIC with the aim of comparing the delay sensitivity between the connection setup of different connection-oriented protocols.

As basis the time discrete event simulation framework OMNeT++ is used. This enables reproducible experiments to be set up and carried out. Based on the INET Framework library, a QUIC simulation model is created to investigate the delay sensitivity of the connection setup.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

1.1. Einführung

Internetdienste sind ein fester und alltäglicher Bestandteil der modernen Gesellschaft und nicht mehr wegzudenken. Beim Kauf von Bekleidung, elektronischen Geräten sowie vielen weiteren Produkten verdrängen sie den herkömmlichen Einzelhandel und sind marktbeherrschend [78]. Ein großer Teil der Interaktionen zwischen Personen und zwischen Organisationen jeder Art sind vollständig abhängig von einem gut funktionierenden Netz. Ansprüche und Erwartungen von Nutzern und Konsumenten an die Leistung und Performanz von Internetauftritten steigen permanent - z.B. wartet ein Kunde nicht mehrere Augenblicke, bis eine Webseite geladen und benutzbar ist [1].

Diese Aussage wird durch die stetige Zunahme des Umsatzes im E-Commerce-Bereich untermauert. Seit 2014 ist der Umsatz innerhalb von drei Jahren von 1336 Milliarden US-Dollar auf 2304 Milliarden US-Dollar im Jahre 2017 gestiegen [21]. Durch dieses Wachstum entsteht ein hoher Konkurrenzdruck unter den Marktteilnehmern, mit der Folge, dass Betreiber von E-Commerce-Plattformen hohe und weiter steigende Anstrengungen unternehmen, um ihren Konsumenten ein möglichst gutes Erlebnis bieten zu können [1].

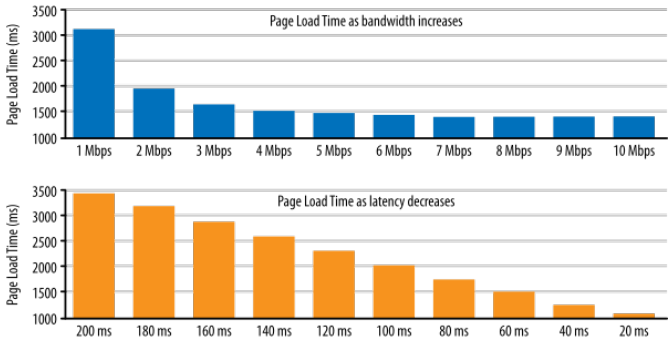
Ein maßgeblicher Faktor für die Zufriedenheit von Kunden eines Webauftritts ist die Seitenladezeit. Einige größere Anbieter wie z.B. Amazon schätzen, dass eine Verschlechterung der Seitenladezeiten von 100 Millisekunden einen Umsatzeinbruch von 1% verursacht [24]. Bei einem Umsatz von 91,4 Milliarden US-Dollar, wie es bei Amazon 2016 für Online-Käufe der Fall war [18], bedeutete dies einen theoretischen Schaden von 914 Millionen US-Dollar.

Die Zeit, die es benötigt, eine Internetseite herunterzuladen, kann durch zwei Faktoren optimiert werden [27].

Einerseits durch die Erhöhung der übertragenen Datenmengen (z.B. Bit) in einem fixierten

Zeitrahmen (z.B. in einer Sekunde). Dies wird im weiteren Verlauf dieser Arbeit als Bandbreite bezeichnet [72].

Eine zweite Optimierungsmöglichkeit ist eine Reduktion von Latenz. Latenz wird in diesem Zusammenhang definiert als der Zeitraum, der benötigt wird, um von einer Quelle ein Bit zu einem Ziel zu senden [72].



Quelle: https://hpbn.co/assets/diagrams/940cb8cfbb433a04b05e15b4868cb8e3.svg

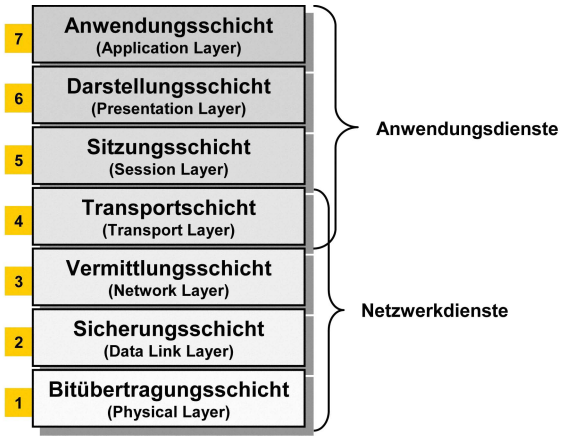
Abbildung 1.1.: Seitenladezeit im Verhältnis zur Bandbreite und Latenz

Die Abbildung 1.1 zeigt, in welchem Maße Bandbreite und Latenz die Ladezeit einer Internetseite beeinflussen.

Wie in der Abbildung zu erkennen ist, hat die Erhöhung der Bandbreite von 5 Mbps auf 10 Mbps kaum noch einen Einfluss auf die Ladezeit einer Internetseite [27]. Aktuell stehen Benutzern des öffentlichen Internets im globalen Durchschnitt 7.4 Mbps Bandbreite zur Verfügung [3]. Es folgt nach Analyse der Abbildung 1.1, dass eine Erhöhung der Bandbreite keine weiteren nennenswerten Effekte haben wird.

Die Latenz ist, wie auf Abbildung 1.1 zu sehen, deutlich besser geeignet, erfolgreiche Optimierungen an der Ladezeit von Internetseiten vorzunehmen. Wie aus der Abbildung ersichtlich, bewirkt jede Latenzreduktion um 20 Millisekunden eine signifikante Verbesserung der Seitenladezeit. Folglich kann durch diesen Faktor, die Reduzierung der Latenz, ein messbarer Mehrwert für Benutzer von Internetseiten erreicht werden [24].

Um die Latenz zu optimieren, können auf unterschiedlichen Schichten einer Netzwerkkommunikation Optimierungen durchgeführt werden [44, 5]. Um diese Kommunikationsabläufe eines Netzwerkes genauer untersuchen zu können, wird ein Schichtenmodell eingesetzt, das Open Systems Interconnection Model (OSI-MODELL) [79]. Die sieben Schichten dieses Modells grenzen jeweils einen Aufgabenbereich ab. Die Schichten sind mit fest definierten Schnittstellen versehen, so dass eine Schicht jederzeit ausgetauscht werden kann.



Quelle: http://www.airnet.de/cr1-gfe/de/html/InfoAustKomp_learningObject6.xml

Abbildung 1.2.: Das OSI-Schichten Modell

Die Aufgaben der verschiedenen Schichten lassen sich folgendermaßen zusammenfassen (siehe Abbildung 1.2):

1. Bitübertragungsschicht: Wandelt die Bits in ein Signal um, das eine physikalische Übertragung ermöglicht.

2. Sicherungsschicht: Fügt Prüfsummen hinzu und segmentiert die Pakete in Frames, um eine möglichst fehlerfreie Übertragung zu ermöglichen.

3. Vermittlungsschicht: Routet die Datenpakete zum richtigen Knoten, damit der adressierte Nutzer die für ihn bestimmten Datenpakete erhält.

4. Transportschicht: Ordnet die empfangenen Daten dem korrekten Port zu und ermöglicht dadurch Anwendungen einen einheitlichen Zugriff auf das Kommunikationsnetz.

5. Sitzungsschicht: Kontrolliert den Datenaustausch und die Verbindung selbst, um eine Prozesskommunikation zwischen den Systemen zu erlauben.

6. Darstellungsschicht: Konvertiert Daten von einem systemabhängigen Format in ein unabhängiges Format, um einen korrekten Datenaustausch zwischen zwei unterschiedlichen Systemen zu ermöglichen.

7. Anwendungsschicht: Stellt die Funktionen für die Anwendung selbst bereit. Ermöglicht es dem Nutzer, Dateneingaben und -ausgaben durchzuführen.

Die auf der Abbildung 1.2 gekennzeichneten Schichten - Bitübertragungsschicht, Sicherungsschicht, Vermittlungsschicht - sind mit Ausnahme der Transportschicht realistisch nicht veränderbar [50], denn dafür müssten weltweit Infrastrukturelemente wie Router, NAT-Boxen, Switches und andere Middleware-Geräte verändert bzw. angepasst werden [50].

Die Schichten, die in Abbildung 1.2 als Anwendungsdienste dargestellt sind - Sitzungsschicht, Darstellungsschicht und Anwendungsschicht - werden zusammen behandelt, da sie bei den für Internetseiten relevanten Protokollen zusammen implementiert werden [5]. Als relevante Protokolle für diese Schichten sollten im Kontext von Internetseiten Hypertext Transfer Protocol (HTTP/1) [23] und Hypertext Transfer Protocol 2 (HTTP/2) [5] genannt werden [12]. Auf HTTP/1 wird in dieser Ausarbeitung nicht weiter eingegangen, da die Limitationen und Probleme dieses Protokolls als ausreichend dokumentiert behandelt werden können [59].

HTTP/2 hingegen ist ein Protokoll, welches u.a. als Ziel hat, die Latenz im Vergleich zum Vorgänger HTTP/1 zu reduzieren [27]. Das Protokoll erhielt dafür neue Funktionalitäten wie Multiplexing (durch eine *stream*-Abstraktion), eine binärkodierte Übertragung und eine Priorisierung von *streams* [5].

Die Ansätze, die HTTP/2 bietet, sind vielversprechend, um die Seitenladezeit optimieren zu können. Das Multiplexing in HTTP/2 ermöglicht es über eine darunter liegende Schicht, Daten über mehrere sogenannte *streams* zu verschicken. Damit ist es möglich, mehrere unterschiedliche Daten über eine Verbindung zu versenden. Eine Herausforderung der Implementierung ist dabei, dass es auf der Transportschicht aufbaut und damit seine Daten über ein Transportprotokoll verschickt [5]. Dadurch können ungewünschte Verhaltensweisen vererbt werden, die bei Funktionalitäten wie z.B. dem Multiplexing auftreten können. Eine solche Herausforderung, welche so vererbt wurde, ist das Head of Line (HoL)-Blocking [63], bei dem durch den Verlust eines Pakets innerhalb des Transportprotokolls auch die *stream*-Datenübertragung vollständig blockiert werden kann. Zur Veranschaulichung soll von einer Webseite mit zwei Bildern ausgegangen werden. Wenn beide Bilder sequenziell auf zwei unterschiedlichen *streams* übertragen werden und beim ersten Bild ein Übertragungsfehler auf der Transportschicht auftritt, kann das zweite Bild nicht an die zu verarbeitende Anwendung gesendet werden, bis der initiale Fehler behoben ist. Um eine solche Herausforderung lösen zu können, muss auf einer tieferen Schicht gearbeitet werden [63]. Es soll im weiteren Verlauf der Arbeit darum gehen, diese Mechanismen wie *stream*-Multiplexing in der Transportschicht zu implementieren [14].

Die Protokolle in der Transportschicht können in zwei Kategorien eingeteilt werden: verbindungslose und verbindungsorientierte Protokolle [50].

Die verbindungslosen Transportprotokolle wie User Datagram Protocol (UDP) [56] und Datagram Congestion Control Protocol (DCCP) [43] bieten keine Möglichkeit eines verlässlichen Verbindungsaufbaus. Um ein verbindungslosen Transportprotokolle einsetzen zu können, müssen alle übertragenen Daten pro Paket voneinander unabhängig sein und dürfen keine Beziehung zu den vorherigen oder nachfolgenden Paketen besitzen [50].

Bei verbindungsorientierten Protokollen, so wie aktuell z.B. Transmission Control Protocol (TCP) [57] oder Stream Control Transmission Protocol (SCTP) [70], gibt es immer drei Phasen: einen Verbindungsaufbau, eine Datenübertragung und einen Verbindungsabbau [50].

Eine weitere Fähigkeit, die Protokolle besitzen können, ist die reihenfolgegesicherte Übertragung. Eine Übertragung ist reihenfolgegesichert, wenn die Datenpakete beim Ziel wieder in die von der Quelle versendete Reihenfolge gebracht werden [57].

Um eine Internetseite zu laden, werden nur verbindungsorientierte und reihenfolgegesicherte Transportprotokolle verwendet [5].

TCP [57] ist das am häufigsten unter HTTP/2 verwendet Transportprotokoll. Ein bekanntes Problem an TCP ist, dass es durch seine Architektur ein HoL-Blocking ermöglicht [64]. Des Weiteren hat es einen Drei-Wege-Handschlag, der benötigt wird, um einen Verbindungsaufbau durchzuführen. Die Herausforderung daran ist, dass der Datenverkehr über HTTP/2 vollständig verschlüsselt stattfindet [53, 29]. Dadurch wird eine weitere Verbindungsaufbauphase benötigt, um die Verschlüsselung zu aktivieren [49]. Eine weitere Herausforderung ist, dass dieses Transportprotokoll überwiegend im Kernel implementiert wird [10]. Im Kernel implementierte Software ist schwierig upzudaten und zu *debuggen*. Als Beispiel seien hier die Smartphones genannt, welche nach etwa 2 Jahren nicht mehr weitere Updates erhalten und damit auch keine Aktualisierung des Transportprotokolls [26, 46].

Ein Hauptgrund für die Entwicklung von SCTP [70] war es, das HoL-Blocking-Problem zu lösen [28]. Dieses wurde hier mit einer Multiplexing-Architektur über *streams* realisiert. Das Problem an diesem Protokoll ist jedoch, dass es als neues Protokoll geringe Durchsetzungschancen besitzt, da bestehende technische Infrastruktur diesen Datenverkehr nicht unterstützt

[30].

Damit ein neues Transportprotokoll auf vorhandener Infrastruktur eingesetzt werden kann, sollte es auf ein bereits vorhandenes Transportprotokoll aufbauen [73]. Ein verbindungsloses Protokoll mit geringer eigener Funktionalität wie UDP bietet sich dazu an. Alle benötigten Funktionalitäten können in der Anwendungsschicht implementiert werden [73]. Dieser Ansatz hat einen potentiellen Nachteil: die Implementierungen des Protokolls können deutlich voneinander abweichen und sogar proprietäre Lösungen entstehen. Ein entscheidender Vorteil dieses Vorgehens aber ist die Flexibilität, Änderungen an Implementierungen schnell durchführen zu können, so wie es bei anderer Softwarentwicklung auch der Fall ist.

Ein Protokoll, welches diesem Ansatz folgt, ist Quick UDP Internet Connection (QUIC).

Ursprünglich wurde QUIC im Jahre 2012 von Google entwickelt [12]. Es ist 2017 für 7 Prozent des weltweiten Traffics verantwortlich gewesen [44].

QUIC ist eine Kombination aus den Funktionaltitäten von TCP [57], Transport Layer Security Protocol (TLS) [19] und HTTP/2 [5] [16], die es ermöglicht, gewisse Merkmale wie z.B. den Verbindungsaufbau von TCP und TLS zu vereinen, mit dem Vorteil einer Zeitersparnis dieser Operation.

Im Jahr 2015 [41] hat Google das Protokoll der Internet Engineering Task Force (IETF) übergeben, um es zu standardisieren. Die IETF ist eine Organisation, die es sich zur Aufgabe gemacht hat, das Internet zu verbessern, indem sie hochqualitative, relevante, technische Dokumente erstellt [4]. Die IETF setzt sich für offene und transparente Prozesse ein, so dass Entscheidungen nachvollziehbar sind und eine Mitwirkung interessierter Parteien möglich ist. Dies wird durch offene und transparente Kommunikation wie z.B. die Öffnung von Mailing-Listen und das zur öffentlichen Verfügung stellen von Mitschnitten von Terminen erleichtert [4].

Die technische Arbeit in der IETF wird in Arbeitsgruppen umgesetzt, die anhand von Themen organisiert werden. Es gibt unter den vielen verschiedenen Bereichen z.B. die Themen *routing*, *transport* und *security* [17]. Die Kommunikation der Arbeitsgruppen findet überwiegend über Mailing-Listen statt [32, 4].

Für das Protokoll QUIC wurde eine IETF-Gruppe eingerichtet [44], die bis voraussichtlich Juli

2019 an der Standardisierung des ersten IETF QUIC-Protokolls arbeitet [58]. Bislang gibt es sogenannte *drafts*, welche Vorschläge für eine Spezifikation des Protokolls sind, aber noch nicht fertig ausdiskutiert wurden. Da das Protokoll sich aktuell in der Entwicklung befindet, gibt es noch keine Studien mit echtem Datenverkehr, mithilfe derer Protokollentwürfe evaluiert werden könnten [58].

Um Evaluationen wie z.B. eine Parameterstudie durchführen zu können, bedarf es einer Implementierung der Spezifikation eines *drafts*. Dies wird durch eine konkrete Implementierung, eine Emulation oder eine Simulation möglich.

Eine konkrete Implementierung auf den Netzwerk-Stack eines Betriebsystems ist die offensichtlichste Möglichkeit, das Protokoll zu implementieren. Der Nachteil an dieser Variante ist, dass durch die äußerlichen Einflüsse die gemessenen Werte beim Evaluieren des Entwurfs schwanken können und dadurch nicht reproduzierbar sind [11]. Aus diesem Grund wurde gegen eine konkrete Implementierung entschieden.

Eine Emulation ist für den Entwicklungsprozess nicht besonders geeignet, da die Emulation in zeitlicher Hinsicht meist so schnell wie das zu emulierende System arbeitet [48].

Bei einer Simulation ist es möglich, kostengünstig große und komplexe Netzwerke mit einem geringen Hardwareaufwand [48, 11] zu simulieren. Des Weiteren sind Evaluation im rahmen von Simulationen reproduzierbar, da sie durch keine äußerlichen Einflüsse beinträchtigt werden. Das ermöglicht es, Parameterstudien durchzuführen. Diese sind Versuchsaufbauten mit unterschiedlichen Parameterkonfigurationen. Als Beispiel sei eine TCP-Verbindung genannt, welche einen Verbindungsaufbau in einem simulierten Netzwerk durchführt, bei dem der Parameter des Netzwerk-Delays variiert wird. Durch diese Parameterstudie könnten Schlüsse auf die *delay*-Sensitivität des Verbindungsaufbaus geschlossen werden. Da es das Ziel ist, in Zukunft Parameterstudien mit der zu entwickelnden Software zu realisieren, ist eine Implementierung in einer Simulation im Rahmen dieser Arbeit zu bevorzugen.

Als Simulations-Framework bieten sich z.B. ns-3 [31] oder OMNeT++ [75] an. Da an der HAW Hamburg häufig mit OMNeT++ gearbeitet wird [71, 47] und es international ein großes Interesse an diesem Framework gibt [36], fällt die Auswahl, auch auf Grund seiner Wart- und Pflegbarkeit, auf OMNeT++ als Simulations-Framework.

OMNeT++ [75] ist ein erweiterbares, modulares, in C++ geschriebenes zeitdiskretes Eventsimulations-Framework [74], welches selbst keine Simulation für eine konkrete Anwendung, sondern die Infrastruktur und Werkzeuge für die Erstellung von Simulationen bereitstellt [74].

Eine der Kernfunktionalitäten der Infrastruktur ist die komponentenbasierte Architektur für Simulations-Modelle. Modelle werden in OMNeT++ aus wiederverwendbaren Komponenten gebaut, die als Module bezeichnet werden [74].

Diese Module werden von Bibliotheken wie INET Framework bereitgestellt. Das INET Framework [37] ist eine Open-Source-Modell-Bibliothek für das OMNeT++ Simulations-Framework.

INET Framework stellt Modelle für Protokolle, Agents und weitere für Forschung an Kommunikationsnetzwerken notwendige Bereiche zur Verfügung [37]. Die Bibliothek beinhaltet Modelle für den vollständigen Internet-Stack (TCP, UDP, IPv4, IPv6 und viele mehr), außerdem noch kabelbasierte und kabellose Protokolle, die auf der Sicherungsschicht angesiedelt sind wie z.B. Ethernet [37].

Es gibt viele weitere Simulations-Bibliotheken, die das INET Framework als Basis benutzen und dessen Funktionalität erweitern. Core4Inet sei an dieser Stelle erwähnt, welches überwiegend Netzwerktechnologien für die Automobilindustrie bereitstellt [68].

Das INET Framework besteht aus Modulen, welche über Nachrichten miteinander kommunizieren. Agenten und Netzwerkprotokolle werden als Komponenten dargestellt, die mit Netzwerkgeräten wie z.B. Hosts, Router oder Switches kombiniert werden können.

Neue Komponenten können relativ einfach implementiert werden. Die Einarbeitung soll dabei möglichst mit geringem Zeitaufwand vollzogen werden können, da bei bestehenden Komponenten darauf geachtet wurde, dass sie intuitiv zu verstehen, zu verändern und zu erweitern sind [37].

1.2. Ziel

Es soll mithilfe des Simulation Framework OMNeT++ [74] und der Bibiothek INET Framework [37] eine Erweiterung erstellt werden, die das INET Framework um ein Modul ergänzt, welches die Funktionalitäten von QUIC bereitstellt. Als Spezifikation für diese Implementation wird

der IETF QUIC *draft* 10 [39] verwendet.

Da eine vollständige Implementierung eines Modells des Protokolls den Rahmen dieser Arbeit überschreiten würde, soll ein Teil des QUIC-Protokolls entwickelt werden: der *1-RTT handshake*, der während der Verbindungsaufbauphase durchgeführt wird. Dieser wird implementiert und anschließend evaluiert. Ziel dieser Arbeit ist es, ein Grundgerüst für die Durchführung von Parameterstudien zu entwickeln.

1.3. Aufbau

Diese Arbeit ist in sieben Kapitel aufgegliedert.

Kapitel 1
In der Einleitung wird ein Überblick über die Relevanz der Arbeit gegeben sowie der Grund dieser Arbeit behandelt.
Kapitel 2
Verwandte Arbeiten werden vorgestellt und mit dieser Arbeit verglichen.
Kapitel 3
Das Transportprotokoll QUIC wird vorgestellt und auf die für das Verständnis der Arbeit notwendigen Funktionen und Strukturen eingegangen.
Kapitel 4
Die Anforderungen an das zu implementierende System werden aufgeführt und spezifiziert.
Kapitel 5
Das Design und die Entwicklung für die QUIC-Implementierung in OMNeT++ werden beschrieben. Der Designansatz, der vorgefundene Status und die Umsetzung werden erörtert.
Kapitel 6
Eine Evaluation einer Testumgebung mit unterschiedlichen *delay*-Konfigurationen wird durchgeführt und anschließend ausgewertet.
Kapitel 7
Die Inhalte der Arbeit werden zusammengefasst. Das Ergebnis wird reflektiert und es wird ein Ausblick mit weiterführenden Fragestellungen gegeben.

2. Verwandte Arbeiten

Im Folgenden werden ausgewählte Arbeiten zu QUIC vorgestellt, und es wird erläutert, in welcher Hinsicht sie sich von dem hier beschriebenen Vorhaben unterscheiden und welche neuen Ansätze diese Arbeit liefern soll.

2.1. Arbeiten zu QUIC

Die zwei hier ausgewählten verwandten Arbeiten „How quick is QUIC?“ und „Quicker: On the design and implementation of the QUIC protocol“ sollen einen Eindruck über den aktuellen Forschungsstand geben und dazu genutzt werden, eine Abgrenzung zu dieser Arbeit durchzuführen.

2.1.1. How quick is QUIC?

In ihrer Arbeit „How quick is QUIC?“ von 2016 befassen sich Péter Megyesi und Zsolt Krämer sowie Sándor Molnár mit der Entwicklung des Internets und im Speziellen mit der Entwicklung der Ladezeit von Internetseiten [49]. Dazu werden die Protokolle SPDY [6], QUIC [45] und HTTP/1 [23] in einer Testumgebung in 48 verschiedenen Testszenarien untersucht und verglichen. Die Testumgebung ist ein Laptop, der von einer Google-Seite über einen *Shaper Server*, welcher die unterschiedlichen Netzwerksituationen emulieren soll, eine Website mit unterschiedlichen Varianzen an Bildergrößen und Bilderanzahl herunterlädt. Jedes dieser Szenarien wird einhundert Mal durchgeführt, um trotz möglichen schwankenden Werten einen aussagekräftigen Mittelwert bilden zu können. Als interessantes Ergebnis der Untersuchung ist herausgekommen, dass QUIC in 40 Prozent der Szenarien die Seiten-Ladezeit reduzieren kann, im Vergleich zu den anderen beiden getesteten Protokollen. Als Vorschlag wird auch erwähnt, dass für mobile Geräte vor allem über mobile Netzwerktechnologien wie 3G, wo es eine größere Paketumlaufzeit bzw. Round Trip Time (RTT) *delays* gibt, QUIC [45] eingesetzt werden sollte, da es unter diesen Voraussetzungen besser als die anderen getesteten Protokolle arbeitet [49]. Als RTT wird hier die Definition:

> „... the round-trip time, the interval between the sending of a packet and the receipt of its acknowledgement, is a key function in many reliable transport protocols ...“ [42]

verwendet. RTT ist demnach die Zeit, welche benötigt wird, um ein Datenpaket in einem Rechnernetz von der Quelle zum Ziel und zurück zu versenden. Es ist also die Summe aus den Zeiten des Transportes von der Quelle A nach Ziel B und der von Ziel B nach Quelle A.

In dieser vorgestellten Arbeit wird noch eine Version von Googles QUIC-Spezifikation verwendet, was dazu führt, dass das Verhalten nicht mit dem QUIC-Protokoll der IETF-Spezifikation vergleichbar ist. Das führt dazu, dass die Testergebnisse ein nicht mehr verwendbares Bild darstellen. Folglich ist das dort Herausgefundene nicht mehr auf diese hier zu implementierende Version von QUIC anwendbar.

2.1.2. Quicker: On the design and implementation of the QUIC protocol

In der Arbeit „Quicker: On the design and implementation of the QUIC protocol“ (2018 von Kevin Pittevils) wird eine IETF QUIC *draft* 12 [40] Implementierung in der Programmiersprache Javascript vorgenommen, die in der Laufzeitumgebung NodeJS [66] ausgeführt wird [55]. In der Arbeit wird versucht, eine Implementierung zu erstellen, welche den Spezifikationen des *draft* 12 [40] entspricht und eine Interoperabilität mit anderen Implementierungen besitzt. Dafür wird eine Kommunikation mit unterschiedlichen Implementationen von dem *draft* durchgeführt, als Beispiel seien hier Mozquic [15] und PicoQuic [34] erwähnt. Die Arbeit beschäftigt sich damit, eine funktionale Implementierung zu entwickeln, welche mit anderen Versionen interagieren kann.

Durch die konkrete Implementierung ist es - wie zuvor bemerkt - aufwendig, Tests mit komplexen Netzwerkstrukturen durchführen zu können. Parameterstudien sind daher nicht möglich. Des Weiteren liegt der Fokus dieser Arbeit auf der Implementierung eines *drafts* und der anschließenden Sicherstellung der Kompatibilität zu anderen Versionen.

2.2. Abgrenzung zu verwandten Arbeiten

In dieser Arbeit wird eine Grundlage zur Durchführung von Parameterstudien realisiert. Die entwickelte Implementierung in dieser Arbeit soll zeigen, wie *delay*-sensitiv diese *draft* 10 [39] Implementierung im Vergleich zu anderen simulierten Transportprotokollen ist.

In der Arbeit „How quick is QUIC?“ [49] geht es um die Seitenladezeit, welche unter verschiedensten Netzwerksituationen mit unterschiedlichen Protokollen evaluiert wurde. In dieser Arbeit hingegen wird eine Grundlage zur Durchführung von Parameterstudien realisiert. Konkret soll die *delay*-Sensitivität der Verbindungsaufbau-Phase dieser Implementierung mit anderen Transportprotokollen verglichen werden. In der erwähnten Arbeit wird auch nicht die QUIC-Implementierung der IETF verwendet, was dazu führt, dass keine Schlussfolgerung auf die IETF Version gezogen werden kann. Ebenfalls erwähnenswert ist, dass QUIC in dieser Evaluation die quic+http [49] Variante von Google ist, welche auf Googles eigener Infrastruktur [49] genutzt wird. Diese Implementierung arbeitet auch auf einer anderen OSI-MODELL-Schicht als das hier zu implementierende IETF QUIC. In der IETF QUIC Version wurden diese zwei Protokolle getrennt, in das Transportprotokoll [39] und die HTTP-Erweiterung [7]. In dieser Arbeit soll es darum gehen, das Transportprotokoll zuerst zu realisieren und anschließend eine Evaluation auf der OSI-MODELL-Schicht der Transportprotokolle durchzuführen.

In „Quicker: On the design and implementation of the QUIC protocol“ [55] wird die Implementierung eines IETF *drafts* von QUIC gezeigt. Die Arbeit unterscheidet sich aber dahingehend von der hier vorliegenden, als dass hier ein Modul für eine zeitdiskrete Eventsimulation entwickelt werden soll und keine konkrete Implementierung angefertigt wird. Außerdem ist das Ziel der Arbeit, einen Stand zu erhalten, der kompatibel zu den bisherigen Implementierungen von QUIC ist. Wobei in dieser Arbeit der Fokus darauf gelegt ist, einen Anfang einer Implementierung zu realisieren, welche für Parameterstudien geeignet ist, um den Protokollentwurf selbst besser untersuchen zu können.

3. QUIC

In diesem Kapitel wird das Transportprotokoll vorgestellt und im Detail beschrieben. Anschließend wird auf die für diese Arbeit besonders relevanten Aspekte von QUIC eingegangen.

3.1. QUIC eine Übersicht

QUIC ist ein Transportprotokoll. Das Design des Protokolls besteht aus einer Ansammlung von Funktionalitäten, die es ermöglicht, das Transportprotokoll für viele allgemeine Anwendungszwecke einzusetzen [39].

Es folgt eine Auflistung einiger Kernfunktionalitäten von QUIC [39]:

1. Versions-Negotiation [39, p. 9]
2. Verbindungsaufbau mit geringer Latenz [39, p. 7]
3. Verschlüsselte Datenübertragung [39, p. 8]
4. *stream*-Multiplexing [39, p. 7]
5. *stream*-Flusskontrolle [39, p. 78]
6. Verbindungs Migration [39, p. 35]

QUIC wird im Kontext und mit dem Hintergrund von bekannten und weit verbreiteten Transportprotokollen wie u.a. TCP, SCTP entwickelt [39]. Es wird im Unterschied zu den meisten anderen Transportprotokollen auf Basis eines anderen Transportprotokolls, nämlich UDP, entwickelt [39]. UDP besitzt nicht alle Funktionalitäten, die von einem Transportprotokoll erwartet werden. Eigentlich leitet es nur Daten über das im OSI-MODELL darunterliegende Protokoll weiter [9]. Andrew Tanenbaum bezeichnet die Arbeitsweise von UDP:

> „It does almost nothing beyond sending packets between applications.“[72]

Dieses darunterliegende Protokoll führt ein Multiplexing/Demultiplexing durch. Als Multiplexing/Demultiplexing wird in diesem Zusammenhang folgende Definition verwendet:

> „Multiplexing is the process by which information bits from N incoming channels are transferred into bit times on one outgoing channel. Demultiplexing is the reverse process: the information bits on one incoming multiplexed channel are separated and transferred onto N outgoing channels. The multiplexed outgoing channel contains some extra bits (in addition to the incoming channels' data bits) that the demultiplexer uses to determine which data bits belong to which incoming channel."[76]

Dies geschieht bei einem Transportprotokoll durch ein Port-Nummern-System. Die Port-Nummern sind in dem Beispiel von Multiplexing die *incoming channels*, welche auf einem *outgoing channel* gebündelt werden [9]. Das gilt auch für das Demultiplexing von einem *channel* zu mehreren *channels*.

QUIC ist deshalb auf Basis von UDP entwickelt, damit existierende Hardware von z.B. Middleboxes [13], deren Betriebssysteme nicht weiterentwickelt werden oder werden können, auch mit diesem Protokoll arbeiten können [50]. Als Middleboxes werden Netzwerkgeräte, die Header und Nutzdaten verändern, bezeichnet (eine ausführliche Beschreibung ist im RFC 3234 [13] zu finden). Nutzdaten sind Daten, die keine Steuer- oder Protokollinformationen transportieren, sondern die Information, die genutzt werden soll, z.B. die Audio-Datei, die versendet werden soll oder der Text, der gesendet wird [72].

Das Protokoll verschlüsselt die Datenkommunikation mit dem TLS 1.3 Standard, um die Nutzdaten sicher zu übertragen [39]. Um eine verschlüsselte Datenübertragung mit TLS 1.3 zu ermöglichen, muss ein Schlüsselaustausch stattfinden. Dieser Schlüsselaustausch wird während des Verbindungsaufbaus durchgeführt. Im weiteren Verlauf wird der Verbindungsaufbau auch als *handshake* bezeichnet. TLS 1.3 wurde im August 2018 [60] von der IETF als offizieller Standard verabschiedet.

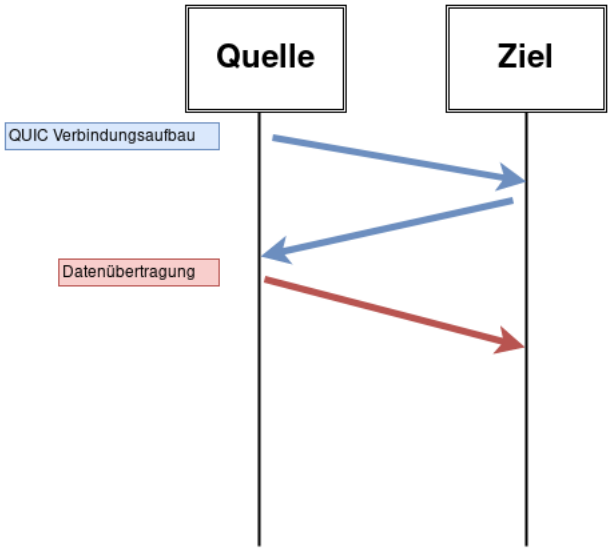
Durch die Versions-Negotiation [39], in dem die Quelle dem Ziel eine Anfrage mit seiner prefärierten Version des QUIC-Protokolls stellt, überprüft das Ziel, ob es diese Version unterstützt. Wenn dies nicht der Fall ist, schickt das Ziel eine Liste mit unterstützten Versionen zurück. Durch dieses Verhalten ist es möglich, eine kontinuierliche Weiterentwicklung des Protokolls zu betreiben oder unterschiedliche Versionen parallel in Betrieb zu haben.

3.2. Verbindungsaufbau

Das Transportprotokoll QUIC besitzt im Gegensatz zu anderen Protokollen einen Verbindungsaufbau, der eine geringe Latenz benötigt, um durchgeführt zu werden [49, 57, 39]. Dies wird durch die *handshake*-Mechanismen des Protokolls erreicht. QUIC besitzt mehrere *handshake*-Mechanismen, was es von anderen Transportprotokollen unterscheidet [39].

Der *0-RTT handshake* ermöglicht es, Daten direkt zu versenden, ohne eine RTT zu benötigen.

Die *0-RTT* Variante des *handshakes* ist nur möglich, wenn die Quelle das Ziel bereits kennt. Diese Art von Verbindungsaufbau wird im weiteren Verlauf der Arbeit nicht im Detail behandelt. Der Fokus liegt auf dem sogenannten *1-RTT handshake*, denn dieser muss vor dem *0-RTT* durchgeführt werden, da durch den *1-RTT* die Quelle dem Ziel bekannt gemacht wird [39].



Quelle: Eigenentwurf

Abbildung 3.1.: Beispiel eines *1-RTT handshakes*

3.2.1. 1-RTT Handshake

Der *1-RTT handshake* benötigt nur eine RTT, um den *handshake* durchzuführen. Wie in der Abbildung 3.1 zu sehen, versendet die Quelle eine Nachricht an das Ziel, diese antwortet darauf und der *handshake* ist erfolgreich durchgeführt. Anschließend kann der Datenstransfer stattfinden [39].

Um die Nachrichten (die im weiteren Verlauf auch Pakete genannt werden), die zwischen der Quelle und dem Ziel versendet werden, genau zu spezifizieren, werden im Folgenden die dafür notwendigen Bestandteile vorgestellt.

3.2.2. Variable-Length Integer Encoding

Das *variable-length integer encoding* wird für fast alle positiven Zahlenwerte innerhalb von QUIC verwendet. Diese Verwendungsstellen werden in dem *draft* 10 [39] mit einem *i* gekennzeichnet, was für diese Arbeit übernommen wird. Das Encoding stellt sicher, dass kleinere Zahlwerte weniger Bytes benötigen als größere. Dieses bringt den Vorteil, dass bei der Übertragung mehr Nutzdaten versendet werden können. Die Codierung verwendet die ersten zwei Bits (*two most significant bits* [39]) im ersten Byte, um die Länge des Zahlenwertes zu definieren.

In den restlichen Bits wird der Zahlenwert in der *network byte order* [72] gespeichert, was auch als *Big-Endian* [72] bezeichnet wird. *Big-Endian* bedeutet, dass die höchstwertigen Bytes an der niedrigsten Adresse gespeichert werden [72].

Der Zahlwert kann 1, 2, 4 oder 8 Bytes groß sein. Damit ist es möglich, Zahlenwerte mit 6, 14, 30 oder 62 Bit darzustellen, wie in Tabelle 3.1 abgebildet ist.

2 Bit	Länge in Bytes	Benutzbare Bits	Wertebereich
00	1	06	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Quelle: [39]

Tabelle 3.1.: Übersicht der Variable-Length Integer Encodings

Zur Erläuterung des Prinzips folgt ein Beispiel.

Es soll der Wert 42 encodiert werden:

```
// 42 in der hexadezimalen Schreibweise.
Hex: 0x2A

// 42 in Variable-Length Integer Encoding
// (Binaere Schreibweise)
Bin: 00 101010

Encoding Bits
Werte Bits
```

Listing 3.1: Der Wert 42 als Variable-Length Integer Encoding

Im Listing 3.1 ist der Wert 42 als hexadezimale Schreibweise zu sehen. Dieser Wert wird in das binäre Format mit *variable-length integer encoding* umgewandelt. Die umgewandelte Zahl ist in zwei Farben dargestellt. Die rot eingefärbten Ziffern stellen die encodierten Bits da. Diese

entsprechen den Werten der Tabelle 3.1, in der ersten Zeile. Die blau dargestellten Ziffern entsprechen dem eigentlichen Wert 42.

3.2.3. Header

Alle Pakete innerhalb einer QUIC-Verbindung besitzen einen *long header* oder einen *short header*. Die *header types* können anhand des *header form bit* [39] unterschieden werden. Dieses Bit ist das erste Bit in einem Paket von QUIC (im weiteren auch als QUIC-Paket bezeichnet). Im Folgenden werden beide Arten von *header* vorgestellt.

Short Header

Der *short header* ist eine minimale Version eines *headers*, welche nach dem einmaligen Verbindungsaufbau verwendet wird. Durch den *short header* ist es möglich, mehr Nutzdaten im Vergleich zu dem *long header* zu versenden [39]. Eine genauere Beschreibung dieses *headers* ist nicht notwendig, da diese Art von *header* in der hier vorliegenden Arbeit nicht verwendet wird.

Long Header

Wie in Tabelle 3.2 zu sehen, besitzt der *long header* ein *header form bit* mit einem auf 1 gesetzten Bit. Dieses *header* Format wird nur in Paketen in der Verbindungsaufbauphase verwendet. Anschließend wird in QUIC zum *short header* Format gewechselt [39].

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
1	Type (7)																														
>	Connection ID (64)																														<
Version (32)																															
Packet Number (32)																															
Payload (*)																															

Quelle: [39]

Tabelle 3.2.: Long Header Paket Format

Die Tabelle 3.2 zeigt die Struktur eines *long headers*. Das erste Bit des ersten Bytes ist auf 1 gesetzt, um zu kennzeichnen, dass es sich um ein Paket mit einem *long header* handelt. Die verbleibenden 7 Bits im ersten Byte definieren, um welchen Typ von Paket es sich handelt. Folgende Typen sind im QUIC *draft* 10 [39] spezifiziert:

Type (Hexadezimal)	Namen
0x7F	Initial
0x7E	Retry
0x7D	Handshake
0x7C	0-RTT Protected

Quelle: [39]

Tabelle 3.3.: Long Header Types

Für den *1-RTT handshake* werden die Typen *intitial* und *handshake* benötigt. Bei einem Verbindungsaufbau mit dem *1-RTT handshake* wird von der Quelle ein *long header* mit dem Typ *initial* geschickt und das Ziel antwortet mit einem *long header* vom Typ *handshake*. Die *Connection ID* ist in den Bytes 1 bis 8 enthalten. Sie wird benutzt, um die Verbindung zu identifizieren. Im Gegensatz zu Protokollen wie zum Beispiel TCP wird bei QUIC eine Verbindung nicht direkt anhand des *4-tuples* (*Source Port*, *Source IP*, *Destination Port*, *Destination IP*) [72] identifiziert, sondern anhand der *Connection ID*. Das Ziel muss ein Mapping der *Connection ID* zu dem *4-tuples* führen, damit die Identifikation über die *Connection ID* möglich ist [51].

Der Vorteil dieser Implementierung ist, dass QUIC durch diese Maßnahme Network Address Translation (NAT) *rebind* [67] und Internet Protocol (IP)-Adressen änderungssicher ist [39]. Bei einem NAT *rebind* werden die IP-Adresse und der Port gegebenenfalls geändert, was eine Änderung des *4-tuples* bedeutet und im Fall von TCP [57] den Aufbau einer neuen Verbindung erfordert [67], da die Identifizierung der Verbindung anhand des *tuples* durchgeführt wird. Bei QUIC hingegen wird das mit einer *Connection ID* korrespondierende *4-tuples* erneuert, wenn es sich ändert und dadurch muss keine neue Verbindung aufgebaut werden. Dieses Verhalten wird auch *Connection Migration* genannt [51].

Die Version ist in den darauffolgenden 3 Bytes definiert. Für die *Packet Number* stehen weitere 3 Bytes zur Verfügung. Der hier als *Payload* bezeichnete Bereich ist dynamisch und kann ab dem 17. Byte gesetzt werden. Als *Payload* wird der Bereich bezeichnet, in den weitere Inhaltselemente (auch *frames* genannt) gepackt werden können.

3.2.4. Payload / Frames

Sowohl in *short headern* als auch in *long headern* gibt es ein Feld *Payload*. Dieses besteht aus einer Anzahl von *frames*, die wie in einer Liste hintereinander angeordnet sind. Die *frames*

haben einen Präfix, der bestimmt, um was für einen Typ von *frame* es sich handelt. Es können 256 unterschiedliche *frame*-Typen erstellt werden, was QUIC sehr flexibel macht [39]. Ein *frame* muss in ein QUIC-Paket passen [39]. Entsprechend können *frames* wie eine Art atomare Einheit begriffen werden, die verschickt werden können.

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
Type (8)										Type-abhängige Felder (*)																				...	

Quelle: [39]

Tabelle 3.4.: Generic Frame Layout

In der Tabelle 3.4 wird der Aufbau eines *frames* gezeigt. Die ersten 8 Bytes definieren den Typ, gefolgt von type-abhängigen Feldern. QUIC hat eine große Menge an unterschiedlichen *frame*-Typen, die hier nicht vollumfänglich behandelt werden können. Es werden nur die für diese Arbeit relevanten *frame*-Typen erörtert:

Padding Frame

Der *PADDING Frame* (type=0x00) wird benutzt, um die Größe des Pakets zu erhöhen. Der *frame* enthält nur den Typ und keine weiteren type-abhängigen Felder.

Streams

Streams sind die wichtigsten *frame*-Typen, denn sie transportieren Nutzdaten. Der *frame*-Type von *streams* ist dynamisch - er variiert von 0x10 bis 0x17 (Hexadezimal). Die drei niederwertigsten Bits 1, 2 und 4 definieren dabei spezielle Eigenschaften:

- Das *OFF* bit (0x04) aktiviert das *Offset*-Feld, wenn es gesetzt ist. Standardmäßig ist bei Nichtaktivierung ein *Offset*-Wert von 0 gesetzt.
- Das *LEN* bit (0x02) zeigt an, dass es ein *Length*-Feld gibt. Wenn das *LEN* Bit nicht gesetzt ist, wird die maximal mögliche *frame* größe verwendet. Mit maximaler *frame* größe sind die übriggebliebenen Bytes des Pakets gemeint.
- Das *FIN* bit (0x01) wird gesetzt, wenn der *stream* beendet wird.

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
Stream ID (i)																															
[Offset] (i)]																															
[Length (i)]																															
Stream Data (*)																															

Quelle: [39]

Tabelle 3.5.: Format des *stream frames*

Tabelle 3.5 zeigt den Aufbau eines *stream frames*. Ein *stream frame* hat folgende Felder:

- Eine *Stream ID*, welche durch einen *variable-length integer encoding* repräsentiert wird (siehe Abschnitt 3.2.2).
- Der *Offset* ist optional und wird durch das OFF Bit aktiviert. Der Wert des *Offsets* ist ein *variable-length integer encoding*. Der *Offset* wird in Bytes angegeben.
- Das *Length*-Feld ist optional und wird durch das *LEN* Bit aktiviert. Der Wert des *Length*-Feldes ist ein *variable-length integer encoding*.
- Das Feld *Stream Data* beinhaltet die zu übertragenden Nutzdaten.

Durch *streams* ist es möglich, Multiplexing [76] in QUIC-Verbindungen zu betreiben. Das *stream*-Multiplexing wird erreicht, indem unterschiedliche *stream frames* unterschiedlicher *streams*, also *stream frames* mit unterschiedlichen *Stream IDs*, in ein QUIC-Paket gepackt werden. Diese *streams* entfernen das HoL-Blocking-Problem, in dem im Fall eines Paketverlusts jeder *stream* separat behandelt werden kann [39]. Daher wird das HoL-Blocking-Problem auf die *stream*-Ebene verschoben. Dadurch ist es möglich, unterschiedliche *streams* unabhängig voneinander zu übertragen.

3.3. Beispiel eines 1-RTT Handshakes

Ein *1-RTT handshake* besteht aus zwei Paketen: dem von der Quelle gesendeten *initial*-Paket und dem vom Ziel als Antwort gesendeten *handshake*-Paket.

3.3.1. Initial Packet

Das *initial packet* wird verwendet, um die erste kryptographische Nachricht von der Quelle zum Ziel zu senden.

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
1	Type: 0x7F																														
>	Connection ID: 0xFFFFFFFFFFFFFFF1 (Zufällig)																														<
Version: 0xFFFFFFFF																															
Packet Number: 0x00000001																															
Type: 0x16										Stream ID (i): 0x00										Offset (i): 0x00											
Length (i): 0x65																															
Stream Data (*): 0xbaba ... // 500 Bytes füllend																															
Type: 0x00										Type: 0x00										Type: 0x00										... bis die 1200 Bytes erreicht sind	

Quelle: Eigenentwurf

Tabelle 3.6.: Beispiel eines *initial packets*

Tabelle 3.6 stellt den Aufbau eines *initial packets* da. In jedem Feld sind der Name des Feldes und ein Beispielwert zu sehen. Ein *initial packet* hat einen *long header* (gekennzeichnet durch das erste Bit), und als Typ wird *initial* gesetzt, was durch den Wert 0x7F (Hexadezimal) repräsentiert wird.

In dieser *initialen* Nachricht wird die *Connection ID* mit einer zufälligen, von der Quelle bereitgestellten Nummer gefüllt. Falls das Ziel zu diesem Zeitpunkt schon ein Paket gesendet hat, wird die vom Ziel bereitgestellte *Connection ID* verwendet.

Das Feld *Version* wird mit der von der Quelle spezifizierten Version des QUIC-Protokolls gesetzt. In der Tabelle 3.6 ist das 0xFFFFFFFF.

Die *Packet Number* wird beim *initial packet* zufällig generiert. Bei jedem folgenden Paket wird diese um eins inkrementiert.

Der *Payload* des *initial packets* ist ein *stream frame* mit der *ID 0*, einem *Offset* von 0 und einer *Length*, die der Länge der zu versendenden kryptographischen Nachricht entspricht. Der *stream* in diesem Paket muss immer einen *Offset* von 0 besitzen und die Nachricht muss in ein QUIC-Paket passen [39]. Das Paket muss mindestens 1200 Bytes groß sein. Falls das nicht der Fall ist, muss es mit *PADDING-Frames* aufgefüllt werden (In der Tabelle 3.6 als Typ 0x00 zu sehen).

3.3.2. Handshake Packet

Das *handshake packet* wird benutzt, um eine kryptographische *handshake*-Anwortnachricht zu senden. Es benutzt einen *long header* mit dem Typ 0x7D (Hexadezimal). Die sonstigen Informationen gleichen denen des *initial packet*. Denn es wird eine Antwort auf die kryptographische Initial-Nachricht zurück gesendet.

4. Anforderung

Dieses Kapitel behandelt die Anforderungsdefinition für die Realisierung der Implementierung. Die Anforderungen werden in zwei Bereiche gegliedert, funktionale und nichtfunktionale Anforderungen. Funktionale Anforderungen definieren, was die Software leisten soll. Nichtfunktionale Anforderungen definieren, was jenseits von Funktionalität beachtet werden muss, um ein gute Lösung zu erstellen.

4.1. Funktionale Anforderungen

Die funktionalen Anforderungen an das Softwaresystem werden nach dem Vorgehen von "Mastering the Requirements Process„[61] aufgelistet. Die Anforderungen werden nummeriert und beschrieben. Weitere Kriterien wie z.B. Reihenfolge und Kundenzufriedenheit werden hier außen vor gelassen, da sie keine weitere Relevanz für diese Arbeit besitzen.

Die hier aufgeführten funktionalen Anforderungen sind außerdem als *Issues* auf GitHub eingestellt. Damit ist der Entwicklungsprozess des zu realisierenden QUIC-Verbindungsaufbaus transparent dokumentiert.

Identifikator	1
Beschreibung	Es soll ein Initial-Paket erstellt werden.
Quelle	*draft*
Abnahmekriterium	Es existiert ein Initial-Paket, welches den *draft*-Spezifikationen entspricht.

Tabelle 4.1.: Anforderung für ein Initial-Paket

Identifikator	2
Beschreibung	Es soll möglich sein, einen *stream* 0 zu erstellen, über den eine *pseudokryptographische*-Nachricht gesendet werden kann.
Quelle	*draft*
Abnahmekriterium	Es muss möglich sein, einen *stream* 0 zu erstellen, welcher weitere Daten übermitteln kann

Tabelle 4.2.: Anforderung einer *stream*-Erstellung

Identifikator	3
Beschreibung	Implementieren der allgemeinen Umgebung für eine QUIC-Implementierung in OMNeT++ z.B. die *socket*-Erstellung und die State-Maschine
Quelle	OMNeT++ und Forschungsgruppe
Abnahmekriterium	Es muss eine allgemeine Grundstruktur vorhanden sein, so dass Pakete von einem Host zu einem anderen Host transferiert werden können.

Tabelle 4.3.: Anforderung für die Erstellung der allgemeinen Implementierung notwendigen Strukturen für OMNeT++

Identifikator	4
Beschreibung	Es soll möglich sein, Pakete mit einem *long header* zu versenden.
Quelle	*draft*
Abnahmekriterium	Es muss möglich sein, einen *long header* zu erstellen.

Tabelle 4.4.: Anforderung für eine *long header*-Nachricht

Identifikator	5
Beschreibung	Es soll möglich sein, einen *1-RTT handshake* Verbindungsaufbau in einer OMNeT++-Simulation durchzuführen.
Quelle	*draft*
Abnahmekriterium	Es muss möglich sein, in einer OMNeT++-Simulation den *1-RTT handshake* durchzuführen.

Tabelle 4.5.: Anforderung eines funktionierenden *1-RTT handshakes*

4.2. Nichtfunktionale Anforderungen

Außerdem sind nichtfunktionale Anforderungen bei diesem Projekt von großer Bedeutung. Im Folgenden werden nichtfunktionale Anforderungen aus dem Standard ISO 25010 [54] verwendet. Die für diese Arbeit relevanten Kategorien sollen hier auf dieses Softwareprojekt angewandt werden.

4.2.1. Wartbarkeit / Änderbarkeit

Die Änderbarkeit ist die wichtigste nichtfunktionale Anforderung an das System. Da die Software ein Teil eines Forschungsprojektes ist und von mehreren Entwicklern bzw. Forschern entwickelt und weiterentwickelt werden soll, muss sie möglichst einfach und effizient implementiert und erweiterbar sein, so dass die Einarbeitung von neuen Mitgliedern der Forschungsgruppe auf ein notwendiges Minimum reduziert werden kann. Des Weiteren basiert diese Arbeit auf einem *draft*, welcher sich gegebenfalls zukünftig ändern kann, weswegen eventuell häufig Änderungen notwendig werden. Durch diese Faktoren ist die Wartbarkeit und Änderbarkeit die wichtigste nichtfunktionale Anforderung.

4.2.2. Richtigkeit

Die Implementierung sollte sich korrekt nach den im *draft* spezifizierten Strukturen verhalten. Nur durch eine richtige Implementierung ist es möglich, verlässliche Aussagen darüber zu treffen, wie sich das Protokoll in der Realität verhält.

4.2.3. Performance

Das zu entwickelnde System sollte Simulationsszenarien schnell berechnen können, damit auch komplexe Simulationsszenarien überhaupt in realistischer Zeit berechenbar sind. Es sollte also auf eine hohe technische Performance geachtet werden, damit die Simulationszeiten nicht unverhältnismäßig lang dauern.

4.2.4. Effizienz

Die Effizienz ist notwendig, um größere Netzwerke sicher und valide simulieren zu können. Da in der Regel Simulationssoftware eingesetzt wird, wenn es ansonsten zu aufwendig wäre, ein reales Netz mit diesen Maßen einzusetzen, muss davon ausgegangen werden, das größere oder komplexere Netze simuliert werden sollen. Durch die Größe wächst auch der Ressourcenverbrauch, wenn damit nicht effizient umgegangen wird. Deshalb ist es wichtig, die Software effizient zu entwickeln.

4.2.5. Benutzbarkeit

Die entwickelte Implementierung sollte für jeden Benutzer von OMNeT++ verständlich nutzbar sein, so dass auch ohne Einblick in den konkreten Source Code des Projektes Testumgebungen aufgebaut werden können.

5. Entwurf und Entwicklung des Simulationsmodells für QUIC

Dieses Kapitel behandelt die Umsetzung der Implementierung, die Anforderungen, die Anpassung der Dokumentation sowie die Anpassungen des Entwurfs, die im Laufe des Entwicklungsprozesses notwendig waren.

5.1. Arbeitsgruppe

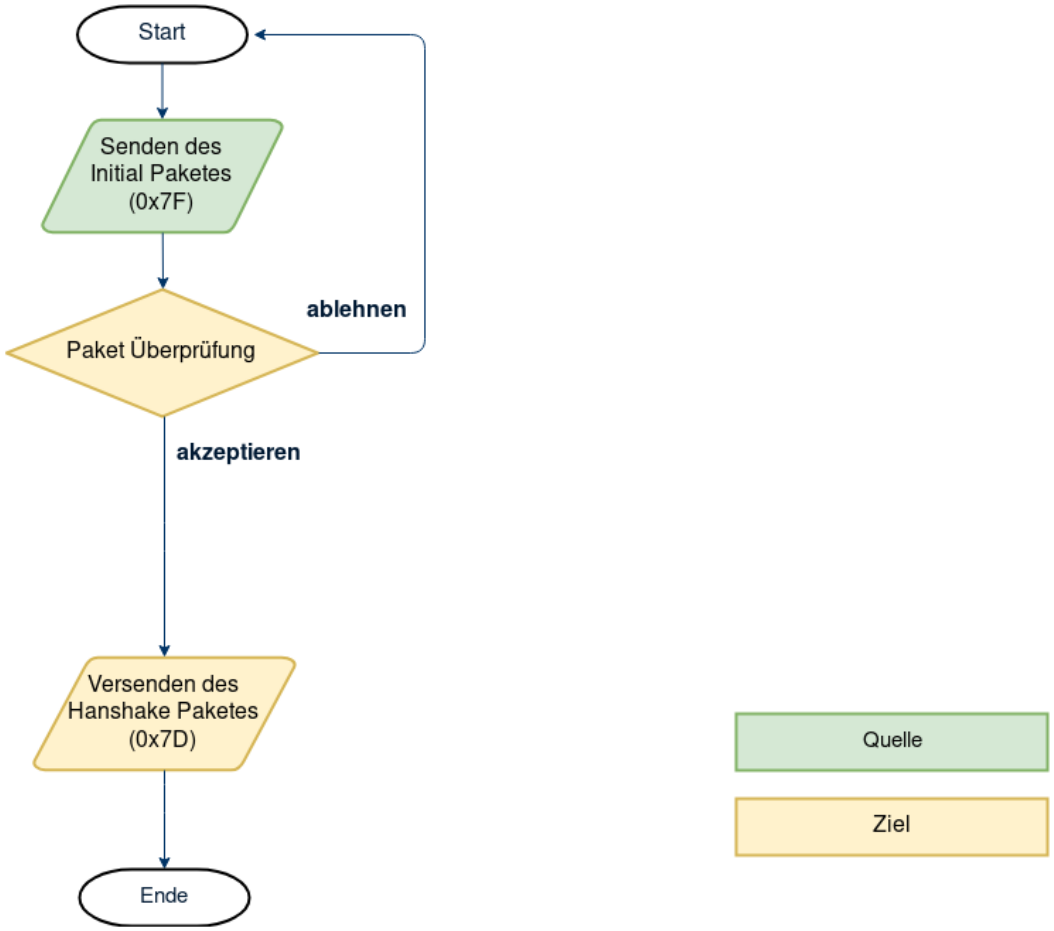
Teile der hier vorgestellten Entwicklung wurden von Prof. Dr. Martin Becke, Denis Lugowski und Marvin Butkereit entwickelt. Beteiligt an dieser Arbeit ist die Arbeitsgruppe QUIC der Hochschule für Angewande Wissenschaften (HAW), die Teil des Communication and distributed Systems (CADS) Forschungsprojektes von Prof. Dr. Martin Becke ist. Im Folgenden soll eine Übersicht dieser Entwicklung gegeben werden.

5.2. Anpassung des Entwurfs

Der Entwurf wurde im Laufe des Arbeitsprozesses nach Scrum [65] iterativ angepasst und kommuniziert. Dies wurde notwendig, da sich durch die Implementierung und das erneute Studium des *drafts* 10 [39] Unstimmigkeiten mit der modellierten Version ergaben. Beispielsweise beinhaltete die erste Implementierung eine *StreamDispatcher*-Klasse, welche das Aufteilen von *streams* bewerkstelligen sollte. Da aber nicht *streams* sondern *frames* die kleinste operative Einheit darstellen, fiel dieses Implementierungsdetail im Entwicklungsprozess als fehlerhaft auf und musste korrigiert werden. In der korrigierten Implementierung, werden nicht *streams* auf Basis ihres Typen differenziert, sondern *frames*.

5.2.1. Entwurf

Dies ist der zu implementierende Grundentwurf des *1-RTT handshakes* auf Basis der Anforderungen (siehe 4 für die Anforderungen) und der im *draft* 10 [39] enthaltenen Spezifikationen.



Quelle: Eigenentwurf auf Basis von Denis Lugowski A.2

Abbildung 5.1.: Design für die *handshake*-Funktionalität

Abbildung 5.1 stellt den *1-RTT handshake* dar. Ein *initial packet* (siehe Abschnitt 3.6) wird von der Quelle versendet. Beim eintreffen des Pakets beim Ziel folgt die Überprüfung, ob es ein *initial packet* ist. Wenn nicht, wird das Paket verworfen [39]. Bei einer Akzeptanz wird danach vom Ziel ein *handshake packet* zurückgeschickt, wie in der Abbildung 5.1 dargestellt. Damit ist der *1-RTT handshake*-Prozess beendet. Aus der Modellierung ist ersichtlich, dass keine *Version Negotiation* implementiert werden soll. Es wurde auf die Implementtierung dieses Aspektes auf Grund seiner geringen Relevanz für diese Simulationsimplementierung verzichtet. Von

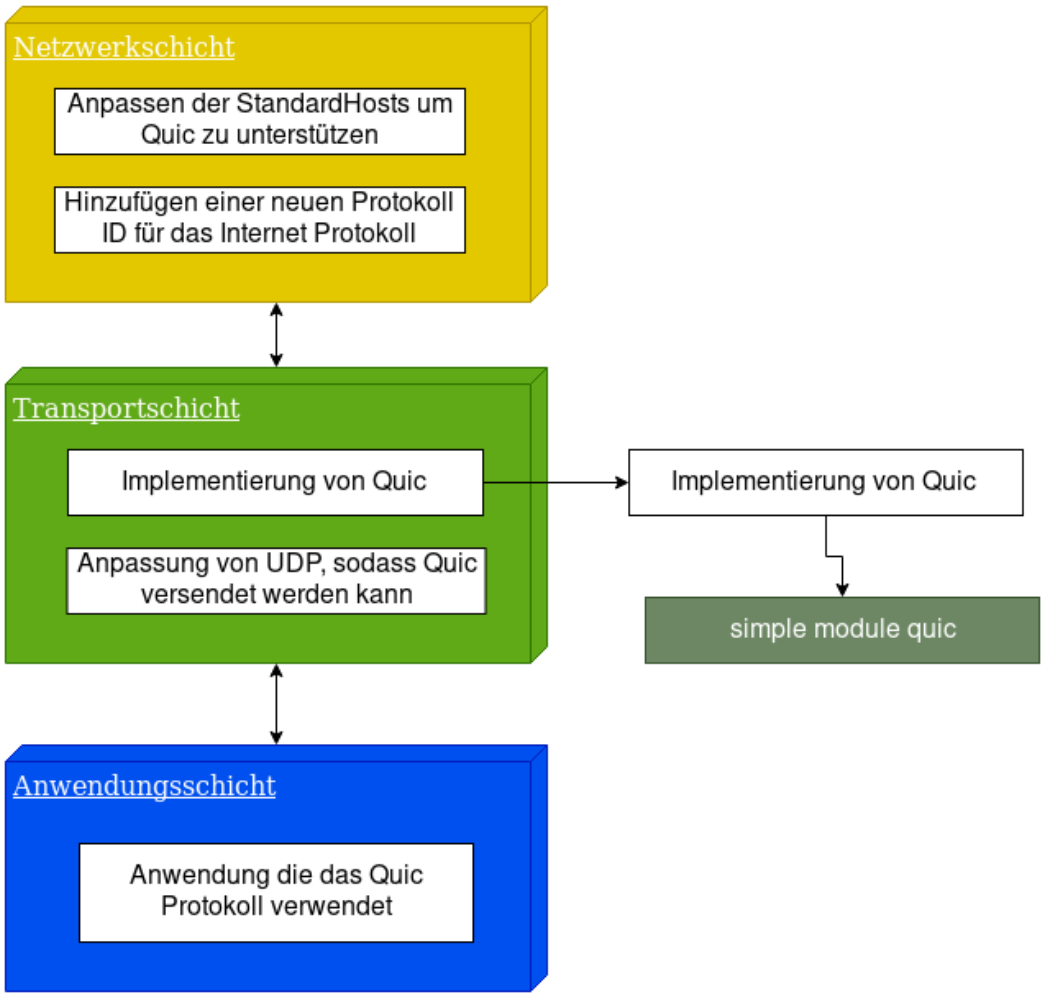
primärem Interesse ist hier der *1-RTT handshake*, auf dem viele weitere Mechanismen aufbauen [39].

5.2.2. OMNeT++

Nachdem ein Grundmodell für den *1-RTT handshake* entworfen wurde, müssen die notwendigen Entwürfe für OMNeT++ realisiert werden. Zuerst werden die notwendigen Komponenten definiert und anschließend der Automat [33] entworfen, welcher die Zustände einer QUIC-Verbindung modellhaft darstellen soll.

Komponenten

Es werden Komponenten aus drei Schichten des OSI-MODELLS [79] benötigt.



Quelle: Eigenentwurf

Abbildung 5.2.: Die unterschiedlichen Schichten, die bearbeitet werden müssen.

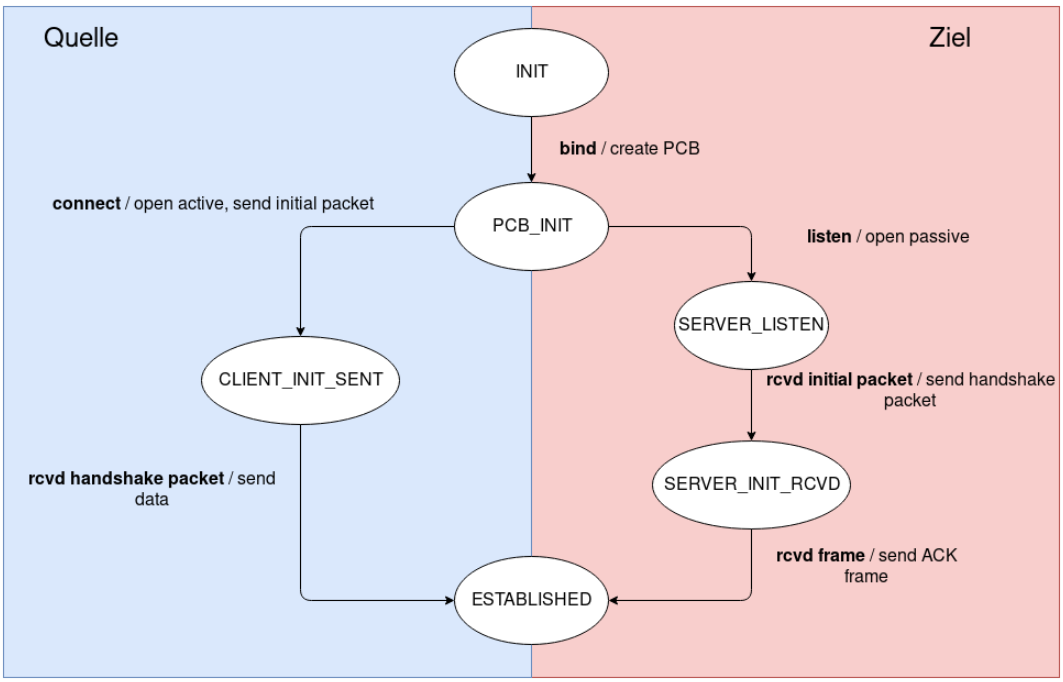
Abbildung 5.2 stellt die drei Schichten in abgetrennten Bereichen dar. Innerhalb jedes Bereichs werden die notwendigen Anpassungen definiert. Bei der Transportschicht zeigt der aus der Schicht ragende Pfeil rechts an, dass an dieser Stelle ein zusätzliches *Simple Module* im INET Framework hinzugefügt werden muss.

Simple Modules sind die kleinsten Modellelemente innerhalb des Framworks OMNeT++. Diese Elemente können nicht weiter aufgeteilt werden. Sie werden in C++ entwickelt und benutzen die OMNeT++-Simulationsbibliothek. Diese Module sind die aktiven Elemente in einem Modell

[75]. Als aktive Elemente werden in OMNeT++ Elemente genannt, die selbst direkt Nachrichten der Simulation verändern können. In der Modellierung in 5.2 ist QUIC ein Beispiele für ein *Simple Module*.

Automaten

Nach der Übersicht der zu implementierenden Teilbereiche soll nun der Ablauf des *1-RTT handshakes* durch eine Finite State Machine (FSM) modelliert werden. FSM sind eine Art von Automaten [33], die verwendet werden, um z.B. die notwendigen Zustände und Transitionen eines Verbindungsaufbaus zu modellieren [62].



Quelle: Eigenentwurf auf Basis von Denis Lugowski A.1

Abbildung 5.3.: Design-Auschnitt für die State Machine

Abbildung 5.3 zeigt einen für den *1-RTT handshake* zu implementierenden Ausschnitt der FSM. Die runden Ausschnitte repräsentieren dabei die Zustände. Die dort enthaltenen Namen sind die Bezeichner der Zustände. Die Pfeile von einem Zustand zu einem anderen sind mögliche Transitionen des Zustandes, welche mit einem fett dargestellten Event versehen sind. Die Transitionen können also nur durchgeführt werden, wenn das Event auftritt. Neben den fett

markierten Events sind die Aktionen, die bei der Transition, also dem Übergang von einem Zustand in den nächsten, ausgeführt werden, zu sehen. Der Startzustand des Automaten ist der **INIT**-Zustand. Der Automat teilt sich in zwei Abschnitte auf: in den Abschnitt für ein Quelle-System (blau gekennzeichnet) und in den Abschnitt für ein Ziel-System (rosa gekennzeichnet). Beide Systeme besitzen ein eigenes Abbild der FSM, um ihren aktuellen Zustand zu verwalten.

Am Anfang wird eine Transition von dem **INIT**-Zustand zu dem **PCB_INIT**-Zustand mit einem **bind** Event hergestellt, wodurch die Aktion *create PCB*, einen Protocol Control Block (PCB) erstellt. Ein PCB ist eine Datenstruktur, welche die für das Protokoll notwendigen Informationen speichern kann, z.B. den aktuellen Zustand der Verbindung [69].

Vom Zustand **PCB_INIT** gibt es mit der Ausführung von **listen** die Möglichkeit, der FSM die Eigenschaften für ein Ziel-System zu geben oder durch ein **connect** die eines Quelle-Systems. Zuerst wird ein **listen**-Event ausgeführt, wodurch eine Transition in den **SERVER_LISTEN**-Zustand durchgeführt und danach auf das *initial packet* gewartet wird.

Sobald das Ziel-System, welches sich in dem Zustand **SERVER_LISTEN** befindet, das *initial packet* erhalt, welches vom Quelle-System gesendet werden muss, wird der Zustand zu **SERVER_INIT_RCVD** gewechselt und auf eine ankommende Nachricht von einem Quelle-System gewartet. Das Received (RCVD) in der FSM steht für erhalten. Mit einem Acknowledgement (ACK) *frame* in einem QUIC-Paket wird nach dem Erhalt einer Nachricht vom Quelle-System die Verbindung bestätigt. Ein ACK *frame* wird gesendet, um den Erhalt einer Nachricht zu bestätigen. Anschließend ist die Verbindung aufgebaut und das Ziel ist im **ESTABLISHED**-Zustand.

Soll einer FSM die Eigenschaften für eine Quelle gegeben werden, wird beim **PCB_INIT**-Zustand ein **connect**-Event ausgeführt. Dadurch wird automatisch ein *initial packet* an das Ziel-System gesendet, außerdem wird in den Zustand **CLIENT_INIT_SENT** gewechselt. In diesem Zustand wird auf ein *handshake packet* vom Ziel-System gewartet. Wenn dieses Paket am Quelle-System eintrifft, können Daten über die Verbindung gesendet werden, da die Verbindung damit aufgebaut ist und das Quelle-System die Transition in den **ESTABLISHED**-Zustand durchgeführt hat.

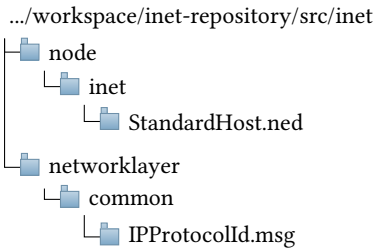
Damit ist der *1-RTT handshake* für eine Verbindung vollständig durchgeführt. Eine komplette Übersicht über den QUIC-Automaten ist in Anhang A.1 zu finden.

5.3. Umsetzung

In diesem Abschnitt wird die Verzeichnisstruktur der Implementierung von QUIC als ein OMNet++-Transportprotokoll-Modell beschrieben. Als Grundlage wird das INET Framework verwendet, da diese Bibliothek schon Protokolle wie TCP und SCTP bereitstellt und einen guten Ausgangspunkt für die Ergänzung um ein neues Protokoll darstellt. Das INET Framework ist so strukturiert, dass im Hauptverzeichnis für jede Schicht des OSI-MODELLs [79] ein Unterverzeichnis existiert, in dem die jeweiligen Protokolle dieser Schicht als Modell implementiert sind.

Im Folgenden wird die Änderung an jeder Schicht separat gezeigt. Im Sinne der Übersichtlichkeit werden nur Ausschnitte der benötigten Verzeichnisstruktur gezeigt. Ein kompletter Verzeichnisbaum ist in Anhang A.1 zu finden.

5.3.1. Netzwerkschicht



Quelle: Eigenentwurf

Abbildung 5.4.: Ordnerstruktur für die Änderungen auf der Netzwerkschicht

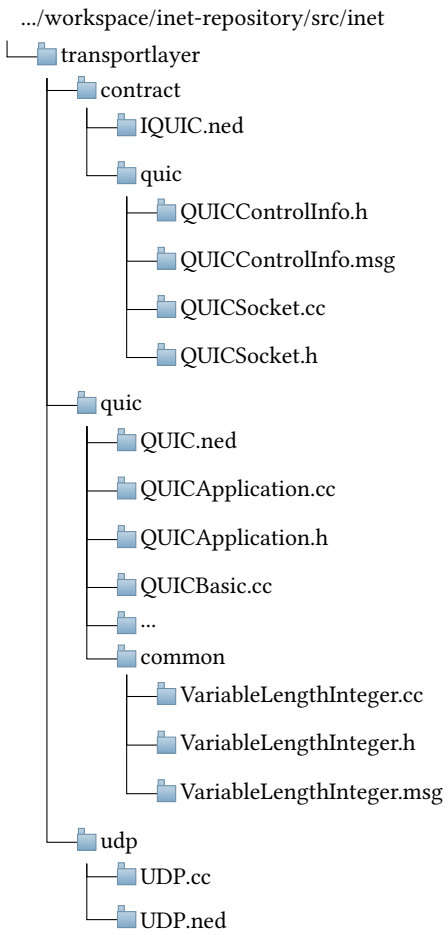
Um ein neues *Simple Module* für QUIC mit den bestehenden INET Framework-Modellen verwenden zu können, benötigen diese einige Anpassungen. Abbildung 5.4 zeigt die Dateien der Netzwerkschicht, die für die Realisierung angepasst werden müssen.

Das OMNeT++-Framework bietet eine Beschreibungssprache für Simulationsmodelle die sogenannte Network Description (NED). Mithilfe dieser können Modelle definiert und konfiguriert werden. Es können auch experimentelle Szenarien aufgebaut werden. Im Allgemeinen wird diese Sprache verwendet, um Modelle zu beschreiben [74]. Gekennzeichnet werden diese Dateien mit dem Suffix **.ned*.

Die *StandardHost.ned* wird um QUIC erweitert, um Kommunikation über das QUIC-Protokoll

zu ermöglichen. Ferner wird im IP-Protokoll *IPProtocolId.msg* eine ID hinzugefügt. Diese Änderung macht den QUIC-Datenverkehr von dem normalen UDP-Datenverkehr unterscheidbar. Hier wird die ID 133 genommen, die noch nicht im Framework verwendet wird und daher unproblematisch verwendet werden kann. Jedoch sollte die ID in Zukunft geändert werden, da sie für ein anderes Protokoll und zwar Fibre Channel (FC) [8], vorgesehen ist [35]. Als Vorschlag wäre eine Zahl aus dem Bereich 143-252 zu nehmen, wo sich noch unzugeordnete IDs befinden [35]. Alternativ wäre es auch möglich, eine ID aus dem Bereich 253-254 [52] zu nehmen. Diese sind für Experimente vorgesehen. Da dieser Bereich aber ziemlich klein ist und wahrscheinlich schnell Konflikte mit anderen experimentallen Implementierungen entstehen können, ist der unzugewiesene Bereich zu präferieren.

5.3.2. Transportschicht



Quelle: Eigenentwurf

Abbildung 5.5.: Ordnerstruktur für die Änderungen auf der Transportschicht

Abbildung 5.5 zeigt einen Ausschnitt der für die Implementierung des QUIC-Modells notwendigen, zu ändernden und hinzuzufügenden Dateien. In der Transportschicht wird das Protokoll UDP geändert. Konkret wird es um die Funktionalität erweitert, dass QUIC als Transportprotokoll zugewiesen werden kann, wenn der Parameter *isQuic* gesetzt ist. Dafür wird die Datei *UDP.ned* um diesen Parameter erweitert. Der *Quellcode* wird entsprechend so angepasst, dass das IP-Protokoll, welches Teil der Netzwerkschicht ist, mit dem Parameter *isQuic* auf *QUIC (133)* anstelle von *UDP (17)* gesetzt wird.

Damit sind alle notwendigen Änderungen in den unteren Schichten durchgeführt und es folgt die Anpassung und Implementierung auf der Transportschicht, die für ein QUIC-Protokoll-Modul benötigt werden. Dafür wird zuerst die *contract information* in dem Verzeichnis *contract* hinzugefügt. Es wird dort ein *module interface IQUIC (IQUIC.ned)* implementiert, welches die Schnittstelle bereitstellt, um die Kommunikation von über und unter der Transportschicht liegenden Schichten entgegenzunehmen.

Darüber hinaus wird in einem Unterordner *quic* die Datei *QUICControlInfo.msg* angelegt, die *ControlInformation* enthält, welche in *QUICSocket (QUICSocket.cc/.h)* verwendet werden, um Nachrichten des Zustandes des *sockets* an die QUIC-Implementierung zu senden. *Sockets* ist eine Programmierschnittstelle die von Forschern der *University of California at Berkeley* geschrieben wurde [72]. Die Schnittstelle ermöglich es mit Transportprotokollen zu arbeiten und einfach Daten über diese zu verschicken [72].

In OMNeT++ wird diese *ControlInformation* benutzt, um eine Kommunikation zwischen Protokollschichten zu ermöglichen. Bei dieser Kommunikation werden weitere Informationen neben den eigentlichen Paketinhalten benötigt, zum Beispiel eine *destination*-IP-Adresse beim Transport eines UDP-Paketes. Diese Informationen werden in OMNeT++ in *ControlInfo*-Objekte geschrieben und mitgesendet [74].

Anschließend wird das im *contract* definierte *interface IQUIC* im Verzeichnis *transportlayer/-quic* implementiert. Die dort erstellte Datei *QUIC.ned* stellt das *Simple Module QUIC_Basic* bereit, welches die Funktionalitäten für QUIC implementiert. Es besteht aus einer Definition in der NED-Datei sowie einer Implementierung in C++ (*QUICBasic.cc*), die durch ein Makro (*Define_Module(QUIC_Basic)*) miteinander verknüpft werden.

Dieses *Simple Module* wird zusammen mit dem *Simple Module* UDP und dem Parameter *isQuic* zu einem *compound module QUIC* zusammengeführt. Dieses implementiert das im Verzeichnis *contract* erstellte *interface IQUIC*.

Ein *compound module* gruppiert bestimmte Module zu eine größere Einheit. Das Modul kann wie ein *Simple Module gates* und *parameter* haben, es besitzt aber kein aktives, mit dem *compound module* verbundenes Verhalten [74].

Die *gates* sind im OMNeT++-Framework-Verbindungspunkte für Module [74]. Es existieren

drei unterschiedliche Typen von *gates*: *input*, *output* und *inout*. Ein *output gate* kann nur mit einem *input gate* verbunden werden und umgekehrt. Eine Ausnahme ist das *inout gate*, dort ist beides möglich. Durch diesen *gates*-Mechanismus ist es möglich, unterschiedliche Module zusammenzuschalten.

Diese Architektur ermöglicht es, QUIC zukünftig auch über andere Protokolle als UDP ohne größere Aufwände zu implementieren, indem einfach das *submodule UDP* mit einem anderen Protokoll ausgetauscht werden kann.

Im Folgenden werden die für die Durchführung eines *1-RTT handshakes* benötigten Nachrichten behandelt. Diese Nachrichten werden mit *message definition* in INET Framework modelliert und in *MSG*-Dateien geschrieben. Dort kann durch eine kompakte Syntax der Inhalt von Nachrichten definiert werden, ohne dass *Quellcode* geschrieben werden muss. OMNeT++ interpretiert diese Dateien und erspart manuellen Entwicklungsaufwand an dieser Stelle [74].

Insgesamt gibt es im QUIC-Protokoll-Modell drei *MSG*-Dateien, die Nachrichten für den *1-RTT handshake* definieren: *QUICPacket.msg*, *VariableLengthInteger.msg* und *QUICStream.msg*.

```
24
25 enum LongHeaderType {
26     PACKET_TYPE_INITIAL = 0x7F;
27     PACKET_TYPE_RETRY = 0x7E;
28     PACKET_TYPE_HANDSHAKE = 0x7D;
29     PACKET_TYPE_0RTT_PROTECTED = 0x7C;
30 };
31
32 //
33 // Represents an QUIC Long Header Packet.
34 // TODO: Extract same attributes for short and long header into a common struct.
35 //
36 packet QUICLongHeaderPacket
37 {
38     @customize(true);
39     // TODO: Temporary solution for LongHeaderType
40     uint8_t longHeaderType @enum(LongHeaderType);
41     uint64_t connectionID;
42     uint32_t version;
43     uint32_t packetNumber;
44 }
45
46 // TODO: Own packet definition for short header
47
```

Quelle: Der *Quellcode* der Implementierung

Abbildung 5.6.: Das *Long Header*-Paket in OMNeT++

Die Datei *QUICPacket.msg* definiert die *LongHeader Types* (siehe Abschnitt 3.2.3) und das *QUICLongHeaderPacket*, wie in Abbildung 5.6 dargestellt. Die Felder des Pakets entsprechen den Vorgaben des *drafts* [39].

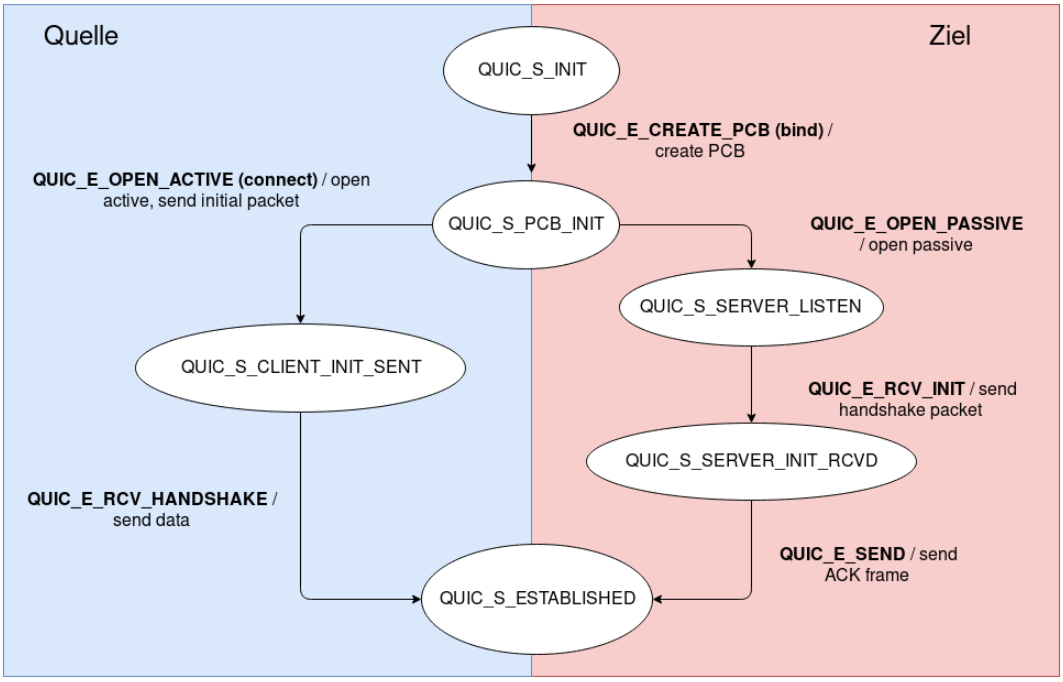
Die Modellierung des *long headers* verwendet ein *packet*, ohne eine Vererbung durchzuführen. Diese Implementierungsweise wird gewählt, da es für den *1-RTT handshake* keine Notwendigkeit gibt, die Ähnlichkeiten zum *short header* [39] zu generalisieren. Der Grund ist, dass er in dieser Implementierungsphase noch nicht implementiert wird.

Die Datei *VariableLengthInteger.msg* definiert die unterschiedlichen Encodingvarianten, die in Abschnitt 3.2.2 vorgestellt wurden, und einen neuen *packet type* mit dem Namen *VariableLengthInteger*. Dieses Paket ermöglicht es, im Kontext des QUIC-Protokoll-Modells mit *VariableLength*-Werten zu arbeiten, ohne sich mit Encoding und Decoding beschäftigen zu müssen.

Die Datei *QUICStream.msg* stellt die *stream*- und *frame*-Pakete mit der Definition von *Stream*-

Types und *FrameTypes* bereit. Des Weiteren enthält sie Strukturen für *Frames* und *StreamFrame* sowie eine Definition des *packet FrameList*. Das *packet FrameList* erlaubt es, eine Liste an *frames* zu speichern, was benötigt wird, um diese an *streams* zu hängen (siehe Abschnitt 3.2.4). Es wurde nur eine *MSG*-Datei erstellt, um die Pflege einer unnötigen Menge kleiner *message definitions* in unterschiedlichen *MSG*-Dateien zu vermeiden.

Implementation der Funktionalität



Quelle: Eigenentwurf auf Basis von Denis Lugowski A.1

Abbildung 5.7.: Design-Auschnitt aus der Implementierung der State Machine

Nach dem Überblick der hinzugefügten Dateien soll nun ein beispielhafter Programmablauf der Implementierung eines *1-RTT handshakes* demonstriert werden. Eine FSM wird als unterstützendes Mittel verwendet. Der Automat ist genau so aufgebaut wie der in Abbildung 5.3 dargestellte Automat, nur sind dieses Mal die Events und Bezeichner der Zustände identisch mit der Implementierung. Der Aufbau einer *1-RTT*-Verbindung läuft folgendermaßen ab (Startzustand **QUIC_S_INIT**):

Um mit der FSM arbeiten zu können, muss der in *contract* definierte *QUIC Socket* verwendet werden. Dieser besitzt alle für den Verbindungsaufbau notwendigen *Service Primitives* [72], angelehnt an die Methoden der *sockets*-Programmierschnittstelle der *University of California at Berkeley*, wie *bind()*, *listen()*, *connect()*, *send()* und *close()* [72]. Eine *recv()* Methode ist nicht notwendig, da OMNeT++ dieses mit der Methode *handleMessage()* des Modells implementiert. Jede dieser Methoden setzt eine *ControlInformation* in die zu versendende Nachricht, die, wie

in Abbildung 5.7 zu sehen, die Event-Bezeichnung (der fett markierte Text an der Transition) beinhaltet. Die Modelle in OMNeT++ können nur mit Nachrichten kommunizieren, weshalb die oben angegebenen Methoden entweder Nachrichten empfangen oder versenden.

Um die FSM zu starten, muss ein *bind()* auf einen *QuicSocket* ausgeführt werden. Bei einem Aufruf der Methode *bind()* eines *QuicSockets* wird im Laufe der Behandlung die Klasse *QUIC-Basic* aufgerufen. Diese ist die erste für den Anwendungsfall interessante Klasse. Dort werden in der aufgerufenen Methode *handleMessage()* die eingehenden Pakete korrekt behandelt.

In *handleMessage()* wird überprüft, auf welchem *gate* eine Nachricht eingetroffen ist. Diese Nachricht wird durch eine *switch*-Anweisung zu dem *Quellcode* geführt, der diese Nachricht behandelt. Im Fall der Ausführung der *bind()* Methode wird innerhalb des dafür zuständigen Bereichs der *handleMessage()* Behandlung ein neues *QuicConnection*-Objekt erstellt und in einer *quicAppConnMap* gespeichert.

Eine *QuicConnection* wird für jeden Kommunikationspartner erstellt. Die *QuicConnection* beinhaltet die FSM , ein UDP-*socket* und verwaltet den weiteren Prozess der Verbindung. Sofern eine Verbindung eine *QuicConnection* besitzt, kann ihr Zustand über das individualisierte Abbild der FSM (siehe Abbild 5.7) gewechselt werden. Dies geschieht mit dem Aufruf von *processAppCommand* der *QuicConnection*.

Die *QuicConnection* verwendet eine *cFSM* von OMNeT++, um die FSM abzubilden. Die Benutzung des von OMNeT++ bereitgestellten *cFSM*-Mechanismus ermöglicht eine einfache Wartbarkeit dieses Aspektes des Modells.

Die *quicAppConnMap* ist eine von drei Datenstrukturen, die verwendet werden, um *QuicConnection* zu speichern. Es werden drei Datenstrukturen verwendet, da die FSM drei unterschiedliche Zustandsbereiche besitzt. Jeder dieser Bereiche tritt zu unterschiedlichen Zeitpunkten auf, zu denen unterschiedliche Verbindungsinformationen vorhanden sind. Die *quicAppConnMap* wird verwendet, um den Anfangsbereich der FSM zu ermöglichen, wo es noch keine Information über das zweite System im Verbindungsaufbau gibt. Die *quicInitialConnMap*, welche den *4-tuples* als Identifikator verwendet, wird benutzt, sobald sich die zwei Systeme (Quelle und Ziel) kennen, also Daten miteinander ausgetauscht haben. Diese *map* wird benötigt, solange der Prozess der Aushandlung einer *Connection ID* noch nicht vollzogen ist. Am Ende des Verbindungsaufbaus wird noch eine *quicConnMap* verwendet. Diese wird benötigt,

sobald die *Connection ID* fest steht, damit die Verbindung anhand dieser auch identifiziert werden kann. Dadurch wird es möglich, im weiteren Verlauf nach dem Verbindungsaufbau auch Funktionen wie die Verbindungs-Migration [39, p. 35] in Zukunft zu implementieren.

Außerdem wird nach dem Aufruf der Methode *bind()* und der Erstellung der *QuicConnection* eine Transition durchgeführt, die die FSM der *QuicConnection* in den Zustand **QUIC_S_PCB_INIT** überführt. Alle notwendigen Informationen werden erstellt, auf dem UDP-*socket* wird ebenfalls ein *bind()* ausgeführt und der *socket* wird der *QuicConnection*-Instanz hinzugefügt.

Dieser Vorgang ist für Quelle und Ziel identisch. Um nun eine Quelle zu erstellen, muss ein *connect()*-Aufruf über den *QuicSocket* durchgeführt werden. Damit wird ein *QUIC_E_OPEN_ACTIVE*-Event an die Methode *handleMessage()* von *QUICBasic* gereicht. Dort wird die Verbindung über die *quicAppConnMap* nachgeschlagen und das Event weiter bearbeitet.

In dieser Eventbehandlung im *Quellcode* wird ein *QUICPacket* erstellt. Danach wird auf dem UDP-*socket* der Verbindung ein *connect()* ausgeführt. Außerdem wird die Verbindung nun in die *quicInitialConnMap* gespeichert, welche als Schlüssel das *4-tuples* der Verbindung verwendet. Dies ist notwendig, da die Verbindung erst eine konstante *Connection ID* besitzt, nachdem sie in dem **QUIC_S_ESTABLISHED**-Zustand ist. Das erstellte *QUICPacket* ist ein *inititial packet*, welches im Folgenden genauer erörtert wird.

Feld	Wert
longHeaderType	PACKET_TYPE_INITIAL (0x7F)
connectionID	Zufällige Nummer
version	PSEUDO_VERSION_NUMBER (0xFFFFFFFF)
packetNumber	Zufällige Nummer

Quelle: Eigenentwurf

Tabelle 5.1.: *long header*-Werte beim initialen Paket

Tabelle 5.1 zeigt die Feldwerte des *long headers packets*, welches in der Eventbehandlung erstellt wurde und einem *initiale packet* (für weitere Informationen 3.6) entspricht. Der *long header* ist vom Typ 0x7F, was als Konstante *PACKET_TYPE_INITIAL* in der Nachrichtendefinition *QUICPaket.msg* gesetzt ist. Als *Connection ID* wird eine zufällige Nummer mit der Methode *getActiveEnvir* der Klasse *cSimulation* und der Methode *getUniqueNumber* der Klasse *cEnvir*

erstellt. Die Version des Protokolls ist in einer konstanten *PSEUDO_VERSION_NUMBER* mit dem zufälligen Wert 0xFFFFFFFF in der Klasse *QUICConnection* definiert. Der Wert für die *packetNumber* des Pakets wird ebenfalls zufällig durch eine Zufallsfunktion in der Methode *calculateInitialPacketNumber()* der *QuicConnection* Klasse erstellt.

In das *long header packet* wird eine *FrameList* injiziert. Diese *FrameList* ist der *Payload* des Paketes. Da die *FrameList* in dem Framework als ein *packet* modelliert ist, kann sie in eine vorhandene *packet*-Struktur integriert werden. Diese *FrameList* repräsentiert die in Abschnitt 3.2.4 beschriebene Liste von *frames* in der OMNeT++-Modellierung: eine *FrameList* ist ein *ByteArray*, in den die erstellten *frames* mit der Funktion *memcopy()* hineinkopiert werden.

```
64
65 packet FrameList {
66     ByteArray frameListArray;
67 }
68
69 // Contains frame header fields
70 struct Frame {
71     uint8_t type;
72 };
73
74 // Frame types with type-dependend fields
75 struct StreamFrame extends Frame {
76     VariableLengthInteger streamID;
77     VariableLengthInteger offset;
78     VariableLengthInteger length;
79 };
80
```

Quelle: Eigenentwurf

Abbildung 5.8.: *Frame*-Struktur in OMNeT++

Die *frames* bestehen aus einem *frame type* und einem *frame*-abhängigen Inhalt. Dies wird, wie in Abbildung 5.8 dargestellt, durch eine Struktur namens *Frame* repräsentiert. Als Beispiel soll der *StreamFrame* dienen. Dieser erweitert den *Frame* und erhält damit das notwendige Feld für den *frame type* und fügt in seiner *struct* (Datenstruktur) die *frame*-abhängigen Felder hinzu. Da für den *1-RTT handshake* zwei Typen von *frames* benötigt werden, wird eine Verallgemeinerung vorgenommen. Für das erstellte Paket wird ein *StreamFrame* angelegt, der die *streamID* 0 hat, und in diesem *frame* wird die kryptographische *handshake*-Nachricht transportiert. Es wird hier der *stream* 0 verwendet, da dieses vom *draft* 10 [39] vorgeschrieben wird (siehe Abschnitt 3.6).

Der *stream frame* ist dynamisch und kann von 0x10 bis 0x17 variieren. Der *stream frame* im *initial packet* hat den Typwert 0x16 (siehe Abschnitt 3.6). Da eine kryptographische Nachricht verschickt wird, werden das *LEN*-Bit (0x02) und das *OFF*-Bit (0x04) gesetzt, wie in der *draft* 10 [39] Spezifikation gefordert ist.

Die ersten 3 Bytes dieses Pakets sehen wie folgt aus: 0x16 (*Stream ID*), 0x00 (*Offset*), 0x65 (*Length*), wobei alle Informationen als *variable-length integer encoding* dargestellt sind.

```
Direkte Umwandlung von -12 / 0x65: 01 00 0001 1111 0100
Binaere: 00 0001 1111 0100
Hex: 0  0       1        F  4
Dezimal: 500
```

Listing 5.1: Umrechnung der *stream Length*

Der *stream* hat die ID 0, ein *offset* von 0 und besitzt wie in Listing 5.1 berechnet eine *LEN* von 500 Bytes, welche die Größe für den *Payload*-Bereich des *stream frames* definiert (weitere Information 3.2.4).

Da die Implementierung der kryptographischen TLS 1.3 in OMNeT++ noch nicht abgeschlossen ist und eine Implementierung den Rahmen dieser Arbeit überschreiten würde, wird eine *pseudokryptographische*-Nachricht als Inhalt des *Payloads* des *StreamFrame* verwendet, bestehend aus dem Wert in Abbildung 5.2.

```
Dezimal: -70
Binaer: 1011 1010
Hex: 0xba
```

Listing 5.2: Inhalt des *pseudokryptographischen handshakes*

Das *initial packet* muss mindestens eine Größe von 1200 Bytes besitzen und darf nicht größer als ein *QUIC packet* sein, da die Spezifikation dieses vorschreibt. Aus diesem Grund wird ein weiterer *frame type* benötigt.

Das *PADDING Frame* wurde in Abschnitt 3.2.4 erwähnt und besteht nur aus dem Typ-Feld, welches wiederum nur aus dem Wert 0x00 besteht. Deshalb wird die *FrameList* nach dem *stream frame* des hier erstellten *initial packets* mit Nullen aufgefüllt, bis 1200 Bytes erreicht werden.

Am Ende dieser Eventbehandlung wird das *QUIC packet* mit dem Event *QUIC_E_OPEN_ACTIVE* von der Quelle an das Ziel geschickt.

Anschließend wird der Zustand der Quelle zu **QUIC_S_CLIENT_INIT_SENT** gewechselt. An dieser Stelle wird auf der Seite der Quelle auf ein *handshake packet* vom Ziel gewartet.

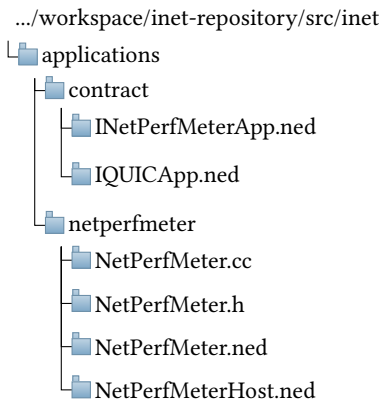
Das Ziel ist im Zustand **QUIC_S_PCB_INIT** und führt über den *QuicSocket* die Methode *listen()* aus, was dazu führt, dass in der Methode *handleMessage()* das Event *QUIC_E_OPEN_PASSIVE* ankommt und verarbeitet wird. Die Verbindung wird nun in einer *quicInitialConnMap* gespeichert, und eine Transition zu dem nächsten Zustand **QUIC_S_SERVER_LISTEN** wird durchgeführt. In diesem Zustand wird auf das Eintreffen des eben erörterten *initial packet* der Quelle gewartet.

Trifft diese Nachricht ein, wird ein Event *QUIC_E_RCV_INIT* erwartet, das in dem *ControlInformation*-Objekt der Nachricht enthalten ist. Alle anderen Nachrichten werden verworfen. Darauf wird das *handshake packet* nachdem erhalt des *initial packet* gesendet. Es besteht wie in Abschnitt 3.3.2 beschrieben aus einem *long header*, welcher als Typ *PACKET_TYPE_HANDSHAKE* den Wert 0x7D hat. Dieser ist als Konstante in der *QUICPacket.msg* definiert. Das Paket setzt eine vom Ziel generierte *Connection ID* und eine zufällige Paketnummer. Das Paket beinhaltet außerdem einen *stream frame*, welcher den *pseudokryptographischen handshake* zurückschickt. Das *handshake packet* sieht genau so aus wie das vorhin erörterte *initial packet* mit der Ausnahme des Types (siehe Abschnitt 3.3.2).

Dieses *handshake packet* wird vom Ziel an die Quelle versendet. Die Quelle wartet in dem Zustand **QUIC_CLIENT_INIT_SENT**, bis ein *handshake packet* ankommt.
Dann wird die Verbindung als **QUIC_S_ESTABLISHED** in eine *quicConnMap* gespeichert und aus der *quicInitialConnMap* entfernt.

Damit hat die Quelle ihren Verbindungsaufbau durchgeführt. Ab diesen Zeitpunkt entspricht die Implementierung nicht mehr genau dem *draft* 10 [39], da die Quelle nun einfach *long header packets* an das Ziel versendet. Dieses wurde so gewählt, damit ein Zustandswechsel des Ziels in den **QUIC_S_ESTABLISHED** möglich ist. Da das Ziel im Zustand **QUIC_S_SERVER_INIT_RCVD** auf ein ACK *frame* wartet und dieses noch nicht implementiert ist, kann keine Transition stattfinden. Deshalb reagiert die Implementierung auf die Pakete, die von der Quelle gesendet werden, und führt die letzte Transition des Verbindungsaufbaus durch. Damit ist der beispielhafte Ablauf eines Verbindungsaufbaus abgeschlossen.

5.3.3. Anwendungsschicht



Quelle: Eigenentwurf

Abbildung 5.9.: Ordnerstruktur für die Änderungen auf der Anwendungsschicht

Um einen Testaufbau zu ermöglichen, muss zusätzlich zu den Implementierungen von Netzwerk- und Transportschicht das QUIC-Protokoll auch in einer Anwendungsschichtmodellierung hinzugefügt werden. Dafür soll die Anwendung NetPerfMeter aus dem INET Framework für diesen Zweck verwendet werden, da es das Erstellen von Testaufbauten zur Untersuchung von Protokollkommunikation ohne viel Aufwand ermöglicht. Die Datei INetPerfMeterApp.ned wird um die *gates quicIn* und *quicOut* erweitert, und es wird ein *contract* in der Datei *IQUICApp.ned* angelegt, indem eine Interface-Definition mit den *quicIn* und *quicOut gates* vorgenommen wird.

Die Anwendung selbst wird um das Transportprotokoll QUIC erweitert. Das bedeutet, es wird ein *QuicSocket* eingebaut, welcher genutzt wird, sobald in der Konfiguration der Parameter für das Protokoll auf *quic* gesetzt wird.

6. Evaluation

Im Folgenden wird untersucht, wie sich eine *delay*-beeinträchtigte Verbindung bei QUIC im Verhältnis zu anderen verbindungsorientierten Protokollen bei einem Verbindungsaufbau verhält.

6.1. Evaluationsbeschreibung

Im Detail soll in diesem Experiment untersucht werden, wie sich der entwickelte QUIC-Verbindungsaufbau gegenüber TCP- und SCTP-Verbindungsaufbauten verhält, wenn sich das *delay* der Verbindung ändert. Dafür wird mithilfe der Tools aus dem OMNeT++-Framework eine Simulationsumgebung aufgebaut, die es ermöglicht, die Auswirkungen von unterschiedlichen *delays* zu evaluieren.

6.2. Erwartung

Es ist zu erwarten, dass sich die Dauer des Verbindungsaufbaus in einem Verhältnis zum *delay* verändert. Außerdem ist zu erwarten, dass QUIC durch die im Vergleich zu anderen Protokollen am geringsten ausfallende Nachrichtenanzahl am schnellsten sein wird. Um die Erwartungen zu verdeutlichen, können folgende Berechnungen angeführt werden:

```
1 QUIC
2
3 Anzahl der Leitungen * delay * Anzahl der Nachrichten =
     Totales Verbindungsaufbau-delay
4
5 Anzahl der Nachrichten: 2
6 delay: 5ms, 10ms 100ms
7 Anzahl der Leitungen: 2
8
9 2*5*2   =  20 ms
```

```
2*10*2  =  40 ms
2*100*2 = 400 ms
```

Listing 6.1: Berechnung der erwarteten Dauer eines Verbindungsaufbaus bei QUIC

```
TCP

Anzahl der Leitungen * delay * Anzahl der Nachrichten =
   Totales Verbindungsaufbau-delay

Anzahl der Nachrichten: 3
delay: 5ms, 10ms 100ms
Anzahl der Leitungen: 3

2*5*3   =  30 ms
2*10*3  =  60 ms
2*100*3 = 600 ms
```

Listing 6.2: Berechnung der erwarteten Dauer eines Verbindungsaufbaus bei TCP

```
SCTP

Anzahl der Leitungen * delay * Anzahl der Nachrichten =
   Totales Verbindungsaufbau-delay

Anzahl der Nachrichten: 4
delay: 5ms, 10ms 100ms
Anzahl der Leitungen: 4

2*5*4   =  40 ms
2*10*4  =  80 ms
2*100*4 = 800 ms
```

Listing 6.3: Berechnung der erwarteten Dauer eines Verbindungsaufbaus bei SCTP

Im antizipierten Ergebnis sollte QUIC bei einer idealen Leitung, also einer Verbindung ohne äußerliche Einflüsse, weniger *delay*-sensitiv sein und ein geringeres totales *delay* als TCP [57] aufweisen. Dieses Verhalten sollte auftreten, da QUIC für einen Verbindungsaufbau zwei Nachrichten anstatt drei (bei TCP) austauschen muss. Als der erwartete *delay*-anfälligste

Verbindungsaufbau in dieser Simulation wird SCTP [70] angenommen, da es vier Nachrichten für einen Verbindungsaufbau austauschen muss.

6.3. Verwendetes System

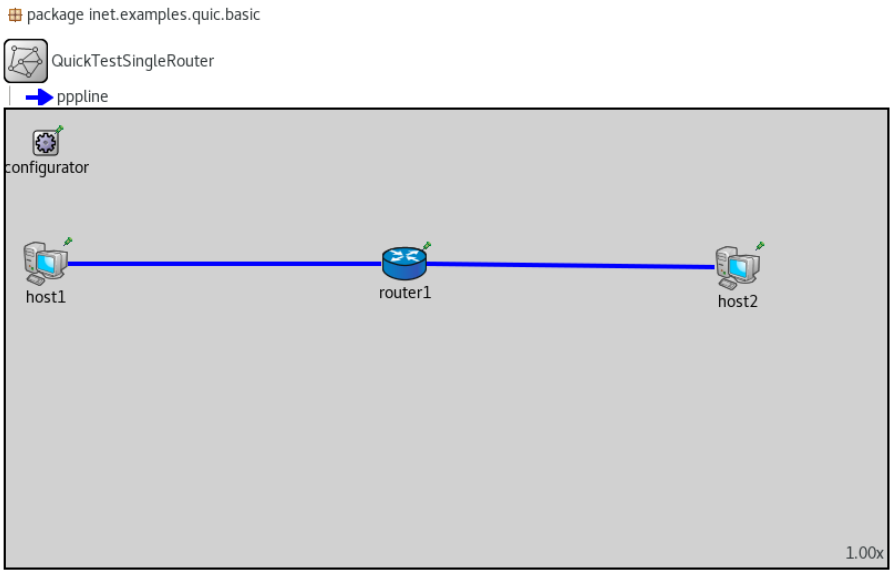
Als Evaluationsumgebung wird OMNeT++ in der Version 5.2.1.171211-da8f6bc verwendet, mit der INeT-Version mit dem Git Commit Hash 5a3bb2c3c4. Die Simulation wird auf einem Linux-System durchgeführt. Die Distribution ist Fedora in Version 29. Bei dem System, auf dem die Evaluation durchgeführt wird, handelt es sich um ein Acer Swift 3 mit Intel Core i5 8th Generation Prozessor.

6.4. Versuchsaufbau und -durchführung

Die hier *PPP* genannte Verbindung ist mit *delays* von jeweils 5ms, 10ms und 100ms konfiguriert. Dies ist der variable Faktor der Evaluation und variiert zwischen den Durchläufen. Die Datenrate beträgt konstant 12 Mbps. Die restliche Konfiguration entspricht den Standardparameterwerten, die in dem *DatarateChannel*-Modell definiert sind.

Für das Ziel und die Quelle werden die Parameter der *gates* konfiguriert, der Rest entspricht den Standardwerten des *NetPerfMeterHost*-Modells.

Als Router ist ein *Router*-Modell mit zwei Anschlüssen gewählt. Des Weiteren werden die Standardwerte des Modells genutzt.



Quelle: Eigenentwurf

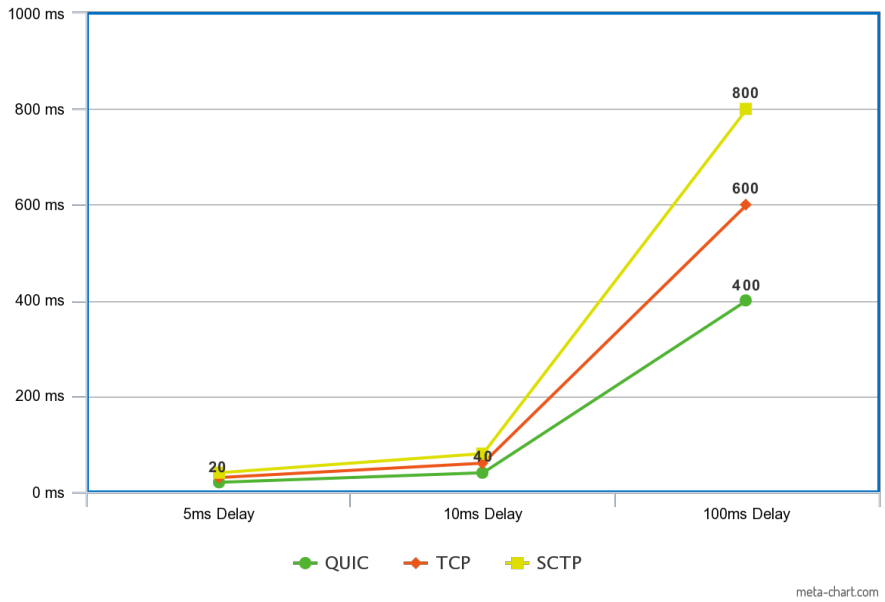
Abbildung 6.1.: Simulations Aufbau mit einem Router

Die Abbildung 6.1 zeigt das NED-Modell, das zur Evaluation aufgebaut wird (siehe Anhang A.1). Der Aufbau besitzt einen host1 als Quelle-System und einen host2 als Ziel-System. Die Quelle ist mit dem Router über eine PPP-Verbindung verbunden, und der Router ist mit dem Ziel ebenfalls über eine PPP-Verbindung verbunden.

In der zur Ausführung des Versuchs benötigten Initialisierungsdatei werden die Werte wie im Anhang A.2 dokumentiert gesetzt. Als variabler Faktor wird das *delay* der *PPP*-Verbindung verwendet.

6.5. Beobachtung

Die Durchführung dieses Versuchs mit den *delay*-Werten von 5ms, 10ms und 100ms ergibt folgendes Bild:



Quelle: Eigenentwurf

Abbildung 6.2.: Chart für die gemessenen Laufzeiten

Protokoll	Bei 5ms delay	Bei 10ms delay	Bei 100ms delay
QUIC	20ms	40ms	400ms
TCP	30ms	60ms	600ms
SCTP	40ms	80ms	800ms

Quelle: Eigenentwurf

Tabelle 6.1.: Gemessene Verbindungsaufbauzeiten

6.6. Auswertung

Auf Basis der gemessenen Werte lässt sich feststellen, dass sich QUIC mit seinem Verbindungsaufbau in dem hier evaluierten Beispiel im Verhältnis zu TCP und SCTP in dieser Simulation deutlich weniger anfällig für *delay*-Änderungen zeigt. Dies entspricht den formulierten Erwartungen: Die unterschiedliche Nachrichtenanzahl der verschiedenen Protokolle korrespondiert beim Verbindungsaufbau mit ihrer relativen Leistung - die Protokolle mit höheren Mengen von Nachrichten weisen eine größere *delay*-Anfälligkeit auf. Es kann gezeigt werden, dass eine Reduktion der Nachrichten in dem hier evaluierten Fall zu einer Optimierung der Verbin-

dungsaufbauphase führt. Daraus folgt, das Nutzdaten zeitlich schneller angefangen werden zu verschicken. Es kann außerdem gezeigt werden, dass QUIC in diesem Fall weniger *delay-anfällig* ist als vergleichbare verbindungsorientierte Protokolle wie TCP und SCTP.

Darüber hinaus die Evaluation, dass sich diese *delay*-Sensibilität bei größeren *delays* stärker negativ darstellt als bei geringeren *delays*, da das gemessene Wachstum ein Vielfaches der Menge der Nachrichten der simulierten Verbindungsaufbauten ist.

In der Abbildung 6.2 wirkt es so, als würden sich die Werte annährend exponentiell verhalten, dies ist aber nicht der Fall. Dieses Bild entsteht nur, da der letzte Versuchswert deutlich größer ist als die vorhergegangenen Messwerte. Das Ergebnis sollte sich linear, also auch proportional zum *delay* verhalten. Dieses kann aus der oben ausgeführten Rechnung geschloßen werden, da *total = Anzahl der Leitungen*delay*Anzahl der Nachrichten.* Dabei sind die *Anzahl der Nachrichten* und die *Anzahl der Leitungen* konstant. Daraus folgt: Die Dauer der Verbindungsaufbauten muss sich wie eine lineare Funktion verhalten.

7. Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit resümiert und ein Ausblick auf weitere Aufgaben gegeben.

7.1. Zusammenfassung

Das Ziel dieser Arbeit war es, den *1-RTT handshake* (Verbindungsaufbau) von QUIC auf Basis des *drafts* [39] in OMNeT++ abzubilden. Dafür wurde ein OMNeT++-Simulationsmodell entwickelt. Die Anforderungen wurden definiert und anschließend als Grundlage der Implementierung verwendet. Das Simulationsmodell wurde anhand von Automaten entworfen, die auf Basis des *drafts* [39] entwickelt wurden. Anschließend wurde eine Evaluation durchgeführt.

In der Evaluation wurden QUIC, SCTP und TCP auf ihre simulierten Laufzeiten beim Verbindungsaufbau mit unterschiedlichen *delay*-Varianten untersucht. Es zeigte sich in dieser Evaluation, dass QUIC eine geringere Anfälligkeit für *delays* aufweist als SCTP und TCP. Dieses Verhalten gilt nur in der simulierten Umgebung und lässt keine Verallgemeinerungen oder verallgemeinerten Schlüsse zu. Da in einer Simulation keine außerlichen Störfaktoren vorhanden sind, die das Ergebnis beeinflussen könnten.

Es konnte also gezeigt werden, dass es unter bestimmten Umständen möglich ist, durch das Verwenden von QUIC als Transportprotokoll theoretisch einen schnelleren Webseitenaufruf zu ermöglichen. Der Grund ist, dass der Verbindungsaufbau weniger *delay*-sensitiv ist als andere aktuelle Protokolle. Dies gilt nur theoretisch, da in der Praxis auch noch andere Protokolle benötigt werden, um mit den Webbrowsern interagieren zu können.

7.2. Herausforderungen

In diesem Abschnitt behandelt und reflektiert die Herausforderungen dieses Projektes.

Um dieses Projekt umsetzen zu können, musste ein Überblick der Struktur des INET Frameworks geschaffen werden, um herauszufinden, an welchen Stellen welche Informationen abgelegt werden. Dies ist eine sehr aufwendige Arbeit, da kaum Dokumentation in diesem Teilbereich vorhanden ist. Darüber hinaus wurde geschaut, wie andere Transportprotokoll-Entwicklungen realisiert wurden. Anhand dieser Beispiele wurde die QUIC-Implementierung realisiert.

Auch ist die Implementierung von QUIC in OMNeT++ keine Individualentwicklung gewesen, sondern eine Arbeitsgruppenentwicklung. Das bedeutet, dass vor allem am Anfang der Entwicklung ein großer Kommunikationsaufwand entsteht, um Sachverhalte zu klären und abzustimmen, damit ein Konsens bei der Implementierung vorliegt und keine größeren unabgesprochenen Entwicklungen gemacht werden.

Außerdem ist die Entwicklung an einem Protokoll, welches noch in der *draft*-Phase ist, relativ herausfordernd, da der *draft* so noch nicht implementiert wurde. Natülich gibt es aktuelle Implementierungen des hier intressierenden *drafts*, aber diese wurden entwickelt, um selbst herauszufinden, ob es noch offene Fragen zu den Spezifikationen gibt. Dadurch sind diese Quellen nicht als verlässlich anzusehen. Dieses Fehlen an Informationsmaterial und unpräziser Beschreibung führt dazu, dass die Entwicklung sehr langwierig wird, da vieles immer wieder nachgelesen werden und ggf. anhand von anderen bereits vorhandenen Implementierungen verifiziert werden muss..

7.3. Ausblick

Der Entwicklungsprozess der Implementierung von QUIC in OMNeT++ ist noch nicht abgeschlossen und die Spezifikationen des Protokoll-Entwurfs sind noch nicht final. Es ist möglich, dass gegebenenfalls kleine und größere Änderungen an der Grundstruktur der Implementierung vollzogen werden müssen.

Die Definitionen der *long* und *short header* sollten in einem weiteren Arbeitsschritt zusammengeführt werden, um einen besser lesbaren und wartbaren *Quellcode* zu erhalten.

Der *Quellcode* sollte an die im neuen *draft* 12 [40] entstandenen Änderungen an z.B. *long* und *short header*, angepasst werden, um die neuesten Änderungen an der Spezifikation testbar zu machen. Die große Herausforderung, die damit einhergeht, ist der zeitliche Aufwand, den

neuesten Entwicklungen Rechnung zu tragen und die Basis parallel dazu mitzuentwickeln.

Die Implementierung von TLS 1.3 in OMNeT++ sollte vorangetrieben werden, um mit echten Implementierungen Testversuche vornehmen zu können, die näher an der Realität einer tatsächlichen konkreten Protokollimplementation liegen.

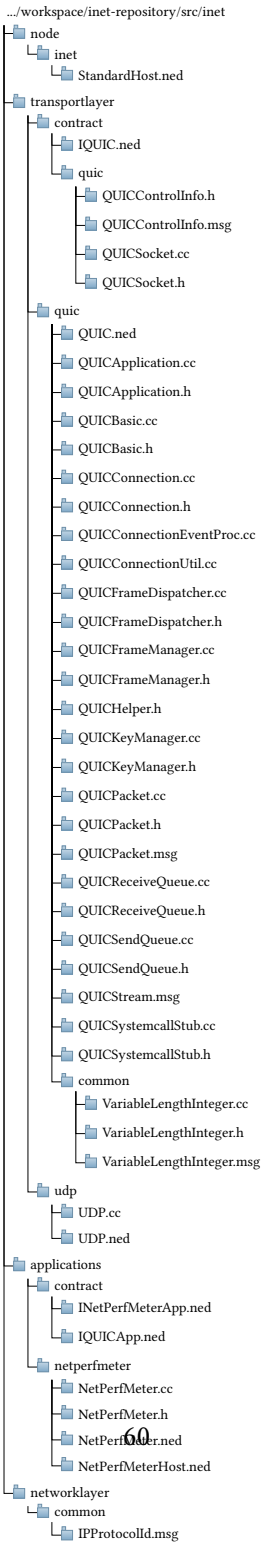
Ein weiterer, naheliegender Schritt ist die Implementierung der noch ausstehenden, im Protokoll spezifizierten Funktionalitäten, um damit eine verlässliche Simulationsumgebung zu schaffen.

Ein Testlauf mit einer Realimplementierung des *drafts* sollte vorgenommen und daraufhin geprüft werden, ob sich diese hier entwickelte Variante mit anderen erstellten Implementierungen konform verhält.

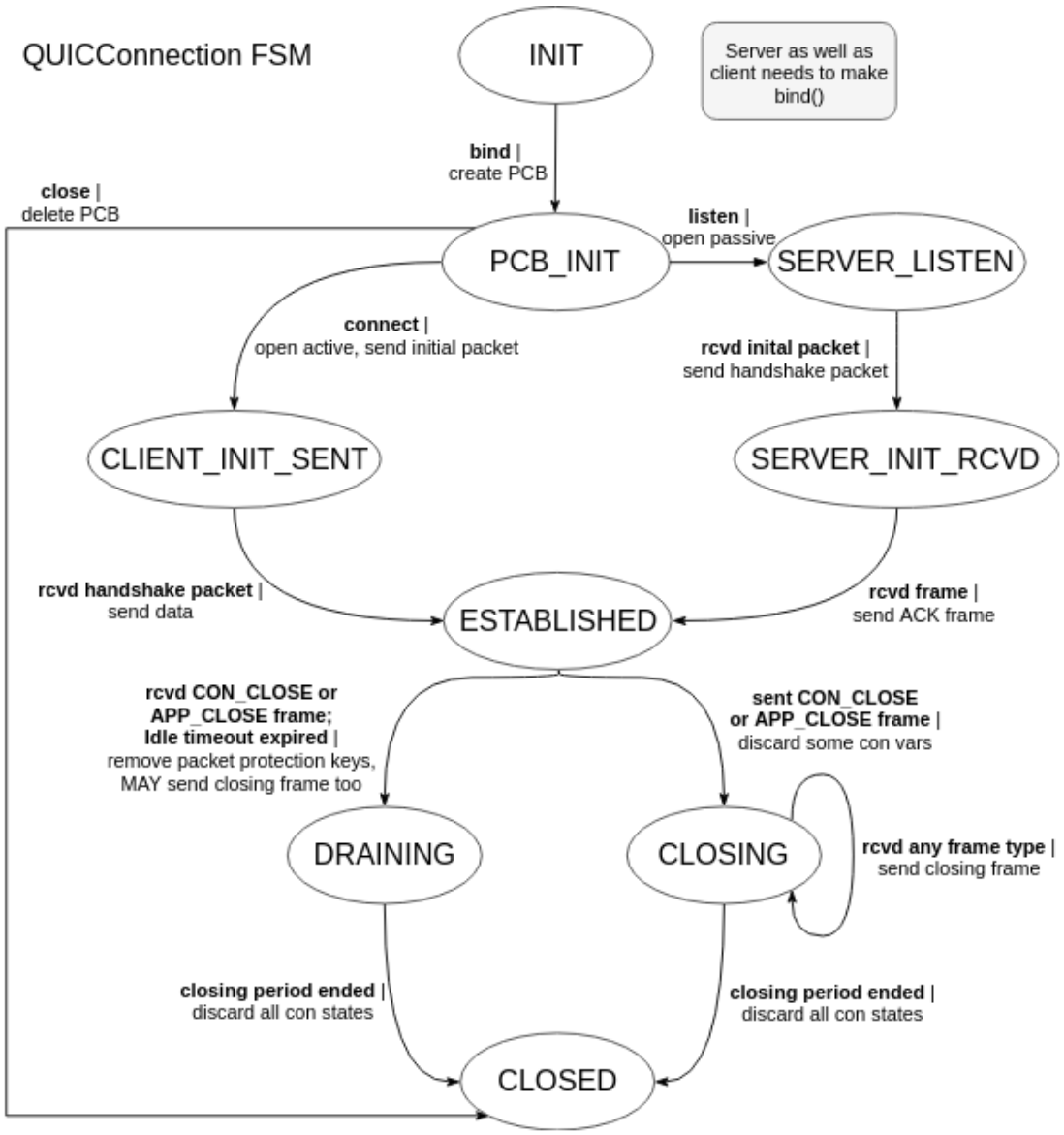
Eine weitere Möglichkeit einer fortgeführten Beschäftigung mit der Thematik ist, es das evaluierte Verhalten in der Simulation mit einem realen System durchzuführen und jene Ergebnisse mit denen dieser Arbeit zu vergleichen.

A. Anhang

A.1. Verzeichnise für die Umsetzung



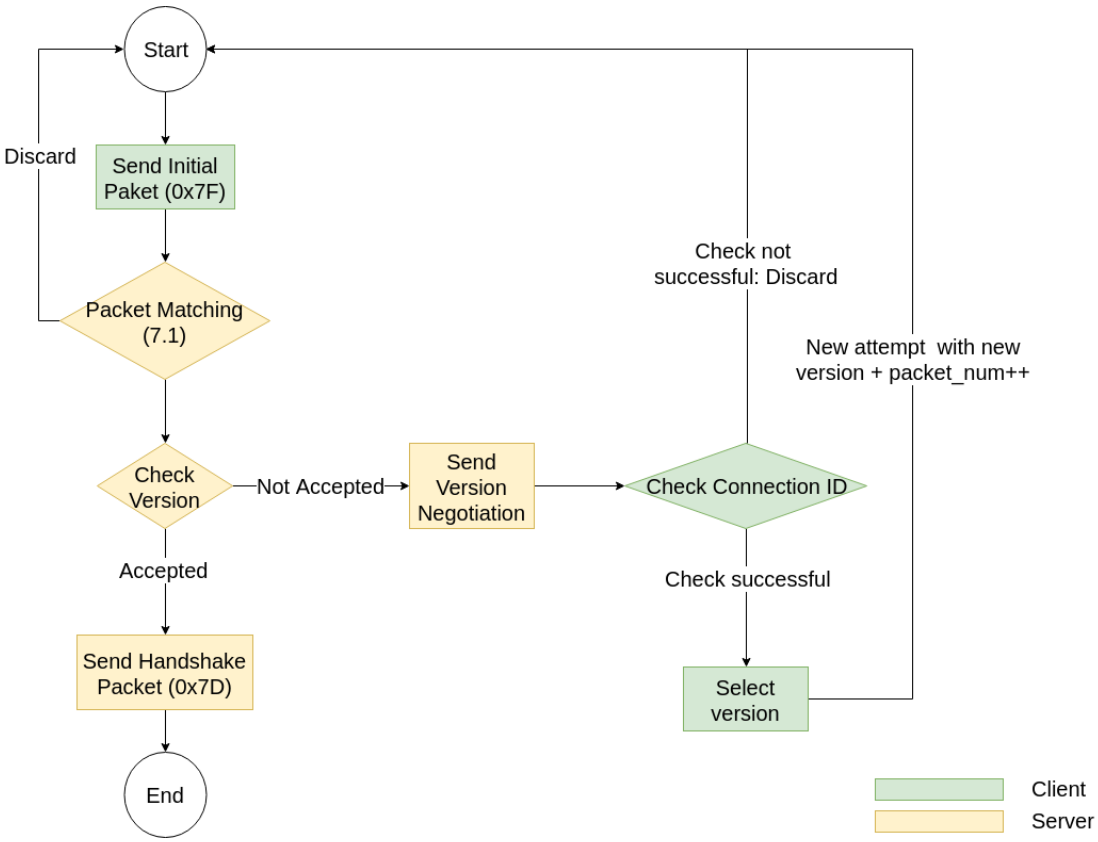
A.2. Design der Finite State Maschine



Quelle: Denis Lugowski

Abbildung A.1.: Design für die State Machine

A.3. Design der Handshake Funktionalitäten



Quelle: Denis Lugowski

Abbildung A.2.: Design der Handshake Funktionalitäten

A.4. QuicTestSingleRouter.ned

```
1  //
2  // This program is free software: you can redistribute it and/or modify
3  // it under the terms of the GNU Lesser General Public License as published by
4  // the Free Software Foundation, either version 3 of the License, or
5  // (at your option) any later version.
6  //
7  // This program is distributed in the hope that it will be useful,
8  // but WITHOUT ANY WARRANTY; without even the implied warranty of
9  // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
10 // GNU Lesser General Public License for more details.
11 //
12 // You should have received a copy of the GNU Lesser General Public License
13 // along with this program.  If not, see http://www.gnu.org/licenses/.
14 //
15
16 package inet.examples.quic.basic;
17
18 import inet.applications.netperfmeter.NetPerfMeterHost;
19 import inet.networklayer.configurator.ipv4.FlatNetworkConfigurator;
20 import inet.networklayer.configurator.ipv4.IPv4NetworkConfigurator;
21 import inet.node.inet.Router;
22 import inet.node.inet.StandardHost;
23
24 import ned.DatarateChannel;
25
26
27 @license(LGPL);
28
29 network QuickTestSingleRouter
30 {
31     @display("bgb=797,414");
32     types:
33         channel pppline extends DatarateChannel
34         {
35             delay = 5.00ms;
36             datarate = 12Mbps;
37             @display("ls=blue,4");
38         }
39     submodules:
40         router1: Router {
41             @display("p=360.44998,139.17375");
42             gates:
43                 pppg[2];
44         }
45         configurator: IPv4NetworkConfigurator {
46             @display("p=37,31");
47         }
48         host1: NetPerfMeterHost {
49             @display("p=37,199");
50             gates:
51                 pppg[1];
52         }
53         host2: NetPerfMeterHost {
54             @display("p=661.82623,143.17874");
55             gates:
56                 pppg[1];
57         }
58
59     connections:
60         host1.pppg[0] <--> pppline <--> router1.pppg[0];
61         router1.pppg[1] <--> pppline <--> host2.pppg[0];
62 }
```

Listing A.1: QuicTestSingleRouter.ned

A.5. omnetpp.ini

```
[General]
network = QuickTestSingleRouter
**.*.version = ${version = 0.3}

## Setup connection line
simtime-resolution = ms
sim-time-limit = ${simTimeEnd = 400}s
**.*.ppp[*].ppp.mtu = 1500B

## Setup Queues at router
**.*.ppp[*].queueType = "DropTailQueue" #REDQueue
**.*router*.**.frameCapacity = 84

## Debug setup
**.scalar-recording = true
**.vector-recording = true
testNetwork.**.vector-recording-intervals = 0..360

## configuration setup
*.configurator.dumpAddresses = true
*.configurator.dumpTopology = true
*.configurator.dumpRoutes = true
*.configurator.dumpLinks = true

## Experiment setup
**.*host*.numNetPerfMeterApps = 1
**.*host*.netPerfMeterApp[*].protocol = "QUIC"
# framerate              5Mb/s            620Hz   || 592Hz
# framerate              10Mb/s           1240Hz  || 1184Hz
**.*host*.netPerfMeterApp[*].frameRate = 1184Hz

**.*host*.netPerfMeterApp[*].startTime = 1.0s
**.*host*.netPerfMeterApp[*].stopTime = 300s
**.host*.netPerfMeterApp[*].resetTime = 60s

**.host1.netPerfMeterApp[*].activeMode = true
**.host2.netPerfMeterApp[*].activeMode = false
**.host2.netPerfMeterApp[*].connectTime = 0.01s

**.host1.netPerfMeterApp[*].localPort = 8000
**.host1.netPerfMeterApp[*].remoteAddress = "host2"
**.host1.netPerfMeterApp[*].remotePort = 8001

**.host2.netPerfMeterApp[*].localPort = 8001
**.host2.netPerfMeterApp[*].remotePort = 8000

**.*.netPerfMeterApp[*].outboundStreams = 1
**.*.netPerfMeterApp[*].maxMsgSize = 1250B
**.*.netPerfMeterApp[*].frameSize = 1250B
**.*.netPerfMeterApp[*].queueSize = 5e+05B
 **.cmdenv-log-level = trace
[Config QUIC]
**.*host*.netPerfMeterApp[*].protocol = "QUIC"
[Config TCP]
**.*host*.netPerfMeterApp[*].protocol = "TCP"
[Config SCTP]
**.*host*.netPerfMeterApp[*].protocol = "SCTP"
```

Listing A.2: omnetpp.ini

Literaturverzeichnis

[1] *Why Performance Matters | Web Fundamentals | Google Developers.* https://developers.google.com/web/fundamentals/performance/why-performance-matters/

[2] IEEE Standard for Software and System Test Documentation. In: *IEEE Std 829-2008* (2008), July, S. 1–150. http://dx.doi.org/10.1109/IEEESTD.2008.4578383. – DOI 10.1109/IEEESTD.2008.4578383

[3] Akamai Technologies, Inc. ; Akamai Technologies, Inc. (Hrsg.): *Global Average Connection Speed Increases 15 Percent Year over Year, According to Akamais First Quarter, 2017 State of the Internet Report.* https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-first-quarter-2017-state-of-the-internet-connectivjsp. Version: 2017

[4] Alvestrand, H.: A Mission Statement for the IETF / RFC Editor. RFC Editor, October 2004 (95). – BCP. – ISSN 2070–1721

[5] Belshe, M. ; Peon, R. ; Thomson, M.: Hypertext Transfer Protocol Version 2 (HTTP/2) / RFC Editor. Version: May 2015. http://www.rfc-editor.org/rfc/rfc7540.txt. RFC Editor, May 2015 (7540). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc7540.txt

[6] Belshe, Mike ; Peon, Roberto: SPDY protocol—Draft 3.1. In: *URL http://www. chromium. org/spdy/spdyprotocol/spdy-protocol-draft3-1* (2012)

[7] Bishop, Mike: Hypertext Transfer Protocol Version 3 (HTTP/3) / IETF Secretariat. Version: March 2019. http://www.ietf.org/internet-drafts/draft-ietf-quic-http-19.txt. 2019 (draft-ietf-quic-http-19). – Internet-Draft. – http://www.ietf.org/internet-drafts/draft-ietf-quic-http-19.txt

[8] Black, D. ; Peterson, D.: Deprecation of the Internet Fibre Channel Protocol (iFCP) Address Translation Mode / RFC Editor. RFC Editor, March 2011 (6172). – RFC. – ISSN 2070–1721

[9] Braden, Robert: Requirements for Internet Hosts - Communication Layers / RFC Editor. Version: October 1989. http://www.rfc-editor.org/rfc/rfc1122.txt. RFC Editor, October 1989 (3). – STD. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc1122.txt

[10] Braun, Torsten ; Diot, Christophe ; Hoglander, Anna ; Roca, Vincent: An Experimental User Level Implementation of TCP / INRIA. Version: September 1995. https://hal.inria.fr/inria-00074040. 1995 (RR-2650). – Forschungsbericht

[11] Breslau, L. ; Estrin, D. ; Fall, K. ; Floyd, S. ; Heidemann, J. ; Helmy, A. ; Huang, P. ; McCanne, S. ; Varadhan, K. ; Xu, Ya ; Yu, Haobo: Advances in network simulation. In: *Computer* 33 (2000), May, Nr. 5, S. 59–67. http://dx.doi.org/10.1109/2.841785. – DOI 10.1109/2.841785. – ISSN 0018–9162

[12] 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 : ACM, 2015 (SAC '15). – ISBN 978–1–4503–3196–8, 609–614

[13] Carpenter, B. ; Brim, S.: Middleboxes: Taxonomy and Issues / RFC Editor. Version: February 2002. http://www.rfc-editor.org/rfc/rfc3234.txt. RFC Editor, February 2002 (3234). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc3234.txt

[14] Cook, Sarah ; Mathieu, Bertrand ; Truong, Patrick ; Hamchaoui, Isabelle: QUIC: Better for what and for whom? In: *IEEE International Conference on Communications (ICC2017)*, 2017

[15] Corporation, Mozilla: *mozquic is an ietf quic library in c++*. https://github.com/mcmanus/mozquic. Version: 2019-04-15

[16] Cui, Y. ; Li, T. ; Liu, C. ; Wang, X. ; Kühlewind, M.: Innovating Transport with QUIC: Design Approaches and Research Challenges. In: *IEEE Internet Computing* 21 (2017), Mar, Nr. 2, S. 72–76. http://dx.doi.org/10.1109/MIC.2017.44. – DOI 10.1109/MIC.2017.44. – ISSN 1089–7801

[17] Datatracker, IETF: *Active IETF working groups*. https://datatracker.ietf.org/wg/. Version: 2018-12-15

[18] Desjardins, Jeff ; Desjardins, Jeff (Hrsg.): *Here's how Amazon makes its money*. https://www.businessinsider.de/how-amazon-makes-money-2017-12?r=US&IR=T. Version: 2017

[19] Dierks, T. ; Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2 / RFC Editor. Version: August 2008. http://www.rfc-editor.org/rfc/rfc5246.txt. RFC Editor, August 2008 (5246). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc5246.txt

[20] Duke, M. ; Braden, R. ; Eddy, W. ; Blanton, E. ; Zimmermann, A.: A Roadmap for Transmission Control Protocol (TCP) Specification Documents / RFC Editor. RFC Editor, February 2015 (7414). – RFC. – ISSN 2070–1721

[21] eMarketer ; eMarketer (Hrsg.): *Retail e-commerce sales worldwide from 2014 to 2021 (in billion U.S. dollars)*. https://www.statista.com/statistics/379046/worldwide-retail-e-commerce-sales/. Version: 2018

[22] e.V., German Testing B.: *Lehrpläne ISTQB Certified Tester Schema Glossar*. https://www.german-testing-board.info/wp-content/uploads/2016/06/ISTQB%C2%AEGTB-Standardglossar-der-Testbegriffe-Deutsch-Englisch.pdf. Version: 2018-12-15

[23] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 / RFC Editor. RFC Editor, January 1997 (2068). – RFC. – ISSN 2070–1721

[24] Flach, Tobias ; Dukkipati, Nandita ; Terzis, Andreas ; Raghavan, Barath ; Cardwell, Neal ; Cheng, Yuchung ; Jain, Ankur ; Hao, Shuai ; Katz-Bassett, Ethan ; Govindan, Ramesh: Reducing Web Latency: The Virtue of Gentle Aggression. In: *SIGCOMM Comput. Commun. Rev.* 43 (2013), August, Nr. 4, 159–170. http://dx.doi.org/10.1145/2534169.2486014. – DOI 10.1145/2534169.2486014. – ISSN 0146–4833

[25] Flach, Tobias ; Dukkipati, Nandita ; Terzis, Andreas ; Raghavan, Barath ; Cardwell, Neal ; Cheng, Yuchung ; Jain, Ankur ; Hao, Shuai ; Katz-Bassett, Ethan ; Govindan, Ramesh: Reducing Web Latency: The Virtue of Gentle Aggression. In: *Proceedings of*

the ACM SIGCOMM 2013 Conference on SIGCOMM. New York, NY, USA : ACM, 2013 (SIGCOMM '13). – ISBN 978–1–4503–2056–6, 159–170

[26] Gao, Richard: *The Pixel and Pixel XL are guaranteed Android version updates for at least 2 years*. https://www.androidpolice.com/2016/10/19/pixel-pixel-xl-guaranteed-android-version-updates-least-2-years/. Version: Oct 2016

[27] Grigorik, Ilya: *High Performance Browser Networking*. O'Reilly, 2013 https://hpbn.co

[28] Grinnem, K. . G. ; Andersson, T. ; Brunstrom, A.: Performance Benefits of Avoiding Head-of-Line Blocking in SCTP. In: *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services - (icas-isns'05)*, 2005. – ISSN 2168–1864, S. 44–44

[29] Group., IETF HTTP W. ; Group, IETF HTTP W. (Hrsg.): *Does HTTP/2 require encryption?* https://http2.github.io/faq/#does-http2-require-encryption. Version: 2018

[30] Handley, M.: Why the Internet Only Just Works. In: *BT Technology Journal* 24 (2006), Juli, Nr. 3, 119–129. http://dx.doi.org/10.1007/s10550-006-0084-z. – DOI 10.1007/s10550–006–0084–z. – ISSN 1358–3948

[31] Henderson, Thomas R. ; Lacage, Mathieu ; Riley, George F. ; Dowell, Craig ; Kopena, Joseph: Network simulations with the ns-3 simulator. In: *SIGCOMM demonstration* 14 (2008), Nr. 14, S. 527

[32] Hoffman, P. ; Harris, S.: The Tao of IETF - A Novice's Guide to the Internet Engineering Task Force / RFC Editor. RFC Editor, September 2006 (4677). – RFC. – ISSN 2070–1721

[33] Hopcroft, J.E. ; Motwani, R. ; Ullman, J.D.: *Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit*. Pearson Deutschland, 2011 (Pearson Studium - IT). https://books.google.de/books?id=_1DCcqaY0gsC. – ISBN 9783868940824

[34] Huitema, Christian: *Minimal implementation of the QUIC protocol*. https://github.com/private-octopus/picoquic. Version: 2019-04-15

[35] IANA: *Protocol Numbers*. https://www.iana.org/assignments/protocol-numbers. Version: 2017-10-13

[36] Inc., Simulcraft: *OMNeT++ Community Summit 2018*. https://summit.omnetpp.org/archive/2018/. Version: 2018-12-15

[37] inet@omnetpp.org ; inet@omnetpp.org (Hrsg.): *What Is INET Framework?* https://inet.omnetpp.org/Introduction.html. Version: 2018

[38] inet@omnetpp.org ; inet@omnetpp.org (Hrsg.): *What Is INET Framework?* https://inet.omnetpp.org/Introduction.html. Version: 2018

[39] Iyengar, Jana ; Thomson, Martin: QUIC: A UDP-Based Multiplexed and Secure Transport / IETF Secretariat. Version: March 2018. https://www.ietf.org/archive/id/draft-ietf-quic-transport-10.txt. 2018 (draft-ietf-quic-transport-10). – Internet-Draft. – https://www.ietf.org/archive/id/draft-ietf-quic-transport-10.txt

[40] Iyengar, Jana ; Thomson, Martin: QUIC: A UDP-Based Multiplexed and Secure Transport / IETF Secretariat. Version: May 2018. http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-12.txt. 2018 (draft-ietf-quic-transport-12). – Internet-Draft. – http://www.ietf.org/internet-drafts/draft-ietf-quic-transport-12.txt

[41] Jana ; Swett, Ian: QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2 / IETF Secretariat. Version: June 2015. http://www.ietf.org/internet-drafts/draft-tsvwg-quic-protocol-00.txt. 2015 (draft-tsvwg-quic-protocol-00). – Internet-Draft. – http://www.ietf.org/internet-drafts/draft-tsvwg-quic-protocol-00.txt

[42] Karn, Phil ; Partridge, Craig: Improving round-trip time estimates in reliable transport protocols. In: *ACM SIGCOMM Computer Communication Review* 25 (1995), Nr. 1, S. 66–74

[43] Kohler, E. ; Handley, M. ; Floyd, S.: Datagram Congestion Control Protocol (DCCP) / RFC Editor. Version: March 2006. http://www.rfc-editor.org/rfc/rfc4340.txt. RFC Editor, March 2006 (4340). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc4340.txt

[44] Langley, Adam ; Riddoch, Alistair ; Wilk, Alyssa ; Vicente, Antonio ; Krasic, Charles ; Zhang, Dan ; Yang, Fan ; Kouranov, Fedor ; Swett, Ian ; Iyengar, Janardhan ; Bailey, Jeff ; Dorfman, Jeremy ; Roskind, Jim ; Kulik, Joanna ; Westin, Patrik ; Tenneti, Raman ; Shade, Robbie ; Hamilton, Ryan ; Vasiliev, Victor ; Chang, Wan-Teh ; Shi, Zhongyi:

The QUIC Transport Protocol: Design and Internet-Scale Deployment. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA : ACM, 2017 (SIGCOMM '17). – ISBN 978–1–4503–4653–5, 183–196

[45] LLC, Google: *Experimenting with QUIC*. https://blog.chromium.org/2013/06/experimenting-with-quic.html. Version: 2013

[46] Mahmoudi, Mehran ; Nadi, Sarah: An Empirical Study of Android Changes in CyanogenMod. In: *CoRR* abs/1801.02716 (2018). http://arxiv.org/abs/1801.02716

[47] Malsch, Michael: *OMNeT++ basierte Simulation der Kommunikationsinfrastruktur im Flugzeug*. 2018

[48] McGregor, Ian: Equipment Interface: The Relationship Between Simulation and Emulation. In: *Proceedings of the 34th Conference on Winter Simulation: Exploring New Frontiers*, Winter Simulation Conference, 2002 (WSC '02). – ISBN 0–7803–7615–3, 1683–1688

[49] Megyesi, P. ; Krämer, Z. ; Molnár, S.: How quick is QUIC? In: *2016 IEEE International Conference on Communications (ICC)*, 2016. – ISSN 1938–1883, S. 1–6

[50] Meister, B. W. ; Janson, P. A. ; Svobodova, L.: Connection-oriented versus connectionless protocols: A performance study. In: *IEEE Transactions on Computers* C-34 (1985), Dec, Nr. 12, S. 1164–1173. http://dx.doi.org/10.1109/TC.1985.6312214. – DOI 10.1109/TC.1985.6312214. – ISSN 0018–9340

[51] Mogensen, Rasmus S.: Reliability enhancement for LTE using MPQUIC in a mixed traffic scenario.

[52] Narten, T.: Assigning Experimental and Testing Numbers Considered Useful / RFC Editor. RFC Editor, January 2004 (82). – BCP. – ISSN 2070–1721

[53] Naylor, David ; Finamore, Alessandro ; Leontiadis, Ilias ; Grunenberger, Yan ; Mellia, Marco ; Munafò, Maurizio ; Papagiannaki, Konstantina ; Steenkiste, Peter: The Cost of the SSïn HTTPS. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. New York, NY, USA : ACM, 2014 (CoNEXT '14). – ISBN 978–1–4503–3279–8, 133–140

[54] P. Miguel, José ; Mauricio, David ; Rodríguez, Glen: A Review of Software Quality Models for the Evaluation of Software Products. In: *International Journal of Software Engineering & Applications* 5 (2014), Nov, Nr. 6, 31–53. http://dx.doi.org/10.5121/ijsea.2014.5603. – DOI 10.5121/ijsea.2014.5603. – ISSN 0975–9018

[55] Pittevils, Kevin: *Quicker: On the design and implementation of the QUIC protocol*, tUL, Diplomarbeit, 2018

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

[57] Postel, Jon: Transmission Control Protocol / RFC Editor. Version: September 1981. http://www.rfc-editor.org/rfc/rfc793.txt. RFC Editor, September 1981 (7). – STD. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc793.txt

[58] quic@ietf.org ; quic@ietf.org (Hrsg.): *QUIC (quic)(Milestones)*. https://datatracker.ietf.org/wg/quic/charter/. Version: 2018

[59] Ramadan, Nagy ; Abdelwahab, Ihab: Impact of Implementing HTTP//2 in Web Services. In: *International Journal of Computer Applications* 147 (2016), 08, S. 27–32. http://dx.doi.org/10.5120/ijca2016911182. – DOI 10.5120/ijca2016911182

[60] Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3 / RFC Editor. RFC Editor, August 2018 (8446). – RFC. – ISSN 2070–1721

[61] Robertson, Suzanne ; Robertson, James: *Mastering the Requirements Process (2Nd Edition)*. Addison-Wesley Professional, 2006. – ISBN 0321419499

[62] Rüngeler, Irene ; Tüxen, Michael ; Rathgeb, Erwin P.: Integration of SCTP in the OMNeT++ Simulation Environment. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008 (Simutools '08). – ISBN 978–963–9799–20–2, 78:1–78:8

[63] Saxcé, H. de ; Oprescu, I. ; Chen, Y.: Is HTTP/2 really faster than HTTP/1.1? In: *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2015, S. 293–299

[64] Scharf, Michael ; Kiesel, Sebastian: Head-of-line Blocking in TCP and SCTP: Analysis and Measurements. In: *GLOBECOM* Bd. 6, 2006, S. 1–5

[65] Schwaber, Ken ; Beedle, Mike: *Agile software development with Scrum*. Bd. 1. Prentice Hall Upper Saddle River, 2002

[66] Souci, Benjamin S. ; Lemaire, Maude: *An inside look at the architecture of nodejs.* 2014

[67] Srisuresh, P. ; Egevang, K.: Traditional IP Network Address Translator (Traditional NAT) / RFC Editor. RFC Editor, January 2001 (3022). – RFC. – ISSN 2070–1721

[68] Steinbach, Till ; Kenfack, Hermand D. ; Korf, Franz ; Schmidt, Thomas C.: An Extension of the OMNeT++ INET Framework for Simulating Real-time Ethernet with High Accuracy. In: *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques.* ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2011 (SIMUTools '11). – ISBN 978–1–936968–00–8, 375–382

[69] Stevens, W. R. ; Wright, Gary R.: *TCP/IP Illustrated (Vol. 2): The Implementation.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63354–X

[70] Stewart, R.: Stream Control Transmission Protocol / RFC Editor. Version: September 2007. http://www.rfc-editor.org/rfc/rfc4960.txt. RFC Editor, September 2007 (4960). – RFC. – ISSN 2070–1721. – http://www.rfc-editor.org/rfc/rfc4960.txt

[71] Szancer, Sebastian: *Ein OMNeT++ Simulationsmodell für SERCOS III mit TSN Interface.* 2017

[72] Tanenbaum, Andrew: *Computer Networks.* 4th. Prentice Hall Professional Technical Reference, 2002. – ISBN 0130661023

[73] Trammell, B. ; Hildebrand, J.: Evolving Transport in the Internet. In: *IEEE Internet Computing* 18 (2014), Sept, Nr. 5, S. 60–64. http://dx.doi.org/10.1109/MIC.2014.91. – DOI 10.1109/MIC.2014.91. – ISSN 1089–7801

[74] Varga, Andras u. a.: Omnet++ user manual. In: *OMNeT++ Discrete Event Simulation System. Available at: http://www. omnetpp. org/doc/manual/usman. html* (2010)

[75] Varga, András ; Hornig, Rudolf: An Overview of the OMNeT++ Simulation Environment. In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops.* ICST, Brussels, Belgium, Belgium : ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008 (Simutools '08). – ISBN 978–963–9799–20–2, 60:1–60:10

[76] Walrand, J. ; Varaiya, P.: *High-Performance Communication Networks*. Elsevier Science, 1999 (The Morgan Kaufmann Series in Networking). https://books.google.de/books?id=YIvYZpCx8PMC. – ISBN 9780080508030

[77] Yemm, G.: *FT Essential Guide to Leading Your Team: How to Set Goals, Measure Performance and Reward Talent*. Pearson Education Limited, 2012 (The FT Guides). https://books.google.de/books?id=SQuxNERDFeIC. – ISBN 9780273772446

[78] Yoo, B. K. ; Donthu, Naveen: Developing a Scale to Measure the Perceived Quality of An Internet Shopping Site (SITEQUAL), 2005

[79] Zimmermann, H.: OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. In: *IEEE Transactions on Communications* 28 (1980), April, Nr. 4, S. 425–432. http://dx.doi.org/10.1109/TCOM.1980.1094702. – DOI 10.1109/TCOM.1980.1094702. – ISSN 0090–6778

Abkürzungsverzeichnis

ACK Acknowledgement. 34, 48

CADS Communication and distributed Systems. 29

DCCP Datagram Congestion Control Protocol. 6

FC Fibre Channel. 36

FSM Finite State Machine. 33, 34, 42–44

HAW Hochschule für Angewande Wissenschaften. 29

HoL Head of Line. 5, 6, 22

HTTP/1 Hypertext Transfer Protocol. 5, 11

HTTP/2 Hypertext Transfer Protocol 2. 5–7

IETF Internet Engineering Task Force. 7, 8, 10, 12, 13, 15

IP Internet Protocol. 20, 36–38

NAT Network Address Translation. 20

NED Network Description. 35, 38, 53

OSI-MODELL Open Systems Interconnection Model. 3, 13, 14, 31, 35

PCB Protocol Control Block. 34

QUIC Quick UDP Internet Connection. 7–17, 19–23, 25, 29, 31, 33–40, 47, 49–51, 54–57

RTT Paketumlaufzeit bzw. Round Trip Time. 11, 12, 16, 17

SCTP Stream Control Transmission Protocol. 6, 14, 35, 50, 52, 54–56

TCP Transmission Control Protocol. 6, 7, 9, 14, 20, 35, 50, 51, 54–56

TLS Transport Layer Security Protocol. 7, 15, 47, 58

UDP User Datagram Protocol. 6, 7, 9, 14, 15, 36–39, 43, 44

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 08. Mai 2019 Marvin Butkereit