HAW HAMBURG

MASTERTHESIS
Mikko Eberhardt

Gesteuertes Testverfahren in einer Simulationsumgebung am Beispiel des QUIC Handshakes

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Mikko Eberhardt

Gesteuertes Testverfahren in einer Simulationsumgebung am Beispiel des QUIC Handshakes

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

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Bettina Buth

Eingereicht am: 15. Juni 2020

Mikko Eberhardt

Thema der Arbeit

Gesteuertes Testverfahren in einer Simulationsumgebung am Beispiel des QUIC Handshakes

Stichworte

Test, Continuos, QUIC, Omnet++, INET, Jenkins, Docker, Continuos Testing, Testverfahren, Automatisierung, Simulationsumgebung

Kurzzusammenfassung

Für die Übertragung von Daten im Internet sind mehrere Technologien notwendig, für die Kooridnation und sicherstellung einer erfolgreichen Übertragung sind sogenannte Transportprotokolle im Einsatz. Damit eine einwandfreie Übertragung ermöglicht wird müssen diese Technologien getestet werden. Simulationsumgebungen werden eingesetzt um diese Tests reproduzierbar zu gestallten. Fraglich hierbei ist wie der Einsatz von Ressourcen und einer Testmethodik in der Entwicklung effizienter eingesetzt werden kann. Dafür wurde ein gesteuertes Testverfahren erstellt, welches kontinuierlich die Änderungen in einem Repository überprüft. Mit diesem gesteuerten Testverfahren ist es Möglich den Einsatz der Ressourcen effizienter zu gestalten, indem unter anderem die Arbeitszeit von Entwicklern reduziert werden kann. Desweiteren wird durch die Überprüfung jeder Änderung in dem Repository das Testverfahren effizient Eingesetzt. So können mit diesem gesteuerten Testverfahren die Ressourcen effizienter eingesetzt und gleichzeitig die Qualität der Software durch das kontinuierliche Testen gesteigert werden.

Mikko Eberhardt

Title of Thesis

Controlled test procedure in a simulation environment using the example of the QUIC handshake

Keywords

Test, Continuos, QUIC, Omnet++, INET, Jenkins, Docker, Continuos Testing, Test Procedure, Automation, Simulation Environment

Abstract

Several technologies are necessary for the transmission of data on the Internet, so-called transport protocols are used to coordinate and ensure successful transmission. These technologies must be tested in order to enable a flawless transmission. Simulation environments are used to make these tests reproducible. It is questionable how the use of resources and a test methodology can be used more efficiently in development. A controlled test procedure was created, which continuously checks the changes in a repository. This controlled test procedure makes it possible to make more efficient use of resources by, among other things, reducing the working hours of developers. Furthermore, the test procedure is used efficiently by checking each change in the repository. With this controlled test procedure, resources can be used more efficiently and, at the same time, the quality of the software can be increased through continuous testing.

Inhaltsverzeichnis

Abbildungsverzeichnis **vii**

Tabellenverzeichnis **ix**

Listings **x**

1 Einleitung **1**

2 Grundlagen **3**
2.1 Begrifflichkeiten und Besonderheiten 3
2.2 Testen und Softwaretests 10
2.3 Transportprotokolle 14
2.4 QUIC Transportprotokoll 15
2.5 Simulationsumgebung für Transportprotokolle 17

3 Analyse **19**
3.1 Problemstellung 19
3.2 Verwandte Arbeiten 21
3.3 Anforderungen und Machbarkeit 23
3.4 Umfrage an Entwickler 25
3.5 Simulationen und Simulationsumgebung 27
3.6 Testen in der Simulationsumgebung Omnet++/INET 31
3.7 QUIC im INET Framework 35
3.8 QUIC Handshake 38
3.9 Gesteuerte Testverfahren im Forschungskontext 41
3.10 Strukturierung & Aufbau 42

4 Konzept und Realisierung **43**
4.1 Gesteuertes Testverfahren 43
4.2 Machbarkeitsuntersuchung mit dem QUIC Handshake 46

4.3 Entscheidungen & Annahmen 48
4.4 Die Umsetzung der Testumgebung 50
4.5 Der Aufbau des Testverfahrens 56
4.6 Die Jenkins Pipeline 62
4.7 Der Arbeitsablauf des Testverfahrens 66

5 Evaluation **69**
5.1 Vorgehensweise 69
5.2 Datengenerierung und Beobachtungen 70
5.3 Zusatzdaten 76
5.4 Einordnung 80

6 Ausblick & Schluss **87**
6.1 Ergebniszusammenfassung 87
6.2 Erweiterungen 88

Literaturverzeichnis **92**

Selbstständigkeitserklärung **101**

Abbildungsverzeichnis

2.1 Ein beispielhafter Arbeitsablauf mit einem Versionskontrollsystem wie Git. 6
2.2 Gegenüberstellung von Virtualisierung und Containerisierung und deren Nutzung von Hardware.[41] 8
2.3 V-Modell Konzept in der Softwareentwicklung.[5] 12
2.4 Ablauf von Tests und der Beschreibung möglicher Fehlerquellen. 13
2.5 Gegenüberstellung des OSI und TCP/IP Referenzmodells.[9] 15
2.6 Gegenüberstellung des TCP Stacks und dem auf UDP aufgesetztem QUIC 16

3.1 Die Aufstellung von Simple/Compound Modulen aus dem Omnet++ Benutzerhandbuch[62] 28
3.2 Der Prozess eine Simulation für Omnet++ zu erzeugen, wie sie im Benutzerhandbuch von Omnet++ beschrieben ist[62] 30
3.3 Das Klassendiagramm wie es für die Implementierung von QUIC in das INET Framework erstellt wurde.[88] 36
3.4 Der Zustandsautomaten für eine QUIC Verbindung von einem Klienten (initiator) und einem Server angelehnt an der Basis von Denis Lugowski[88] 37
3.5 Der Ablauf eines QUIC Handshakes ohne die Version Negotiation und TLS wie QUIC aktuell im INET Framework Implementiert ist. Aus den Arbeiten von Lugowski[88] 39
3.6 Die Gegenüberstellung von einem 1-RTT und einem 0-RTT Handshake und die Möglichkeiten wann Daten gesendet werden können. 40

4.1 Die benötigten Komponenten für das Testsystem und wie sie in Abhängigkeit zur Ablauf stehen. 46
4.2 Diese Abbildung beschreibt die Architektur wie sie auf dem Server eingerichtet wird für das Testverfahren 46
4.3 Die Docker Engine als Docker im Docker und mit geteilter *.lock* Datei. . . 63
4.4 Der Arbeitsablauf für das gesteuerte Testverfahren mit dem auftretenden Informationsaustausch zwischen den Technologien. 67

5.1 Kurze Übersicht der von Jenkins bereitgestellten Historie über die letzten Pipeline Durchläufe. 72
5.2 Detailierte Aufstellung der letzten Pipeline Durchläufe mit einer Zeitangabe und der Markierung eines Schrittes in der die Pipeline fehlschlägt. . 73
5.3 Beispielhafte Ausgabe der Logdateien von Jenkins zur Unterscheidung von Ausgabe der Pipeline und der ausgeführten Kommandos 74
5.4 Eine Übersicht über die Antworten aus der Fragerunde zur Analyse 76
5.5 Die Auswertung der Antworten aus Kategorie A 78
5.6 Die Auswertung der Antworten aus Kategorie B 80
5.7 Ein beispielhafter Arbeitsprozess eines Entwicklers bei dem die Suche nach dem Fehlschlagen von bestehenden Test erschwert wird. 83
5.8 Ein beispielhafter Arbeitsprozess bei dem ein Entwickler direkt auf das Fehlschlagen von Tests durch ein Fehlschlagen der Pipeline hingewiesen wird. 84
5.9 Ein beispielhafter Ablauf einer Entwicklung mit und ohne das Feedback eines Testverfahren. 86

Tabellenverzeichnis

3.1 Beschreibung der Ressourcen die für einen Testdurchlauf gegeben sein müssen. 34

4.1 Auflistung der im Jenkins Server verwendeten Plugins. 49
4.2 Eine detaillierte Beschreibung der Events die innerhalb des Testverfahrens auftreten. 68

Listings

3.1 Pseudocode für die benötigten Schritte des Kontrollscripts zum erzeugen der Tests, dem Zusammenbau und dem ausführen der Simulation 33
3.2 Die Ausgabe eines Demo Testdurchlaufs mit dem Kontrollscript für das durchführen von Tests 33

4.1 QUIC Network definition für die verwendeten Simulationen 52
4.2 Die Definition der benötigten Ereignisse für einen erfolgreichen Handshake 55
4.3 Das Dockerfile, das den Jenkins Server um die Installation von Docker erweitert. 59
4.4 Die *docker-compose.yml* Datei, welche benötigt wird um einen Stack mit vollständiger Konfiguration zu starten. 60
4.5 Das Pipeline Script welches den Arbeitsschritt zum Ausführen der Tests in Form eines Pseudocodes beinhaltet. 64

1 Einleitung

In unserem Alltag verbringen wir Menschen einen Großteil unserer Zeit im Internet und benutzen infolgedessen eine massenhafte Anzahl von Anwendungen[71]. Im Internet kommunizieren diese Anwendungen über eine Vielzahl von Software-Technologien[10]. Einen Teil dieser Technologien bilden die sogenannten Transportprotokolle[47][10], welche auf dem Protokollstack[32][10] arbeiten. Dieser Protokollstack des Internets wird durch Referenzmodelle wie das TCP/IP Referenzmodell [52][76][46] oder durch das OSI Referenzmodell[44] beschrieben. Das TCP/IP Referenzmodell besteht aus vier Schichten, welche äquivalent zu den oberen sechs Schichten des OSI Referenzmodells sind. Auf der Transportebene der zuvor genannten Referenzmodelle arbeiten Transportprotokolle[47] wie UDP[47], TCP[47], SCTP[47] oder QUIC[39] für die Datenübertragung. Doch aus welchem Grund haben diese Protokolle eine solch entscheidende Bedeutung?

Diese Transportprotokolle bilden die Basis für die Datenübertragung im Internet und bekleiden damit eine sehr wichtige Stellung[47]. Ohne eine korrekte und vollständige Datenübertragung ist eine Kommunikation über das Internet nicht möglich. Aus diesem Grund sind diese Transportprotokolle täglich für jedwede Kommunikation im Internet millionenfach im Einsatz. Sie stellen folglich eine Grundlage des Internetverkehrs dar und können somit ausschließlich bei einer korrekten Funktionsweise Datenübertragungen ermöglichen[10].

Bei steigender Nutzung des Internets werden immer mehr Daten durch diese Transportprotokolle gesendet. Um einen reibungslosen Prozess einer Übertragungen sicherzustellen, sind Tests notwendig. Welche Auswirkungen Fehler in der Implementierung hervorrufen können, zeigt der Heartbleed-Bug[31], durch den über TLS-Verbindungen[18] private Daten von Klienten und Servern ausgelesen werden konnten.

Dabei stellt sich die Frage, wie diese Protokolle getestet werden können. Die Realität ist dies hingehend schwierig gestaltet: Der Aufwand ist vergleichsweise hoch, ein flexibles und komplexes Protokoll mit einer möglichst hohen funktionalen Abdeckung zu

testen. Um der Komplexität der Transportprotokolle habhaft zu werden, ist es erforderlich, dass die Tests reproduzierbar sind. Eine Simulationsumgebungen wie Omnet++[60] oder NS2/NS3[59] bieten eine definierte Arbeitsumgebung für die Protokolle.

Es soll untersucht werden wie ein gesteuertes Testverfahren für Simulationen in einer festen Umgebung die Entwicklung unterstützen kann. Dabei soll untersucht werden wie ein effizienter Einsatz von Ressourcen erreicht werden kann. Zusätzlich soll die Testmethodik effizient eingesetzt werden. Die Antwort darauf wird durch ein gesteuertes Testsystem gegeben, welches aufgebaut wird und durch ein Beispiel mit dem QUIC Transportprotokoll[39][1] einen Testfall überprüft. Dadurch beispielhaft ein Entwicklungszyklus dargestellt und die Tests automatisiert ausgeführt.

In dieser Arbeit wird der QUIC Handshake als Demotestfall verwendet um einen Durchgang zu beschreiben. Dafür werden zunächst in den Grundlagen alle notwendigen Technologien und Begrifflichkeiten, die in dieser Arbeit verwendet werden, erläutert. Anschließend folgt ein Abschnitt, der eine Analyse des Problems und die Anforderungen enthält zusammen mit der Untersuchung der verwendeten Tools/Software. Im nächsten Schritt wird das entwickelte Konzept für das gesteuerte Testverfahren und der Ablauf mit dem QUIC Handshake erläutert. Im Anschluss darauf folgt die Evaluation der Vorgehensweise und der geschaffenen Daten, die aus den Testdurchläufen gesammelt werden. Zum Schluss der Arbeit gibt es eine Zusammenfassung, die einer kritischen Betrachtung der durch das gesteuerte Testsystem gesammelten Daten. Ebenfalls werden hier ein Ausblick über mögliche Erweiterungen dieses Testverfahren beschrieben.

2 Grundlagen

Im folgenden Kapitel werden alle Technologien beschrieben, welche innerhalb dieser Arbeit verwendet werden. Um ein erforderliches Grundverständnis zu erhalten, folgt eine Erläuterung aller notwendigen Begrifflichkeiten, eine Einführung in die Thematik des Testens und die Beschreibung der Transportprotokolle. Insbesondere das QUIC Transportprotokoll wird dahingehend fokussiert, da es als das Hauptprotokoll Anwendung findet.

2.1 Begrifflichkeiten und Besonderheiten

In diesem Abschnitt werden zunächst die grundlegenden Begriffe erläutert, welche für das Verständnis dieser Arbeit benötigt werden. Dazu gehören die Begriffe Firewall[10][64], Versionskontrolle[50], Virtualisierung[54], Containerisierung, Continuous Testing, Testen und Softwaretests, Transportprotokolle, das QUIC Transportprotokoll sowie die Simulationsumgebung für Transportprotokolle. Diese Begrifflichkeiten beschreiben Technologien, die innerhalb des Testverfahrens verwendet und somit bei diesem Design berücksichtigt werden. Ein Design in diesem Zusammenhang meint die Abfolge von zusammenarbeitenden Technologien als Beschreibung eines Gesamtsystems.

Firewall

Der Begriff Firewall[64] bedeutet im historischen Kontext “Brandschutzmauer“ und beschreibt den Schutz von zwei getrennten Bereichen. Die Funktionen der heutigen digitalen Namensverwandten sind somit angelehnt an die ursprüngliche Idee einer Brandschutzmauer. Es ist die Aufgabe einer Firewall die Internet abhängigen Systeme zu schützen. Der Schutz entsteht dadurch, dass ein potentieller Angriff durch Trennung von Netzwerken abgewehrt werden kann[10]. Dies können Firmennetzwerke, das Heimnetzwerk oder

ähnliche Subnetze sein. Die Firewall ist ein System, welches in der Lage ist, Datenverkehr im Internet auszuwerten und zu analysieren. Dementsprechend schützt eine Firewall nicht nur Netzwerke sondern auch komplette IT-Systeme innerhalb der Netzwerke vor unbefugtem Zugriff [42]. Dabei gibt es zwei unterschiedliche Komponenten einer Firewall: Zum Einen die Softwarekomponente und zum Anderen die Hardwarekomponente[49]. Die Softwarekomponente besitzt unter anderem die Eigenschaft einzelne Netzwerkpakete zu lesen und anhand von eingerichteten Regeln Datenpakete durchzulassen oder diese zu blockieren[23]. Die klassische Firewall beinhaltet folgende Funktionen um unter anderem Netzwerke zu schützen:

- Paketfilter[87]
- Network Address Translation (NAT)[22]
- URL-Filter[34]
- Content-Filter[69]
- Proxy[19]
- Virtual Private Networks (VPN)[57]
- Stateful Packet Inspection[37]
- Deep Packet Inspection[36]

Die Hardwarekomponente einer Firewall wird meist bei Firmennetzwerken oder in Institutionen verwendet. Diese erlaubt eine detailliertere Analyse der Pakete und des Datenstroms. Nachteilig daran ist, dass dadurch ein vergleichsweise intensiverer Administrationsaufwand einhergeht. Erweiterungen, wie sog. Next Generation Firewalls (NGFW)[72] haben die Möglichkeit zusätzliche Funktionen, wie Intrusion Prevention System (IPS)[90], TLS/SSL und SSH Inspection[66] bereitzustellen. Das bedeutet je mehr Erweiterungen zur Verfügung stehen, desto zuverlässiger ist die Verbindung zwischen den Netzwerken in Bezug auf unbefugten Zugriff[67].

Versionskontrolle

Die Idee der Versionskontrolle[35] ist eine nahtlose Rückverfolgung aller getätigten Änderungen während eines Projekts. Dabei werden alle Ressourcen in einem sogenannten Repository gespeichert. Ein Repository beschreibt einen Speicherort an dem mittels vordefinierter Befehle Änderungen getätigt werden können. Direkte Veränderungen können innerhalb des Repositorys nicht vorgenommen werden. Deshalb haben die jeweiligen Entwickler ihre definierten Befehle um programmierte Änderungen zu bestätigen und in das Repository einzupflegen. Dies hat beispielsweise zur Folge, dass zu jeder Zeit auf den zuletzt gespeicherten Entwicklungsstand zurück gegriffen werden kann. Ein Vorteil ergibt sich dabei bei Problemen, welche zum Beispiel durch eine fehlerhafte Programmcodezeile ausgelöst werden. Es ist daher möglich einen vorausgegangenen Stand aus dem Repository wieder herzustellen, um fehlerhafte Software innerhalb des Repositories zu vermeiden. Nach dem gleichen Prinzip können ebenso explizite Punkte im Entwicklungszyklus hervorgehoben werden. Dies geschieht mit sogenannten Tags, die als Schlagwörter zur prägnanten Beschreibung dienen. Die Tags bilden eine gekennzeichnete Version einer Software, die zum Beispiel mit einer Versionsnummer versehen und anschließend veröffentlicht werden kann.

Das in dieser Arbeit zugrunde gelegte und verwendete Versionskontrollsystem trägt den Namen Git[35]. Git ist ein System, bei dem der einzelne Entwickler eine lokale Kopie des Repositories in seinen Arbeitsbereich abspeichert und die einzelnen Änderungen auf dem Server zusammenführt. Die verwendeten Begriffe innerhalb von Git sind Branches, Commits, Tags, Pullen und Pushen. Ein Branch beschreibt zusammenhängende Schritte innerhalb einer Implementierung. Es existiert initial beim Erstellen des Repositories ein Hauptbranch, namens Master Branch. Von einem Branch können beliebig viele Branches abgezweigt werden. Änderungen an einem Branch werden zunächst durch eine Bestätigung und einer Beschreibung der Änderungen committed und dann mittels Push auf einen definierten Branch zusammengeführt. Um den Entwicklungsstand eines Branches zu erhalten kann dieser in den aktuellen Arbeitsbereich gepullt werden.

Beim Erstellen eines Branches wird der Entwicklungsstand des Abstammungsbranch in den aktuellen Arbeitsbereich kopiert. Alle anderen Branches bleiben von den Änderungen auf diesem Branch unberührt. Die Abbildung 2.1 beschreibt einen Master Branch mit drei unterschiedlichen Release Tags (Branches) und die dazu zugehörigen Versionen. Des Weiteren zeigt die Abbildung einen parallelen Entwicklungsbranch. Weiterhin stellt ein für die Behebung von Fehlern erstellter Branch einen Hotfix dar.

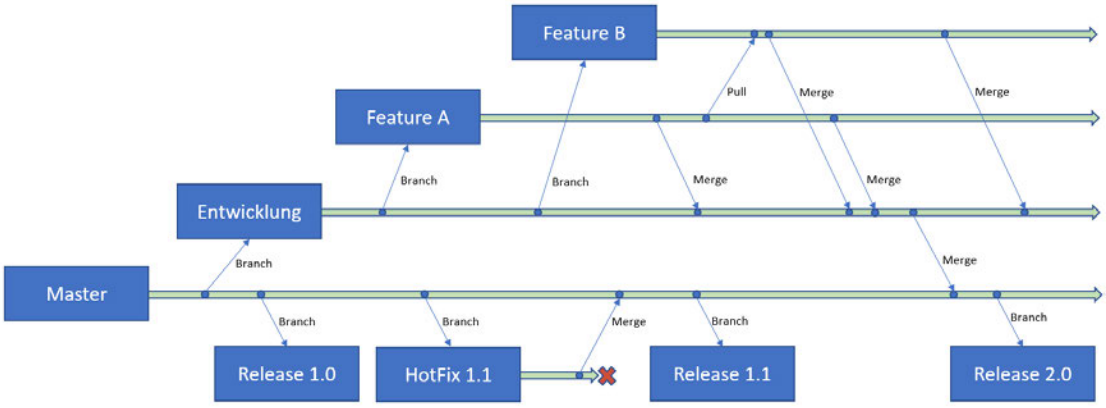

Abbildung 2.1: Ein beispielhafter Arbeitsablauf mit einem Versionskontrollsystem wie Git.

Zusammenfassend ist zu erwähnen, dass das GitHub[27] in dieser Arbeit angewendet wird, da zum einen ein Onlinespeicher zur Verfügung steht. Zum anderen stellt GitHub zusätzliche Technologien bereit, wie das Versenden eines Webhooks[17] bei Änderungen im Repository.

Virtualisierung

Die Virtualisierung[38] bezeichnet die Abstraktion zwischen Hardware und Software. In der Regel stellt die Software in diesem Zusammenhang ein Betriebssystem[79] dar. Ein Betriebssystem, das innerhalb einer virtuellen Umgebung arbeitet wird virtuelle Maschine genannt. Das Virtualisierungssystem ist dafür zuständig, dass die Hardware den unterschiedlichen virtuellen Maschinen zur Verfügung steht. Weiterhin gehören das Management und Überwachen der einzelnen virtuellen Maschinen zu seinen Aufgaben.

Auf jeder virtuellen Maschine ist ein eigenständiges Gast-Betriebssystem zu installieren, auf dem der einzelne Nutzer anschließend im Rahmen des Betriebssystems die Software installieren und letztendlich ausführen kann. Ein Nachteil bei dieser Virtualisierung ist der tendenziell hohe Aufwandsanteil der konstanten Hardware, der für eine virtuelle Maschine reserviert ist und dadurch dauerhaft blockiert wird. Selbst wenn die virtuelle Maschine nicht im Einsatz ist, kann die erforderliche Hardware nicht anderweitig verwendet werden.

In der virtuellen Maschine wird ein vollständiges Betriebssystem benötigt, welches eingerichtet, konfiguriert und verwaltet werden muss. Dies schließt unter anderem eine Konfiguration, Firewall und Virenschutz mit ein. Werden indes mehrere virtuelle Maschinen betrieben, ist der Aufwand diese Maschinen zu pflegen unter anderem ein ausschlaggebender Faktor, welcher in einer Kalkulation berücksichtigt werden sollte.

Containerisierung

Der Begriff Containerisierung[38] beschreibt den Ansatz, Hardware variabel für verschiedene Maschinen zu organisieren. Dabei sollen die Container möglichst oft wiederverwendet werden können. Die Containerisierung belegt im weiten Sinne eine Weiterentwicklung der Virtualisierung[38]. Damit ist gemeint, dass die Hardware derart abstrahiert wird, sodass einzelne Maschinen keine Kenntnis darüber haben, welche anderen Systeme außer der ihr zur Verfügung gestellten Hardware existieren.

Der Ansatz der Containerisierung baut folglich auf dem zuvor beschriebenen Ansatz der Virtualisierung auf. Darüber hinaus bietet die Containerisierung dem Betriebssystem eine konfigurierbare Umgebung. Somit erhält jeder Container seinen eigenen Kontext und lediglich eine Referenz auf ein Betriebssystem. Im Gegensatz zur Virtualisierung bedarf die Umsetzung mit Containern weniger Aufwand, da nicht jede “Maschine“ bzw. jeder Container ein vollständiges Betriebssystem benötigt. Aus diesem Grund lassen sich Container in der Regel innerhalb von Sekunden starten und beenden. Dabei hinterlassen diese weder reservierte Hardware als auch Daten auf einem Datenträger. Die Container werden oftmals *Wegwerf-VMs* genannt, da sie größtenteils zu einem vorher definierten Zweck gestartet werden und anschließend verschwinden. In der Regel wird für jede Applikation ein eigener Container definiert. Über die Konfiguration mittels der Host-Virtualisierung kann die konfigurierbare Umgebung parametrisiert werden, sodass einzelne Applikationen direkt miteinander kommunizieren.

Die Abbildung 2.2 zeigt eine Gegenüberstellung eines Ansatzes der Virtualisierung mit fest konfigurierten virtuellen Maschinen und der Containerisierung. Bei den einzelnen Applikationen in separaten Containern, hier am Beispiel von Docker[55] wird ersichtlich, dass diese lediglich Bibliotheken verwenden und keine eigenen Betriebssysteme benötigen.

In dieser Arbeit wird für die Containerisierung das Programm Docker verwendet. Docker besteht aus der in Abbildung 2.2 zu sehenden Docker Engine, Images und den Contai-

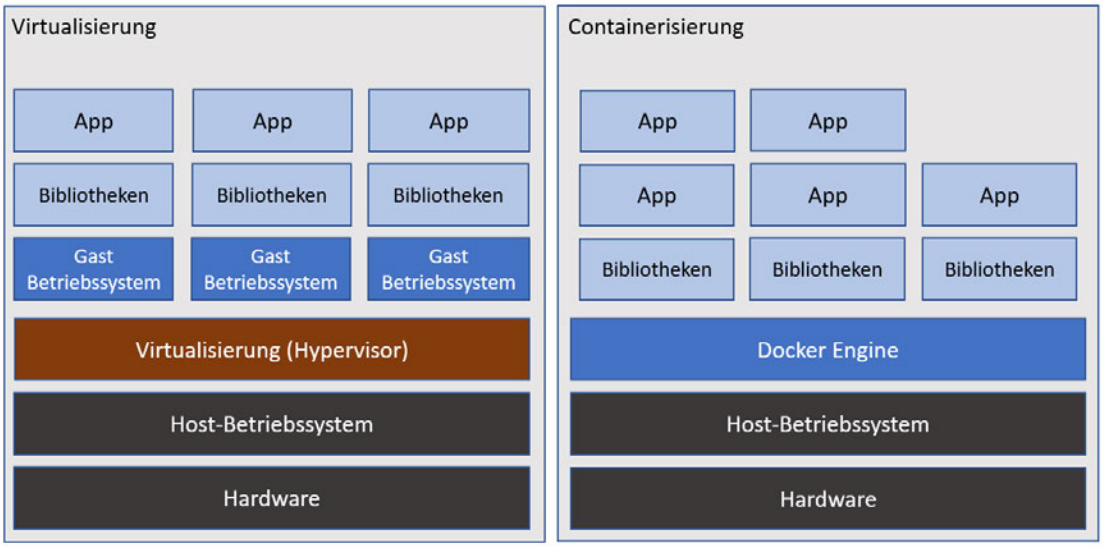

Abbildung 2.2: Gegenüberstellung von Virtualisierung und Containerisierung und deren Nutzung von Hardware.[41]

nern. Ein Image ist eine formale Beschreibung für eine Abbilddatei, die einer virtuellen Maschine gleicht. In diesem Image sind ebenfalls alle Abhängigkeiten, Bibliotheken und Hilfsprogramme vorhanden. Weiterhin enthält ein Image die Beschreibung der Applikation, welche in dem jeweiligen Container ausgeführt wird. Die Beschreibung des Images erfolgt in einem sogenannten Dockerfile. Aus einem Dockerfile baut die Docker Engine ein Image zusammen. Somit entsteht ein Container anhand eines von Docker Engine ausgeführtes Images. Ein Image kann demnach für beliebig viele Container verwendet werden, allerdings kann ein Container ausschließlich aus einem Image bestehen. Die verwendete Struktur des Dockerfiles und die daraus resultierenden Images sowie die Handhabung der Container für diese Arbeit befinden sich in Abschnitt 4.

Continuous Testing

In einer Welt in der Software weitreichend genutzt wird, ist es grundsätzlich notwendig Softwaresysteme laufend weiter zu entwickeln[12]. Insbesondere für Entwicklungsunternehmen ist es von immenser Wichtigkeit die Kontrolle über ihre Software zu behalten. Hauptsächlich ist für diese Unternehmen das Reagieren auf Fehler und Probleme innerhalb einer Software existenziell[12]. Es lässt sich in jedem Entwicklungsprozess eine Veränderung der vergangenen Jahren feststellen. Somit ist es heutzutage möglich mehrmals

täglich Updates für eine Software zu veröffentlichen. Die Intervalle zwischen einzelnen Updates für Programme haben sich zunehmend verkürzt[24]. Da die Veröffentlichungen in kürzeren Abständen folgen, ist eine stetige Angleichung der Entwicklungsprozesse unabdingbar. Das Continuous Testing (CT), Continuous Integration (CI) oder auch das Continuous Delivery (CD) bilden dabei die Basis für diese flexible Art der Softwareentwicklung. Das CI, CT und CD, welche für die Veröffentlichung von Software stehen, bilden dabei den Fokus einer sogenannten agilen Entwicklung[5]. Eine derartige Entwicklung gewinnt immer weiter an Bedeutung im Rahmen der Entwicklungsprozesse, da dieser Ansatz zum Vorteil hat auf Ereignisse kurzfristig reagieren zu können. Die Begriffe Continuous Integration, Continuous Testing und Continuous Delivery werden regelmäßig im Zusammenhang mit dem Begriff DevOps[7] verwendet. Der Begriff DevOps setzt sich aus den Bereichen Developement und Operations zusammen. DevOps bedeutet Wartung, Konfiguration und Erweiterung von CI,CT, oder CD Systemen und allen weiteren Tools, die Entwickler verwenden, um effizient zu arbeiten. Dadurch bleiben keine zusätzlichen Aufgaben, wie beispielsweise das Integrieren von Änderungen im Aufgabenbereich der Entwickler. Im Endeffekt können Firmen somit in der Entwicklung von Updates oder Bugfixes im Vergleich zu der ursprünglichen Herangehensweise schneller agieren und Software Updates häufiger und einfacher veröffentlichen.

Grundsätzlich existieren unterschiedliche Ansätze um CI, CT und CD in Softwareprojekten zu realisieren. Das womöglich bekannteste Tool für einen Ansatz ist Jenkins[40]. Jenkins ist ein Open Source Projekt und ist grundsätzlich für alle denkbaren Aufgaben, die automatisiert ausgeführt werden können, verwendbar. Dafür benutzt dieses Programm Pipelines, um automatisierte Arbeitsabläufe abzubilden. Diese Arbeitsabläufe umfassen sowohl simple Aufgaben, wie zum Beispiel dem Bauen eines Docker Images, als auch das Compilieren, Integrieren, Testen und Veröffentlichen von Software. Jenkins ist im Bereich der Automatisierungsserver im Vergleich zu ähnlichen Programmen bekannt. Der Grund hierfür liegt in der Flexibilität, die Jenkins mitbringt. Nahezu jede Aufgabe, die in einem Entwicklungsprozess anfällt, kann durch eine Aufgabe in einer Pipeline abgebildet werden. Dies hat zur Folge, dass die Ausgabe einer Pipeline ein unmittelbares Feedback über die geleistete Arbeit darstellt. Diese Rückmeldung ermöglicht, dass einen direkten Einblick in den Zustand eines Systems sowie die zusätzlich zugeordneten Informationen über die Ausführung des Systems. Dadurch wird erreicht, dass die Entwickler den Fokus auf die Entwicklung eines Produkts behalten. Indes wird ermöglicht, dass in einem Projekt lediglich Funktionalität in die Software integriert werden, die zuvor einen Durchlauf in der Pipeline (zum Beispiel: compilieren, integrieren, testen, überprüfen, verifizieren)

erfolgreich absolviert hat.

In diesem Zusammenhang ist jedoch ebenfalls GitLab[28] zu erwähnen, da mit dem GitLab CI/CD[28] Tool bereits ein Mechanismus existiert mit dem sogenannte Pipelines verwendet werden können. Die Verwendung von Pipelines kann dabei unmittelbar innerhalb des Versionskontrollsystems erfolgen. Weitere Tools sind TravisCI[15], CircleCI[14], Bitrise[11], TeamCity[83] und Atlassian Bamboo[6]. Diese Tools arbeiten jeweils auf unterschiedliche Weise um die Funktionalitäten bereitzustellen, die für CI, CT und CD genutzt werden können.

Letztendlich wird in dieser Arbeit jedoch Jenkins verwendet. Jenkins bietet eine vergleichsweise große Anzahl an Plugins, welche zusätzlich installiert werden können. Dadurch stehen eine Vielzahl an möglichen Zusatzfunktionalitäten für die Softwareentwicklung zur Verfügung.

2.2 Testen und Softwaretests

Die Definition eines Tests[24] benennt das methodische Überprüfen von Funktionalität und Qualität für Teil- und vollständige Produkte. Das Testen wird innerhalb von unterschiedlichen Organisationen bzw. Industrie- oder Berufszweigen wiederum unterschiedlich verstanden. Im Allgemeinen ist das Testen eine Voraussetzung für die Bestimmung und Einhaltung von Qualität. Um beispielsweise bei Konsumgütern ein qualitativ hochwertiges Produkt zu entwickeln, werden spezielle Tests eingesetzt, sodass ein gewisses Qualitätsniveau erreicht wird. In diesem Zusammenhang ist die Institutionen Stiftung Warentest[86] zu nennen, die als unabhängige Instanz Produkttests durchführt.

Der Begriff Testen[48][24] ist zugleich im Bereich der Informatik eine weitreichende Bezeichnung, da dieser in einer Vielzahl von unterschiedlichen Situationen angewendet werden kann. Beginnend von einer visuellen Überprüfung, inwieweit eine erwartete Prognose eintritt, bis hin zu Unit Tests, Integrationstest oder Systemtests findet diese Bezeichnung Anwendung. Das Testen beschreibt die Überprüfung eines Programms oder Softwaresystems auf korrekte Funktionsfähigkeit und Attribute nach denen eine Bewertung stattfinden kann.

Das Testen eines Programms ist in vielen Bereichen sogar Voraussetzung um eine Software im Endprodukt einzusetzen. In der Automobilindustrie beispielsweise darf keine Software innerhalb der Automobile eingesetzt werden, welches nicht zuvor nach einem

zertifizierten Verfahren getestet wurde. Metriken und Regeln für Software, die in der Automobilindustrie eingesetzt werden, sind zum Beispiel MISRA[4] Coding Standard Regeln und AUTOSAR[25] (AUTomotive Open System ARchitecture). Anhand dieser Regeln wird die Einhaltung eines qualitativ hochwertigen Programmcodes in der Software gewährleistet.

Wie in jedem Fachbereich treten in der Softwareentwicklung diverse Ansätze für das Testen auf. Zwei dieser Ansätze heißen "Testgetriebene Entwicklung (TDD)"[24] (*Test Driven Development*) und das V-Modell[24]. Der Name "Testgetriebenen Entwicklung" beschreibt bereits die Grundlage für diesen Ansatz. Bei der testgetriebenen Entwicklung werden zunächst die Tests aus den Requirements erstellt und anschließend die Implementierung gegen die Tests entwickelt. Diese Vorgehensweise soll den Programmcode einer Software dichter an die geforderten Requirements koppeln.

Im Unterschied zur testgetriebenen Entwicklung zeichnet der Aufbau einer Software nach dem V-Model eine besondere Spezifikation aus. Spezifisch in diesem Zusammenhang ist die Gegenüberstellung des Entwicklungsprozesses und die dazugehörigen Tests angeordnet als Buchstabe V. Obgleich die Entwicklungsschritte in einer V-Form angeordnet sind und den jeweiligen Detaillierungsgrad beschreiben, beinhalten die einzelnen Schritte unterschiedliche Verfahren der Softwareentwicklung. Die Abbildung 2.3 zeigt das V-Model und die darin vorkommenden vier unterschiedlichen Ebenen, die durch folgende Merkmale ausgezeichnet werden. Beginnend mit einer Anforderungsdefinition steht am Ende des V-Modells die Überprüfung dieser Anforderungen gegenüber. Es wird der Detaillierungsgrad beim Durchlaufen der Entwicklung kontinuierlich verfeinert und ist folglich beim Testen auf dem Weg nach *oben* durch die Ebenen mit Tests abgedeckt. Nach der zuvor genannten Anforderungsdefinition der Software auf der ersten Ebene folgt ein funktionaler Systementwurf. Die dritte Ebene stellt der technische Systementwurf und zuletzt die Komponenten Spezifikation dar. Die untere Spitze des V-Modells beschreibt die Implementierung der Software. An dieser Stelle entsteht ein Programmcode aus den zuvor genannten Schritten. Im Anschluss werden die einzelnen Ebenen überprüft. Nach der Implementierung knüpft ein Komponententest an, der die Komponenten Spezifikation verifiziert und überprüft. Der technische Systementwurf wird unterdessen anhand von Integrationstests überprüft. Das gleiche Vorgehen gilt für den funktionalen Systementwurf, der mittels Systemtests verifiziert wird. Wie bereits erwähnt erfolgt eine Abnahme für die Anforderungsdefinition zusammen mit der implementierten Software anhand eines Abnahmetests.

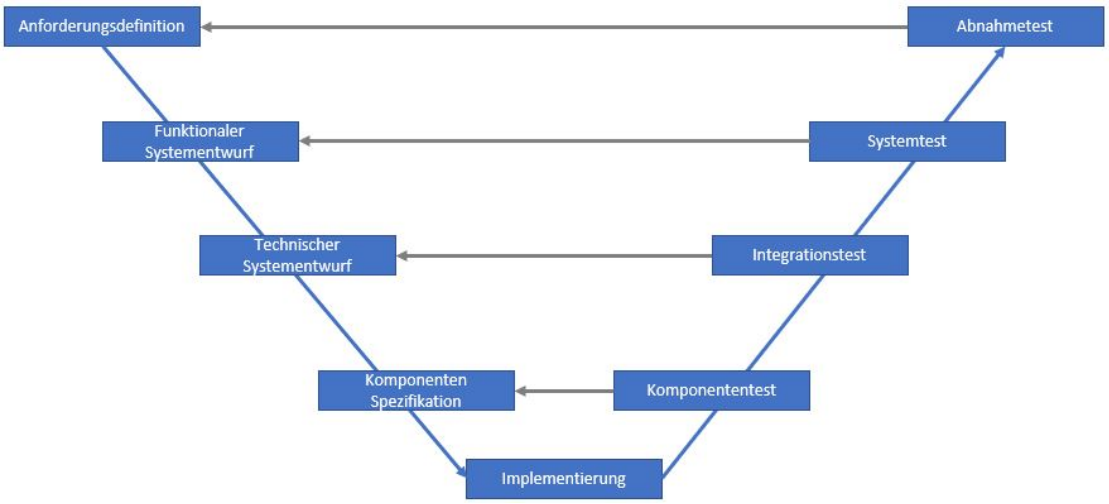

Abbildung 2.3: V-Modell Konzept in der Softwareentwicklung.[5]

In der Softwareentwicklung besteht ein Zusammenhang zwischen den verschiedenen Ebenen beim Entwickeln und Testen. In jeder Ebene bedarf es einer anderen Testmethodik. Für die Tests auf der funktionalen-/technischen Systementwurf Ebene beinhaltet die Testmethodik dabei sowohl einen System- als auch einen Integrationstest. Die Definition für Integrationstests sieht die Überprüfung und das Verifizieren einzelner verbundener Komponenten des Systems vor. Dabei sind die Komponenten bereits eigene, funktionsfähige Teile eines Systems. Anders als bei Unittest wird indes nicht die Funktion von einzelnen Programmzeilen oder Funktionen überprüft, sondern die Interfaces einer Komponente. Darauf aufbauend definiert ein Systemtest das komplette zusammengebaute System, welches aus allen Komponenten besteht und damit alle Integrationstests vereint. Diese Arbeit beschreibt anhand des gesteuerten Testsystems den Systemtest für das Beispiel QUIC. Die einzelnen Simulationen bilden dabei sogenannte Integrationstests, da diese lediglich Teile des Systems zusammen verwenden, welche das Omnet++ Testtool ausführt und bestimmte Funktionen und Komponenten überprüfen.

Die Abbildung 2.4 zeigt eine denkbare Vorgehensweise, die in der Softwareentwicklung eingesetzt werden kann um die unterschiedlichen Stadien der Entwicklung zu überprüfen und auf mögliche Fehlerquellen im Systemdesign zurück zuschließen. Dabei ist vorstellbar, dass sich in jeder der drei Stadien eine Fehlerquelle befindet. Die drei Stadien in der Abbildung beschreiben zuerst die Analyse, welche einen Analysefehler produzieren kann. Die zweite Stufe beinhaltet den Entwurf eines System aus dem sowohl Entwurfsfehler als auch Analysefehler resultieren können. Zuletzt die zeigt Implementierung, dass alle drei

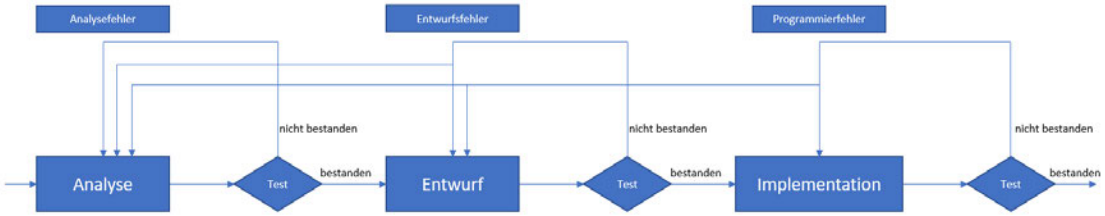

Abbildung 2.4: Ablauf von Tests und der Beschreibung möglicher Fehlerquellen.

Stufen der Fehler zu berücksichtigen sind, um mögliche Designfehler zu erkennen.

Eines der vergleichsweise großen auftretenden Probleme sowohl bei den Integrationstest als auch Systemtests ist das Erzeugen einer realen Umgebung. Eine solche Umgebung setzt voraus, dass die Software ausgeführt und gleichzeitig untersucht werden kann. Darunter fällt ebenfalls das kontinuierliche Testen und Verifizieren der bereits vorhandenen Software. Für die Entwicklung bzw. Weiterentwicklung des Testens von Software gibt es mittlerweile eine Vielzahl von Frameworks, die Funktionalitäten bereitstellen, welche das Ausführen von Tests unter bestimmten Aspekten ermöglichen. Für Unittest sind die bekanntesten Beispiele GTest[84], Cantata[29], Boost[2] oder JUnit[82]. Für die meisten Programmiersprachen existieren einige Testframeworks, bei denen die Tests fest eingebaut sind, wie beispielsweise Python oder Ruby.

Nicht nur für den Integrationstest sondern auch für den Systemtest steigt der Aufwand ein Framework zur Überprüfung einzusetzen. Die Software benötigt ein individuell anpassbares Gerüst, indem die Tests und die Software ausgeführt werden kann. Anders als beispielsweise bei Unittests, in denen unabhängige Frameworks vollständige Tools beschreiben, sind die Frameworks für Integrationstests und Systemtests enger an die Software gekoppelt. Eines der bekanntesten Frameworks für Integrationstest ist VectorCAST/C++[85]. Diese Frameworks sind in der Regel kostenpflichtig und bieten eine breite Spanne an Funktionalität an; beginnend bei herkömmlichen Unittests bis hin zu kompletten Abnahmetests.

Da die Tests in den meisten Fällen auf die speziellen Aspekte einer Software angepasst werden müssen, liefern Programme und Simulationsumgebungen, wie beispielsweise Omnet++[60], in einzelnen Fällen ein Testframework mit. Anhand dessen können innerhalb der Programmumgebung die Funktionalitäten für Testszenarien verwendet werden, die auf die Möglichkeiten der Umgebung begrenzt sind.

Das Ziel ist - neben dem Erstellen eines gesteuerten Testverfahrens - die Tests reproduzierbar zu gestalten. Die Reproduzierbarkeit beschreibt das Verhalten bei Tests, anhand dessen die Tests bei gleichen Eingaben stets dasselbe Ergebnis produzieren.

2.3 Transportprotokolle

Die Koordination des Datenverkehrs im Internet wird von Transportprotokollen[10][53] durchgeführt. Diese Transportprotokolle arbeiten auf der Transportschicht innerhalb von Modellen welche den Aufbau des Internets skizzieren. Die Beschreibung dieser Modelle wird auch Referenzmodell genannt. Die gängigen Referenzmodelle für den Aufbau des Internets sind das OSI und TCP/IP Referenzmodell. Die Abbildung 2.5 veranschaulicht anhand einer Gegenüberstellung der OSI und TCP/IP Modelle, welche Unterschiede innerhalb der einzelnen Modelle zu finden sind. Unter anderem zeigt die Abbildung in der linken Spalte die folgenden 7-Schichten des OSI Modells: Application, Presentation, Session, Transport, Network, Data Link und Physical.

Auf der rechten Seite der Abbildung ist das TCP/IP Modell zu erkennen, welches lediglich aus vier Schichten besteht. Diese Schichten heißen Application, Transport, Internet und Network Interface. Bei beiden Modellen sind Aufgaben für das Netzwerk Interface, die IP-Netzwerk Kommunikation und der Datentransport auf separaten Schichten angesiedelt. Auf den einzelnen Schichten werden Protokolle für die Erledigung spezieller Aufgaben eingesetzt. Somit arbeiten beispielsweise auf der Transportschicht sogenannte Transportprotokolle wie TCP, UDP, SCTP und das aktuell noch in der Standardisierung befindliche QUIC[39]. Hierbei ist QUIC ein Sonderfall, da es auf UDP aufsetzt und auf der Anwendungsebene arbeitet.

Das OSI Referenzmodell beschreibt, inwiefern einzelne Software- und Hardwarekomponenten innerhalb eines Netzwerks zusammen arbeiten und untereinander interagieren. Die Netzwerke können dabei sowohl ein lokales Netzwerk, ein Intranet als auch das gesamte Internet darstellen. Wie in Abbildung 2.5 zu sehen ist, definiert das OSI Modell sieben Schichten, namens Layer. Jede Schicht entspricht einer separaten Funktionalität, die in der Zusammenarbeit eine Kommunikation innerhalb des Internets ermöglichen. Abweichend vom OSI Modell sind TCP und IP jeweils Netzwerkstandards[9], welche einen Teil des Internets definieren. Diese beiden zentralen Internettechnologien sind generell für eine intakte Übertragung zuständig. IP ist dabei für die Adressierung und das Routing verantwortlich, wohingegen Transportprotokolle wie TCP, SCTP und QUIC für

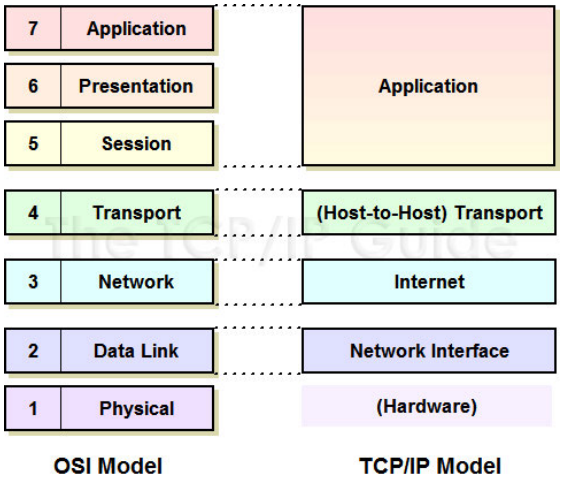

Abbildung 2.5: Gegenüberstellung des OSI und TCP/IP Referenzmodells.[9]

die möglichst verlustfreie und korrekte Kommunikation verantwortlich sind. Der wesentliche Unterschied zwischen den beiden Referenzmodellen ist die Verteilung der Aufgaben. Benötigt eine Anwendung im TCP/IP Modell eine Funktionalität, die über TCP oder IP Protokolle nicht geliefert werden kann, beinhaltet die Anwendung die erforderlichen Informationen. Im OSI Modell hingegen wird davon ausgegangen, dass eine Anwendung keinesfalls Funktionalitäten implementieren kann, welche durch einen vorherigen Layer bereits abgedeckt wurde.

2.4 QUIC Transportprotokoll

Eines der Protokolle, die auf der Transportschicht arbeiten, ist das sogenannte QUIC Transportprotokoll. Dieses Protokoll ist zum Zeitpunkt dieser Arbeit noch in der Phase der Standardisierung bei der IETF[80] und im Vergleich zu weiteren Protokollen ein Sonderfall. Das QUIC Protokoll ist deshalb besonders, da es auf der Transportschicht das UDP Protokoll verwendet und alle weitere Funktionalität in der Applikation abgebildet werden kann.

Das QUIC Protokoll ist für die Verbesserung der Performance bei HTTPS Datenverkehr mit Verschlüsselung, Multiplexing und niedrigen Latenzen entwickelt worden[39]. QUIC verbessert die Performance anhand von Funktionen, wie Multiplexing, und anhand von unmittelbar im Handshake initialisiertem TLS[1] für die Übertragung. Das QUIC-Protokoll bildet dabei die Verbindung zwischen dem direkten HTTPS in der Applikation

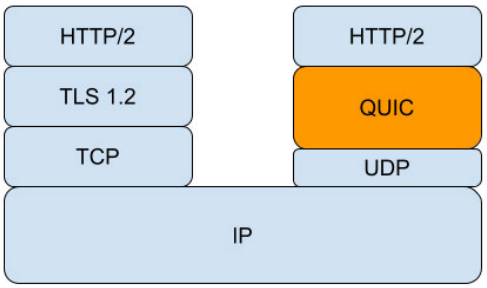

Abbildung 2.6: Gegenüberstellung des TCP Stacks und dem auf UDP aufgesetztem QUIC

und dem IP Layer. Die Abbildung 2.6 zeigt einen Vergleich zwischen dem TCP Applikation Stack und dem von QUIC. Obwohl die beiden Seiten in der Abbildung für QUIC und dem TCP ähnlich aussehen, kann QUIC bereits im Handshake die TSL-Verschlüsselung mit einbeziehen ohne dafür zusätzliche Nachrichten zu benötigen. Der TCP benötigt hierfür mehrere Nachrichten, insbesondere da die Funktionalitäten auf unterschiedlichen Schichten angesiedelt sind und dadurch nicht kombiniert werden können.

Die maßgebliche Änderungen von QUIC gegenüber TCP sind der Connection Handshake[39] und das Multiplexing. TCP nutzt einen sogenannten 3-Wege Handshake für den Verbindungsaufbau. Nach dem Handshake werden bei TCP zusätzliche Nachrichten benötigt um die Verbindung mit TLS zu verschlüsseln. QUIC verwendet für diesen Aufbau einer mit TLS verschlüsselten Verbindung lediglich einen Nachrichtenaustausch. Der kurze Nachrichtenaustausch entsteht durch Reduzierung der Anzahl von Nachrichten, die für das Aufbauen einer Verbindung benötigt wird. Eine Latenz[1] reduziert die Übertragungszeit von Netzwerknachrichten. Ermöglicht wird die Reduzierung der Nachrichten, da QUIC auf dem UDP aufgesetzt ist und alle weiteren Funktionalitäten in der Anwendung angesiedelt sind. Dieser eine Nachrichtenaustausch für den Verbindungsaufbau ist deshalb möglich, da TLS bei QUIC bereits integriert ist. Bei bekannten Parteien einer Verbindung kann sogar ein 0-RTT Handshake(0-Round Trip Time) verwendet werden, bei dem bereits im Handshake Paket Nutzdaten gesendet werden. Ein weiterer Vorteil von QUIC ist das Verteilen einer Verbindung über mehrere Känale für den Datenaustausch. Durch das Multiplexing sind Probleme aus TCP, wie das head-of-line blocking[70], umgangen und bieten mehr Performance für die Datenübertragung.

[1] Latenz beschreibt in diesem Zusammenhang die Zeit, die die Daten für die Übertragung zum Server benötigen. Berechnet wird die Latenz durch die Zeit vom Senden bis zum Eintreffen einer Antwort.

2.5 Simulationsumgebung für Transportprotokolle

Eine Simulationsumgebung hat als Aufgabe eine definierte Umgebung für eine Software unter Test bereitzustellen[60]. Dabei beschreibt die Simulationsumgebung für die Software ein Abbild der realen Umwelt. Ein positiver Nebeneffekt dabei ist, dass eine Software keine Veränderungen benötigt, um innerhalb einer Simulationsumgebung getestet zu werden. Der Aufbau einer Simulationsumgebung ist mithin ein Aufwand, der unbedingt in Projekten mit einkalkuliert werden muss. Für vergleichsweise kleine Software sind die dafür erforderlichen Entwürfe grundsätzlich durch bestehende Frameworks umsetzbar. Das bedeutet es bedarf einen relativ geringen Aufwand um einen Prototypen einer Software zu erstellen. Bei größeren Projekten hingegen bzw. bei Software, die nicht innerhalb einer definierten Umgebung arbeiten, gestaltet sich das Erstellen von Entwürfen komplexer. Ein Begriff, der in diesem Zusammenhang häufig verwendet wird, ist die Reproduzierbarkeit. Reproduzierbarkeit bedeutet, dass Testfälle bei unveränderten Eingaben innerhalb eines Tests fortwährend dieselben Ergebnisse liefern. Dies führt dazu, dass Ergebnisse bei Testfällen in einer realen Umgebung abhängig von vorhandenen Umständen zur Laufzeit abweichen können.

Ein Beispiel, welches an dieser Stelle zu nennen ist, sind Transportprotokolle. Diese Protokolle, wie in Abschnitt 2.3 beschrieben, sind komplexe und von den Übertragungssignalen abhängige Technologien. Da diese einen essentiellen Beitrag zur aktuellen Funktion des Internets leisten, sind diese ausreichend zu testen. Jedoch arbeiten im Internet mehr als ein Protokoll zur gleichen Zeit. Diese Tatsache führt dazu, dass einzelne Testfälle beim wiederholten Ausführen unterschiedliche Ergebnisse produzieren können. Dies geschieht, da beispielsweise die Datenpakete von anderen Transportprotokollen dafür sorgen, dass in einem Testdurchgang ein Timeout aktiv wird, der in vorherigen Durchgängen nicht aufgetreten ist. Das unterschiedliche Verhalten zeigt sich dadurch, dass Datenpakete den Empfang oder das Senden blockieren bzw. verzögern können und bei einem erneuten Durchgang der Tests anschließend somit nicht mehr involviert sind. Dadurch ensteht ein externern Zustand in Komponenten, den man eigentlich vermeiden will.

Neben der Schaffung einer konkreten Arbeitsumgebung dient eine Simulationsumgebung außerdem dem Bereitstellen zusätzlicher Funktionalitäten. Am Beispiel der Transportprotokolle ist dies die Funktionalität den Datenverkehr des Internets abzubilden und zu verwalten. Das heißt, für das Protokoll wirkt die Simulationsumgebung wie eine reale Implementierung, da alle die das Transportprotokoll betreffenden Technologien ebenfalls vorhanden sind. Die bekanntesten Simulationsumgebungen heißen NS2/NS3[59] und

Omnet++[60]. Das in dieser Arbeit verwendete Omnet++ bietet den Vorteil, dass es zusammen mit dem INET Framework bereits alle Technologien für die Transportprotokolle mitbringt. Da Omnet++ eine eventbasierte Simulationsumgebung ist, eignet sich diese besonders für den Einsatz von nachrichtenbasierten Technologien, da message passing[2] die grundlegende Technologie darstellt. Omnet++ verwendet diese Nachrichten für Pakete, Nachrichten und Kommunikationen zwischen den einzelnen Modulen. Folglich wird bereits ein Paket für die Netzwerkkommunikation in der Implementierung von Omnet++ mitgeliefert. Die Funktionalität von Timing und Interrups wird dabei über die Möglichkeit von *self messages* bereitgestellt. Die self messages besitzen die Fähigkeit sich selbst zu benachrichtigen und über ein aufgetretenes Verhalten zu informieren.

Wie bereits im Abschnitt 2.2 hervorgehoben, bringt Omnet++ zudem ein integriertes Tool mit, welches das Ausführen von Tests in erstellten Simulationen möglich macht. Dieses Tool heißt *opp_test* und ist ursprünglich entwickelt worden, um den Software Kern von Omnet++ zu testen. Die Besonderheit dieses Tool ist, dass es ohne eine Injektion von zusätzlichem Programmcode in der Software auskommt. Generell arbeitet das Testtool auf die Weise, dass eine definierte Simulation ausgeführt wird und im Nachhinein die Aufgaben in den Logdateien nach bestimmten aufgetretenen Ereignissen durchsucht werden. Ebenfalls kann überprüft werden ob bestimmte Ereignisse bei einem Datenaustausch einer Simulation auftreten.

Mit der Beschreibung von Begrifflichkeiten und Besonderheiten, sowie dem Testen, Softwaretests und der Einführung von Transportprotokollen, hier insbesondere dem QUIC Protokoll sind alle benötigten Grundlagen vorhanden um einen genauen Blick auf die Technologien zu werfen.

[2] Message passing (ENG) oder auch Nachrichtenaustausch beschreibt die Kommunikation zwischen Parteien durch ein definierten Satz an Nachrichten, bei denen Daten angehängt werden können, um Informationen zu übertragen.

3 Analyse

Das nachfolgende Kapitel stellt die Analyse dar, die auf die aktuelle Problemstellung - Erstellung eines gesteuerten Testverfahrens im Kontext von Transportprotokollen - eingeht. Eine genaue Betrachtung der Problemstellung wird gefolgt von einer Abgrenzung zu verwandten Forschungen. Diese zeigen, welche Themen bereits auf diesem Gebiet untersucht wurden. Danach folgt eine Auflistung der Anforderungen und eine Betrachtung der Machbarkeit für die Umsetzung eines gesteuerten Testverfahrens.

Im Anschluss wird ein detaillierte Betrachtung auf die Simulationsumgebung gerichtet, wie mit dieser Umgebung gearbeitet wird und zugleich wie innerhalb eines Frameworks am Beispiel von INET[61] neue Transportprotokolle implementiert werden können. Anschließend folgt die Antwort auf die Fragestellung, wie das Testen innerhalb einer Simulationsumgebung funktioniert und welche Vor- und Nachteile ein bereitgestelltes Testtool, wie das *opp_test* Tool, mitbringt. Im weiteren Verlauf zeigt die Analyse in welchem Entwicklungsstand die Implementierung für das QUIC Transportprotokoll im INET Framework ist. Diese Frage bestimmt die Auswahl der Testszenarien für die Testfälle. Am Ende des Kapitels schließt eine Einordnung in den Forschungskontext eines solchen Testverfahrens die Analyse ab.

3.1 Problemstellung

Das Internet ist eine globale Technologie mit dessen Hilfe Datenpakete rund um den Globus innerhalb von Millisekunden verschickt werden.[9][52] Um diesen Informationsfluss gewährleisten zu können, ist es notwendig, dass dafür erforderliche Technologien perfekt zusammenarbeiten. Das bedeutet somit, dass selbst bei Fehlern oder Datenverlusten in der Übertragung stets ein Informationsfluss sichergestellt sein muss. Insbesondere in Zeiten der Digitalisierung, in der beispielsweise Zugsignale, medizinische Geräte und IoT Geräte[3] über das Internet gesteuert werden, müssen diese relevanten Technologien zuverlässig funktionieren.

Die auf der Transportebene des OSI- oder TCP/IP-Referenzmodells arbeitenden Protokolle sind dabei eine fundamentale Komponente, da diese die Übertragung der Datenpakete kontrollieren und sicherstellen[52]. Um Transportprotokolle zu testen werden unterschiedliche Frameworks eingesetzt, sodass möglichst alle Bereiche der Software durch Tests[48] abgedeckt sind. Grundsätzlich ist das Testen eines komplexen Systems wie ein Transportprotokoll im realen Betrieb aufwändig, da viele Faktoren wie Daten von anderen Transportprotokollen die Test verfälschen können. Der Grund dafür ist, dass eine Vielzahl an Protokollen parallel arbeiten und unterschiedliche Daten über das Internet verschicken bzw. empfangen. Daher sind Tests für Transportprotokolle im realen Betrieb nicht unmittelbar reproduzierbar.[43]

Eine mögliche Lösung für dieses Problem stellt eine Simulationsumgebungen wie Omnet++/INET[60] oder NS2/NS3[59] dar. Mit Hilfe einer solchen Simulationsumgebung sind die Systemgrenzen bekannt und der Einfluss von störenden Signalen kann insoweit außer Acht gelassen werden.

Die Wahl des Testframeworks kann eine Entscheidung über die Wahl der Simulationsumgebung ebenfalls beeinflussen. Der Unterschied für die Implementierung und einer Systemintegration über das erwartete Ziel schränkt die Wahl eines Testframeworks ein. Wie am Beispiel des Grundprojektes[21] *Automatisiertes System zum Ausführen von Tests in einer Pipeline* zu sehen ist, lässt sich auch ein Testframework wie *packetdrill*[13] einsetzen um innerhalb einer Simulation Tests durchzuführen. Eine andere Variante gegenüber dem *packetdrill* Framework ist der Einsatz eines *opp_test* Tools, welches direkt von der Simulationsumgebung bereitgestellt wird.

Das Testen soll nach Möglichkeit bereits während des Entwicklungsprozesses betrieben werden. Um den einzelnen Entwicklern ein möglichst kurzfristiges Feedback über ihre Implementierung bieten zu können, werden automatische Testsysteme eingesetzt. Bei jeder Änderung innerhalb der Implementierung eines Transportprotokolls wird das gesteuerte Testverfahren angeregt und die hinterlegten Tests ausführt und ausgewertet. Wichtig dabei ist nicht nur, dass eine neue Funktionalität getestet wird sondern auch die Frage, ob mit den neuen Änderungen keine bestehende Funktionalität beschädigt wurde. Weiterhin ist fraglich, welchen zusätzlichen Umfang ein gesteuertes Testverfahren benötigt um während einer laufenden Entwicklung die Implementierung eines Transportprotokolls kontinuierlich mit einem automatischen System zu überprüfen. Dazu ist - beginnend von dem Schreiben bis hin zu der ersten Implementierung von Tests - für die Funktionalität

ebenfalls entscheidend, dass ein automatisiertes Testverfahren die Umgebung kontinuierlich im gleichen Umfang aufbaut und die Tests ausführen kann.

Als Vergleich über einen Nutzen des automatisierten Testsystems soll im Folgenden ein beispielhafter Arbeitstag eines Entwicklers schematisch betrachtet werden. Dabei liegt das Augenmerk auf solche Szenarien, in denen ein Entwickler eigenständig arbeitet und somit die Fehleranalyse betreibt. Dazu gehört, dass der Entwickler anhand eines gesteuerten Testverfahrens unterstützt werden kann, indem er alsbald Feedback in Bezug zu problematischen Änderungen erhält.

3.2 Verwandte Arbeiten

Das QUIC Transportprotokoll baut auf dem auf der Transportschicht angesiedelten UDP auf. Anders als TCP ist UDP ein verbindungsloses Transportprotokoll. Damit sind Probleme wie die *slow start* Strategie[91] nicht für die Performance im weg. Eine Evaluation der Performance von UDP beschreiben die Autoren Zhaojuan, Yue und Yongmao Ren, Jun Li in ihrem Paper: *Performance Evaluation of UDP-based High-speed Transport Protocols*[91] Dort werden vier unterschiedliche auf UDP basierende Protokolle verwendet um die Performance zu analysieren. So kann dieses UDP Protokoll durchaus eine vergleichsweise hohe Performance für QUIC bereitstellen.

Da QUIC sich zum Zeitpunkt dieser Arbeit noch in der Standardisierung bei der IETF befindet gibt es verschiedenste Untersuchungen zu diesem Transportprotokoll. In dem Paper: *Is QUIC becoming the New TCP? On the Potential Impact of a New Protocol on Network Multimedia QoE* beschreiben die Autoren den wachsenden Einfluss von QUIC welcher bereits als Standard für Google Services eingesetzt wird[73]. Die Autoren zeigen eine Untersuchung der QoE Möglichkeiten[45] und in welchem Umfang diese spürbar sind. Dafür wurden Messungen auf QoE Level bei QUIC und TCP durchgeführt welche als Ergebnis lieferten das QUIC eine signifikant höhere Performance für das Video Streaming bietet.[73] Allerdings muss hier ebenfalls wie bei allen Protokollen sichergestellt werden das diese ausgiebig getestet sind um alle Vorteile mit voller Wirkung nutzen zu können. Da auf der Transportschicht viele Protokolle gleichzeitig den Datenverkehr regeln vereinfacht es das Testen indem man eine definierte Umgebung für ein Protokoll aufbaut.

Eine Möglichkeit für das Testen von Software wie Transportprotokollen in einer abgeschirmten Umgebung sind Simulationsumgebungen. In diesen können reale Technologien ausgeführt werden ohne dabei durch Störfaktoren wie zusätzliche Daten von anderen Programmen behindert zu werden. Da das Testen solch einer Technologie in einer definierten Umgebung die Testszenarien wiederholbar und reproduzierbar macht werden hier diverse Ansätze verfolgt um dieses Testen zu gestalten. Die Autoren M. Miegler und W. Wolz beschreiben die Ansätze eines realen Testsystems als kostenintensiv, durch limitierten Zugriff und Verfügbarkeit zur Ausführung als einen hohen Druck auf die Entwickler. Was auch in vielen fällen zu Verzögerung in der Testentwicklung führt.[56] Als eine Lösung beschreiben die beidem in dem Paper die Entwicklung von Testprogrammen innerhalb einer virtuellen Testumgebung oder auch einer Simulationsumgebung. Dadurch werden viele direkte Möglichkeiten sichtbar, welche kontinuierliches überprüfen von Software ermöglichen.

Das Gebiet der sogenannten Continuous Integration (CI), Continuous Delivery (CD) oder auch Continuous Testing (CT) hat in den letzten Jahren immerzu an Popularität gewonnen. Ein Grund dafür ist die Geschwindigkeit mit der Unternehmen Updates und Änderungen in ihrer Software veröffentlichen.

Dabei bleibt das Ziel immer dasselbe: Sich wiederholende Arbeiten, die viele Schritte beinhalten, zu automatisieren und mithilfe eines Systems steuern zu lassen. Ein solches System beschreibt Mitesh Soni in dem Paper *End to End Automation On Cloud with Build Pipeline: The case for DevOps in Insurance Industry*[74] unter dem Oberbegriff DevOps[7]. Der Begriff DevOps beschreibt ein Mindset nachdem die Bereiche Entwicklung und Operations zusammen integriert werden.

Die Bezeichnung DevOps erweitert den agilen Ansatz wie SCRUM[16] für Softwareentwicklungen um die Mentalität Arbeitsprozesse zu automatisieren und Entwicklungszyklen kürzer zu gestalten. In dem zuvor genannten Paper definiert Mitesh nicht nur die einzelnen Aufgaben, wie zum Beispiel das Programmieren, Kompilieren, Integrieren und Testen, sondern stellt zudem dar, in welcher Weise durch den Einsatz eines Systems automatisiert gearbeitet werden kann. Die Automatisierung führt mithin zu einer vergleichsweise effizienteren und zuverlässigeren Arbeit[74]. Im Grunde beschreibt das Paper die Grundlage für den Einsatz eines gesteuerten Testsystems um die Arbeitsweise der Entwickler zu vereinfachen und zu unterstützen. Zusätzlich soll dadurch zugleich die Qualität der Software gesteigert werden indem kontinuierliches Überprüfen des Programmcodes Fehler, ausschließen. Diese Grundlage wird in dieser Arbeit als Hypothese herangezo-

gen unter der detaillierten Betrachtung die Arbeit von Entwicklern zu vereinfachen und mithilfe des Einsatzes eines gesteuerten Testverfahrens zu beschleunigen. Solch ein Testverfahren fällt somit innerhalb einer CI unter den Begriff Continuous Testing. In dieser Arbeit wird ausschließlich jeglicher Teil des Testens angewendet, den Mitesh innerhalb seiner Ende-zu-Ende Automatisierung beschreibt.

Ein weiteres Paper von Peter Zimmerer mit dem Titel *Strategy for Continuous Testing in iDevOps*[92] erklärt das Continuous Testing als einen essentiellen Part um schnellstmöglich Software mit steigender Qualität zu entwickeln. Dies soll erreicht werden, indem Fehler durch die Variable "Menschäusgeschlossen werden, da die Prozesse vollständig automatisiert ablaufen. Zimmerer beschreibt diese Strategie am Beispiel der Industrie. Laut Zimmerer bietet die Industrie nicht ausschließlich Vorteile bei der Verwendung einer reinen Softwarelösung. Er beschreibt dieses Problem da in der Industrie immer die physischen Produkte als relevanter Faktor bestehen und nicht alles durch eine Software abgebildet werden kann. Doch letztendlich ist die Aufgabe der Automatisierungen mit dem DevOps Mindset stets eine Verbesserung der Softwarelösungen zu erreichen.

Somit bilden die beiden zuvor genannten Paper die Basis für diese Arbeit. Die Aussagen von Mitesh und Zimmerer geben umfassend die Bedeutung eines Continuous Testing Ansatzes wieder. Mitesh analysiert den kompletten Automatisierungsprozess und Zimmerer den Bereich des Testens. Beide Autoren fassen das Testen als Basis für steigende Qualität und schnellere Bearbeitung der Auslieferungen von Software zusammen.

3.3 Anforderungen und Machbarkeit

Im Folgenden werden Anforderungen an ein Testsystem und die in diesem Zusammenhang mögliche Machbarkeit dargelegt. Grundsätzlich ist zu prüfen, welche Anforderungen an ein solches Testsystem gestellt werden. Als nächstes folgt die Frage, inwieweit die Umsetzung nach den zuvor aufgestellten Anforderungen möglich ist. Eine der zentralen Anforderungen ist das vorhandene Verständnis über die benötigten Technologien eines Testsystems. Ein solides Wissen über die Vorgehensweise und den Umfang der Technologien, die anhand von Referenzmodellen arbeiten und schließlich das Internet ermöglichen, wird demnach vorausgesetzt. Dieses Knowhow schließt die Transportprotokolle ein, welche auf der Transportschicht dieser Referenzmodelle agieren und für die Koordinierung der Datenübertragung[9] zuständig sind.

Die auf der Transportschicht arbeitenden Protokolle können jeweils bestimmte Probleme unterschiedlich lösen, was Abhängig von der Problemstellung effizienter bei der Datenübertragung ist. Somit definiert TCP ein Ende-zu-Ende Protokoll[9], bei dem sichergestellt wird, dass die Daten vollständig und in der gesendeten Reihenfolge beim Empfänger ankommen. UDP hingegen ist ein verbindungsloses Protokoll[46], wodurch Daten gesendet werden ohne Sicherstellung, dass diese Daten überhaupt, vollständig und zugleich in der richtigen Reihenfolge ankommen. Als Folge bietet UDP eine kürzere Übertragungszeit obgleich es zusätzlich mögliche Datenverluste in Kauf nimmt. Das neue QUIC Protokoll ist auf UDP aufgebaut und agiert auf der Anwendungsebene der Referenzmodelle.

Die Grundlagen für die Entwicklung eines Transportprotokolls bieten in der Regel, die von der IETF Standardisierung veröffentlichten RFCs[39][77] zu den Protokollen. Ein RFC ist eine detaillierte Beschreibung einer Technologie. Darin sind die Mechanismen beschrieben und erläutert an die sich solches Protokoll halten sollte. Im Falle von QUIC ist der Standardisierungsprozess nicht abgeschlossen, da zum Zeitpunkt dieser Arbeit lediglich die Draft Versionen zum QUIC Protokoll[39] existieren. Als Basis der Implementierung in INET wurde die Draft Version 10 festgesetzt. Im Rahmen dieser Arbeit wird nicht auf das Thema Implementierung sondern das für die Implementierung notwendige Testverfahren fokussiert. Im Folgenden wird deshalb das QUIC Protokoll und die in dieser Arbeit verwendete Omnet++/INET Simulationsumgebung analysiert.

Eine weitere Voraussetzung ist ein fundiertes Wissen zum Thema Testen[48]. Das bedeutet es ist zu prüfen, inwieweit im statischen oder dynamischen Testverfahren geprüft werden kann. Statische Tests werden prinzipiell direkt auf der Implementierung durchgeführt und besitzen die bereits erwähnten Metriken, wie das Code Coverage[89]. Unter anderem können dabei zudem Reviews, Pair-Programming oder ähnliche statische Codeanalysen in Form von Formatierung wie Clang Format[81] eingesetzt werden. Beim dynamischen Testen stellt sich sie Frage, ob die Tests im White-Box[58] oder Black-Box[58] Verfahren getestet werden. Diese Frage beschäftigt sich dahingehend, ob ein Einblick in die Implementierung gewährt wird um spezielle Funktionen und Objekte unmittelbar zu testen. Bei einem vollständigem Zugang zum Programmcode einer Software wird von White-Box Tests gesprochen. Black-Box Test hingegeben beschreiben das Testen der Schnittstellen einer Software, da hierbei ausschließlich Input- und Output-Schnittstellen getestet werden können. Die konkrete Implementierung bleibt bei den Black-Box Tests unbekannt. In Bezug zu dieser Arbeit sind die Grenzen zwischen den Ressourcen für die Testfälle nicht eindeutig. Da ein Einblick in das Erstellen der Simulationen und die unmittelbare Verwendung der Implementierung des QUIC Transportprotokolls erfolgt, kann in

diesem Zusammenhang von einer Art Glas-Box Tests gesprochen werden. Die Glas-Box Tests[1] beschreiben das Testen der Schnittstellen, dementsprechend die Kommunikation der einzelnen Funktionalitäten untereinander.

Insgesamt sind die Anforderungen dahingehend zu überprüfen in welchem Umfang diese eingesetzt werden können um ein konkretes Ziel zu verfolgen und nicht eine generalisierte Lösung für alle Probleme zu erarbeiten. Daher wird im Rahmen dieser Arbeit ein Demotestfall erstellt um das gesteuerte Testverfahren aufzubauen und zu überprüfen. Das Augenmerk soll folglich nicht auf das vollständige Testen des Transportprotokolls liegen; es liegt vielmehr auf der Grundlage eines vollständigen Durchgangs des Testverfahrens um die genannte Hypothese zu überprüfen. Dabei sollen die Ressourcen effektiv eingesetzt werden und zudem ebenfalls der Einsatz einer Testmethodik effizient umgesetzt werden. Die Simulationsumgebung eignet sich dafür besonders, da ein Netzwerk mit festen Parametern und Funktionen erstellt werden kann, das die Tests reproduzierbar und gleichzeitig übersichtlich gestaltet. Für diese Umgebung sind keine anderen Daten von beispielsweise parallellaufenden Protokollen vorhanden, welche eine Auswertung erschweren. Aus diesem Grund können bestimmte Verhalten in der Simulation für einen Testfall explizit provoziert werden um die Reaktion des Transportprotokolls zu untersuchen.

3.4 Umfrage an Entwickler

Zusätzlich zu den gesammelten Daten soll die in dieser Arbeit untersuchte Hypothese anhand einer Umfrage überprüft und belegt werden. Dazu werden insgesamt 16 Softwareentwickler zu dem Feedback, das mit einem Testverfahren generiert werden kann, befragt.

Insgesamt werden den Befragten 12 Fragen gestellt, die in zwei Kategorien unterteilt sind. In der ersten Kategorie befassen sich die Fragen mit den Aufgaben von Entwicklern, welche an einer gemeinsamen Codebasis arbeiten. Die Bewertung erfolgt mit einer Skala von $1-5$. Für diese erste Kategorie bedeutet 1 - *ich stimme nicht zu* und 5 - *ich stimme voll zu*. Bei der zweiten Kategorie steht der Aufwand mit bzw. ohne ein gesteuertes Testverfahren im Vordergrund. Diese soll durch den Befragten geschätzt werden. Hier stellt die 1 - *niedriger Aufwand* und die 5 - *sehr hoher Aufwand* dar. Die Befragten

[1] Glass-Box Tests sind zwischen White-Box und Black-Box Tests angesiedelt. Es beschreibt die Testfunktionalität der Black-Box Tests zusammen mit einem Einblick in die Implementierung und die möglichen Interfaces um bestimmte Testszenarien zu generieren.

erhalten vor Beginn der Befragung eine Einführung in den Aufbau und das Ziel des gesteuerten Testverfahrens. Dazu gehört die Erläuterung eines Szenarios, welches als Vergleich für die Auswertung mit und ohne solch ein gesteuertes Testverfahren verwendet wird. Das Szenario wird in Kapitel 4 detailliert beschrieben und gegen das Testverfahren abgegrenzt. Auf der Grundlage dieses Szenarios sollen die Befragten anhand ihrer eigenen Erfahrungen die folgenden Fragen in der $1 - 5$ Skala bewerten.

Fragen der Kategorie 1 (A) und Kategorie 2 (B):

A1 Sie arbeiten oft mit Sourcecode Management für die tägliche Arbeit.

A2 Fehlerhafte Software sollte schnellstmöglich wieder repariert werden.

A3 Viele Entwickler bedürfen einer Koordination des Sourcecode Management.

A4 Problemfindung dauert lange.

A5 Sie haben schon mal einen Fehler gesucht ohne die zugehörige Änderung zu kennen.

A6 Sie wären mit dem Feedback eines gesteuerten Testverfahrens schneller gewesen.

A7 Würden Sie ein Testverfahren einsetzen für die Überprüfung und das Feedback?

B1 Wie hoch schätzen Sie den Aufwand für die Fehlersuche im Allgemeinen?

B2 Wie hoch schätzen Sie die Arbeitszeitersparnis wenn Sie die zugehörige Änderung kennen?

B3 Wie hoch schätzen Sie den Aufwand der Fehlersuche mit so einem gesteuerten Testsystem?

B4 Wie hoch ist der Mehraufwand durch ein Testsystem wenn wenige Entwickler an einer Software arbeiten?

B5 Wie hoch ist der Mehraufwand durch ein Testsystem wenn viele Entwickler an einer Software arbeiten?

Die Befragung der Entwickler soll aufzeigen, wie die Meinung zu einem effektiven Einsatz von Ressourcen und effektiven Einsatz eines Testverfahren aussieht. Dazu sollen die Meinung der Entwickler über die beiden Unterschiede, welche mit und ohne ein gesteuertes Testverfahren bestehen aufgezeigt werden. Insbesondere soll gezeigt werden wie die Erfahrungen der Entwickler den effektiven Einsatz von Ressourcen bewerten indem sie Zustimmen oder Ablehnen zu den zuvor gestellten Fragen. Zudem sollen die Bewertungen zu einer Aufwandsschätzung benötigte Ressourcen für die jeweiligen Szenarien beschreiben. Es existiert der Zusammenhang von Aufwand zu benötigte Ressourcen, da ein hoher Aufwand einen erhöhten Einsatz von Ressourcen (Arbeitszeit oder Geld in Form von Hardware/Software) bedarf.

3.5 Simulationen und Simulationsumgebung

Der Begriff Simulation bezeichnet im Allgemeinen den Versuch reale Szenarien abzubilden. Dabei wird eine Abstraktion der realen Bedingungen und Abläufe aus der Realität analysiert und innerhalb der Simulation nachgebildet. Die Nachrichten-basierte Simulationsumgebung Omnet++[60] bieten zusammen mit dem INET Framework[61] die Möglichkeit Netzwerke und deren Systeme nachzubilden und anschließend zu untersuchen. Das INET Framework ermöglicht Transportprotokollen wie TCP, SCTP und dem noch nicht vollständig implementierten QUIC, dass diese innerhalb der Simulationsumgebung nutzbar gemacht werden.

Die Simulationsumgebung Omnet++ ist eine Nachrichten-basierte Umgebung und verwendet Message passing für die Kommunikation. Die einzelnen Module innerhalb der Implementierung eines Transportprotokolls sind hierarchisch aufgebaut, denn dadurch können einzelne Module angeordnet und gruppiert werden. Anderenfalls würden die einzelnen Bereiche der Software unüberschaubar und überdurchschnittlich groß werden, da die vollständige Implementierung enthalten ist. Das kleinste aktive Modul aus dieser Umgebung wird in diesem Zusammenhang als *simple module* bezeichnet und bildet die kleinst mögliche Funktionskomponente. Diese besonderen Module können letztendlich sowohl zu sogenannten *compound modules* als auch in unendlichen Schichten innerhalb der Hierarchie kombiniert und zusammengefasst werden. Die Abbildung 3.1 aus dem Omnet++ Manual zeigt die zuvor genannte hierarchische Anordnung von *simple modules* und den gruppierenden *compount modules*. Die nachrichtenbasierte Kommunikation befindet zwischen den einzelnen Modulen. Dabei existiert die Kommunikation sowohl zwi-

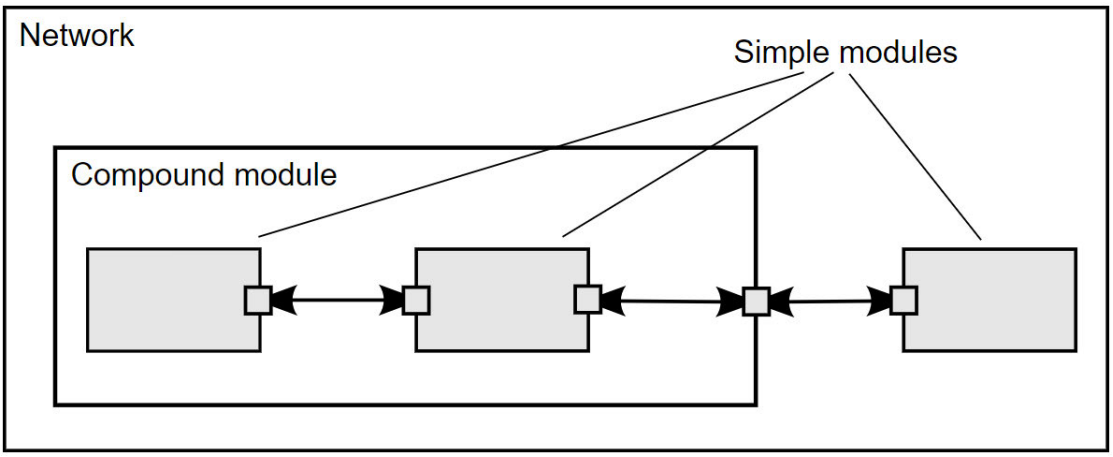

Abbildung 3.1: Die Aufstellung von Simple/Compound Modulen aus dem Omnet++ Benutzer- handbuch[62]

schen einzelnen *compount modules* als auch innerhalb der *compount modules* zwischen den enthaltenen Modulen und dem umschließenden Modul.[62]

Das zuvor erwähnte kleinste Module (Simple Module) beinhaltet in der Regel Algorithmen. Diese werden von Entwicklern in der C++ Programmiersprache mit der Omnet++ Simulations-Bibliothek geschrieben. Für die Kommunikation zwischen den Modulen werden Gates verwendet um Nachrichten von einem Modul zu einem anderen Modul zu übertragen. Dabei können Module beliebig Input- und Output-Gates besitzen. Eine Verbindung zwischen zwei Modulen wird als Link bezeichnet und beschreibt den Kanal zum Übertragen von Nachrichten zwischen diesen. Mittels hierarchischen Aufbaus von Modulen können Nachrichten durch eine Reihe von Verbindungen bis zu dem Bestimmungsort übertragen werden. Die Links, demnach eine direkte Verbindung zwischen Modulen, haben die Möglichkeit mit bestimmten Parametern belegt zu werden um somit unterschiedliches Verhalten zu erzeugen. Dabei können diese Links einen physikalischen Link simulieren und durch Parameter wie *Datenrate, Nachrichtenverlust, Nachrichtenfehler, Übertragungsverzögerung* oder *Deaktivierung* gesteuert werden. Somit können Nachrichten einzelne Informationen zwischen Modulen abbilden und zugleich die Verbindung innerhalb der Simulation zwischen einem Klienten und einem Server bei der Ausführung des Protokolls beschreiben.[62]

Objekte, wie zum Beispiel Nachrichten, Module und Queues werden durch C++ Klassen abgebildet und sind für eine Vielzahl von Objekten bereits in der Simulations-Bibliothek vorhanden. Um ein Model mit Omnet++ aufzubauen, werden mehrere Teilstücke be-

nötigt, die letztendlich als vollständige Simulation ausgeführt werden können. Die Simulationsumgebung in Omnet++ liefert die benötigten Funktionen bereits mit um die Module auszuführen. Im ersten Schritt bietet ein Simulations-Kernel die Grundlage für die Ausführung der Simulationen. Der Simulations-Kernel ist der zentrale Knoten an dem die Simulationen berechnet werden. Dieser Kernel ermöglicht das Auführen der erstellten Simulationen, indem die statischen und geteilten Bibliotheken zur Verfügung gestellt werden. Die von dem jeweiligen Entwickler benötigten Ressourcen sind zum einen die Beschreibung der Netzwerktopologie, welche in Form von NED Dateien erstellt wird. Zum anderen wird die Beschreibung eines Simulationsdurchlaufs mit der Konfiguration des Netzwerkes und einer Beschreibung der Übertragung benötigt. Eine solche Konfiguration wird in einer Datei mit der Endung *.ini* definiert und beschreibt die einzelnen Schritte, die während der Simulation durchlaufen werden.

Damit eine Simulation überhaupt ausgeführt werden kann, sind die relevanten Ressourcen für eine Simulation gebündelt zu betrachten. Das Zusammenbauen der Simulation erfordert in der Regel das Vorhandensein von sämtlichen projektbezogenen Ressourcen der C++ Anwendung inklusive aller Hilfsdateien (Header Files). Ebenfalls wird eine NED Datei benötigt, welche die Komponenten und benötigte Netzwerktopologie beschreibt und definiert. Eine solche NED Datei kann bereits für mehrere Simulationen eingesetzt werden. Zusätzlich zur NED Datei wird für jede Simulation eine *.ini* Datei benötigt, da diese jegliche Einstellungen für die laufenden Anwendung in Form der Parameter und Verknüpfungen enthält. In den meisten Simulationen wird zusätzlich eine weitere Ressource benötigt, welche die in einer Simulation verwendeten Nachrichten definiert und die zugrunde liegenden C++ Dateien übersetzt.

Sind alle Ressourcen vorhanden, kann die Simulation zusammengebaut werden. Die Abbildung 3.2 aus dem Omnet++ Nutzerhandbuch[62] veranschaulicht den Prozess die Ressourcen für eine Simulation zusammenzustellen. Der Output richtet sich dabei nach dem Hostsystem. Die entstehenden Dateien sind unter Unix Systemen mit den Endungen *.a* - für statische und *.so* für geteilte Bibliotheken zu finden. Für Windows und MacOS Systeme sind diese Dateien an die entsprechenden Richtlinien des Betriebssystem angepasst. Befehle, um die Simulationen zu erstellen sind primär in Omnet++ das *opp_ makemake*, welches aus den vorhandenen Ressourcen sogenannte Makefiles erzeugt. Danach ist ein herkömmliches *make*, welches ein build automation Tool entspricht, ausreichend um aus Programmcode, Bibliotheken und Abhängigkeiten eine ausführbare Datei zu erzeugen. Das *make* Tool[51] benötigt die zuvor erzeugten Makefiles um alle Abhängigkeiten zu finden und zu bündeln. Das Ausführen einer Simulation geschieht anschließend indem

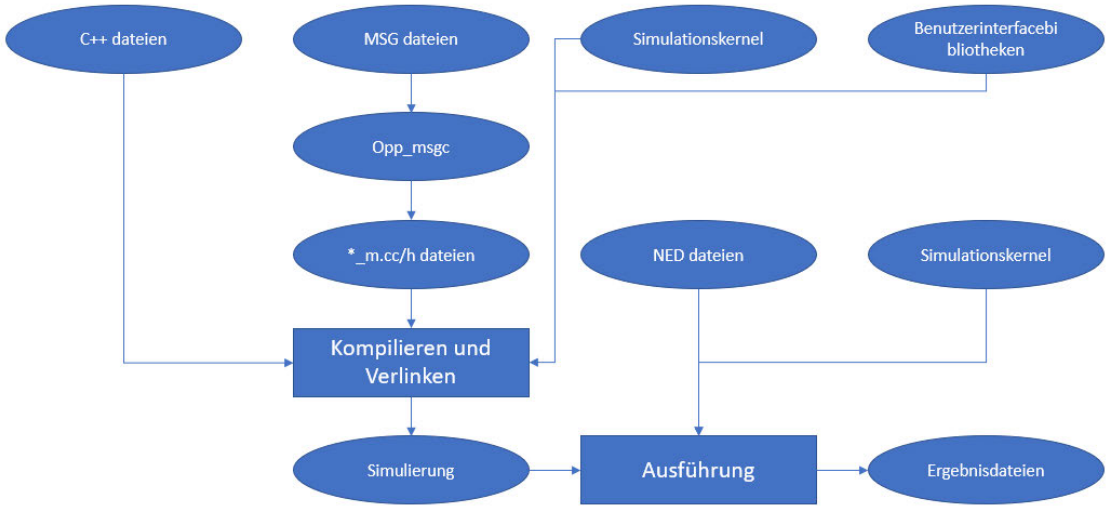

Abbildung 3.2: Der Prozess eine Simulation für Omnet++ zu erzeugen, wie sie im Benutzerhandbuch von Omnet++ beschrieben ist[62]

das Aufrufen der zusammengebauten Datei zum Beispiel mit dem Befehl *./helloWorld* für ein gebautes Hello World Projekt angewendet wird.

Wie zuvor beschrieben, ist Omnet++ eine eventbasierte Simulationsumgebung, die anhand eines Austausches von Nachrichten als Event kommuniziert. Da diese Nachrichten in Omnet++ allerdings nicht ausschließlich die auftretenden Events beschreiben, ist eine Untersuchung notwendig, welche Informationen diese Kommunikation beinhaltet. Eine Nachricht kann sowohl ein Event beschreiben und in Form von Paketen, Befehlen, Aufgaben oder andere Formen von Datenübertragungen. Die Basisfunktionalität von Nachrichten wird durch die C++ Klasse *cMessage* bereitgestellt. Eine Klasse, namens *cPacket*, erbt von der *cMessage* Klasse und wird innerhalb von Simulationen für die Netzwerkkommunikation verwendet. Dem einzelnen Entwickler ist es dabei freigestellt welche der beiden Klassen er ableitet und somit eigene Nachrichtentypen zu erstellen. Eine besondere Form der Nachrichten in einem Modul stellt beispielsweise die sogenannten *self-messages* dar. Diese Nachrichten werden für interne Kommunikation benutzt. Unter anderem werden anhand dessen Funktionalitäten wie Timer oder Interrupts bereitgestellt und verwendet. Die Kommunikation in einer Simulation erfolgt, wie bereits erwähnt, über reinen Nachrichtenaustausch. Diese Nachrichten sind in Omnet++ das zentrale Mittel für die Funktionsweise. Alle weiteren am Transportprotokoll beteiligten Bereiche hingegen sind für die Handhabung und Bearbeitung der Aufgaben aus Nachrichten zuständig. Da die Module an ihren Gates mit Nachrichten kommunizieren, ist es

möglich, dass verschiedene Nachrichten innerhalb einer Simulation auftreten. Die Simple Modules in Simulationen (bei einer abstrakten Betrachtung können die Compound Modules ebenfalls als Simple Modules behandelt werden) besitzen die Fähigkeit Nachrichten zu erzeugen, senden, empfangen, speichern, verändern, planen und vernichten. Dadurch schaffen diese Nachrichten eine Abbildung der Kommunikation, in welcher Weise diese für die Transportprotokolle implementiert werden.

Eine Betrachtung der Visualisierung von einer Simulation und Simulationsumgebung findet in dieser Arbeit keine Anwendung, da dies keine Voraussetzung für die Funktionalitäten eines gesteuerten Testverfahrens darstellt.

3.6 Testen in der Simulationsumgebung Omnet++/INET

Innerhalb einer Simulationsumgebung ist es hilfreich Software ebenfalls durch Tests zu überprüfen. Um die Software in einer Simulation testen zu können, sind zusätzliche Schritte notwendig. Es müssen alle benötigten Ressourcen gebündelt werden und die Simulation zusammengebaut werden.

Es müssen Eintrage definiert werden welche die Tests verifizieren. Mögliche Einträge für das Testen einer solchen Simulation können beispielsweise Constrains[30] sein, die bestimmte RegEx[30] verwenden um einzelne Ausführungen von Software zu verifizieren. Das Testen innerhalb einer Simulationsumgebung ist abhängig von den Konstrukten und dem Aufbau der vorhandenen Simulationsumgebung. Dafür ist zunächst eine Untersuchung notwendig inwiefern in Omnet++ Simulationen getestet werden können. Omnet++ beinhaltet ein eigenes Tool, das *opp_test* Tool, welches ein fester Bestandteil von Omnet++ ist und die Funktionalitäten um das Ausführen von Tests für Simulationen erweitert. Anhand dessen werden die Simulationen zunächst ausgeführt und zusammengebaut und im Anschluss durch das Testtool überprüft. Das Testtool wurde ursprünglich für Omnet++ entwickelt, sodass es möglich ist, den Simulationskernel von Omnet++ eigenständig zu testen. Dabei ist ein Vorteil die Funktionsfähigkeit des Testtools ohne weiteres hinzufügen von Quellcodes in die vorhandene Implementierung zu ermöglichen. Einige Testframeworks benötigen eine Ergänzung im Programmcode, damit beispielsweise zu bestimmten Events eine Überprüfung stattfindet. Bei Omnet++ ist bewusst darauf verzichtet worden, da keine zusätzlichen Programmcodes in die Kernelimplementierung injiziert werden. Dies führt zu der Entscheidung, die Simulation durchlaufen zu lassen

und schließlich im Nachhinein mit den erzeugten Dateien die Implementierung zu verifizieren. Dadurch werden die Logdateien der laufenden Simulation auf Auftreten oder Ausbleiben von Events überprüft. In diesen Logdateien stehen erzeugte Ausgaben eines Simulationsdurchgangs. Dies betrifft sowohl sämtliche auftretenden Events als auch Logausgaben, die bereits vom Entwickler in der Implementierung eingebaut wurden. Nach einem Simulationsdurchlauf durchsucht das Testtool alle vorhandenen Dateien nach einem Vorhandensein bestimmter Ereignisse bzw., abhängig von der Implementation, nach dem Ausbleiben bestimmter Ereignisse. Somit können Simulationen nicht zur Laufzeit überprüft werden, sondern werden im Nachhinein anhand der Durchsuchung auf vorhandenen oder ausgebliebenen Ereignissen kontrolliert.

Wie in jeder Simulation, wird für einen Test innerhalb einer Simulationsumgebung ein Netzwerk in einer NED Datei benötigt. Dieses Netzwerk beschreibt die Topologie, die für das aktuelle Testszenario vorhanden sein muss. Es enthält die Aufzeichnung von benötigten Modulen sowie eine Auflistung der vorhandenen Links zwischen den Modulen. Eine weitere notwendige Ressource ist eine *.test* Testdatei. Die Testdateien sind in unterschiedlichen Sektionen unterteilt, die mittels eines *%<schlagwort>:* Konstrukts eingeleitet werden. Dafür finden Schlagwörter wie Beschreibung, Quelldateien oder Testkonstrukte Verwendung. Zusätzlich enthält die Testdatei eine Verlinkung zu einer *.ini* Datei, welche das Szenario der Simulation darstellt und benötigte Parameter setzt. In dieser Sektion wird zudem das Netzwerk angegeben, die mittels NED Datei beschrieben wird. Der letzte Abschnitt zeigt jegliche Schlagwörter, die den eigentlichen Test analysieren. Es werden Schlagwörter wie *%contains: <Ausgabedatei>*, *%contains-regex: <Ausgabedatei>*, *%not-contains: <Ausgabedatei>* und *%not-contains-regex: <Ausgabedatei>* benutzt um das Bestehen eines Tests zu überprüfen. Die Ausgabedatei beschreibt dabei in welcher der erzeugten Logdateien die folgenden Ereignisse vorhanden sein sollen. Anstelle der Ausgabedatei kann ferner der Standardausgabekanal *stdout*, der innerhalb der Kommandozeile angezeigt wird, durchsucht werden.

Für das erfolgreiche Durchlaufen eines Tests ist im letzten Schritt ein sogenanntes Kontrollscript erforderlich. Wie bereits beschrieben wird damit die Simulation zusammenbaut. Zusätzlich zu dem Kontrollscript werden alle angegebenen Tests generiert und ausgeführt. Die wesentlichen Schritte, die das Kontrollscript ausführt, sind im Listing 3.1 als Pseudocode angegeben. Dabei werden zunächst alle Tests durch das Tool *opp_test* generiert. Dafür kopiert das Testtool die gesamten verfügbaren Dateien mit der Endung *.test* in das Arbeitsverzeichnis um sie von dort aus auszuführen. Als nächstes sind dieselben Schritte durchzuführen, wie bereits in Abschnitt 3.5 genannt, um die Simulation

zu bauen und auszuführen. Dazu gehört das Erzeugen der Makefiles sowie das Zusammenbauen der Simulation mit dem Buildtool *make*. Zuletzt kann folglich das Testtool *opp_test* aufgerufen und die Tests ausgeführt werden. Die Ausgabe des Kontrollscripts ist beispielhaft mit einem Demotestfall in Listing 3.2 aufgeführt. An dieser Stelle sind sämtliche mögliche Ergebnisse für die ausgeführten Tests von PASS, FAIL und UNRESOLVED zusammengetragen.

```
1 #! /bin/sh
2 opptest gen alles mit .test oder abbruch
3 erstelle mit opp_makemake alle Makefiles
4 baue die Simulation mit dem Build Tool make
5 opptest run alle verfuegbaren Tests mit .test oder abbruch
```

Listing 3.1: Pseudocode für die benötigten Schritte des Kontrollscripts zum erzeugen der Tests, dem Zusammenbau und dem ausführen der Simulation

```
1  $ ./runtest demo_testszenario.test
2  opp_test: extracting files from *.test files into work...
3  Creating Makefile in omnetpp/test/core/work...
4  demo_testszenario/test.cc
5  Creating executable: out/gcc-debug/work
6  opp_test: running tests using work.exe...
7  *** demo_testszenario.test: PASS
8  ==========================================
9  PASS: 1    FAIL: 0    UNRESOLVED: 0
10
11 Results can be found in work/
```

Listing 3.2: Die Ausgabe eines Demo Testdurchlaufs mit dem Kontrollscript für das durchführen von Tests

Das hier aufgelistete Kontrollscript ist kein Bestandteil von Omnet++, da hierfür notwendig ist, jede Anwendung neu zu schreiben bzw. zu konfigurieren. Es befindet sich dennoch ein Basisscript für das Kontrollscript in den Ressourcen zu Omnet++ unter dem Pfad *omnetpp/test/core/*. Zusätzlich zu diesen Script sind sowohl die Codebasis von Omnet++ in den Systempfad als auch die benötigten NED Dateien in den NED Path einzutragen. Diese Vorgehensweise geschieht in der Regel innerhalb der Konfiguration von Omnet++, wobei unter Umständen eine Erneuerung dieser Konfiguration notwendig ist, sofern sich an einem System die Parameter geändert haben. Ein Beispiel hierfür ist das im Grundprojekt[21] aufgeführte System, bei dem ebenfalls Jenkins eingesetzt wurde. Dieses System hat Verwendung gefunden um auf simple Weise mit Packetdrill

einen Test auszuführen. Da sich hierbei jedoch in dem temporären Arbeitsordner, der automatisch von Jenkins erstellt wird, die Konfiguration verändert, sind die Variablen vor der Testausführung neu zu setzen.

Sind für das Testtool alle notwendigen Abhängigkeiten vorhanden wird mit dem Aufruf *./runtest demo_ testszenario.test* ein spezieller Test und mit *./runtest *.test* alle in dem Ordner vorhandenen Testfälle ausgeführt. Wie der Aufruf *./runtest demo_ *.test* zeigt, ist zudem eine Kombination mit dem Wildcard Operator möglich. Eine Kombination, die alle Tests ausführt, beginnen mit *demo_* . Die folgende Tabelle 3.1 trägt sämtliche benötigten Ressourcen zusammen mit einer Kurzbeschreibung, die für einen vollständigen Testdurchlauf vorhanden sein müssen.

Ressource	**Beschreibung**
Testdatei	Die Testdatei beschreibt eine Datei mit der Endung *.test* die vom Testtool eingelesen werden kann.
Testtool	Das von Omnet++ bereitgestellte Testtool *opp_ test*
Kontrollscript	Das speziell angepasste Kontrollscript da es sich durch den Einsatz anderer Testframeworks signifikant ändern kann.
Testprogramm	Das Testprogramm ist entweder durch das Kontrollscript dynamisch erzeugt oder beschreibt eine Ausführbare Datei für die Auswertung der Tests.
Testordner	Ein spezieller Ordner innerhalb dessen die Tests ausgeführt und alle erzeugten Dateien hinterlegt werden bei der Ausführung einer Simulation. Standardmäßig ist dies *work/<testname>/* vom aktuellen Verzeichnis aus.

Tabelle 3.1: Beschreibung der Ressourcen die für einen Testdurchlauf gegeben sein müssen.

Mit der Variante nur die Ausgabedateien zu untersuchen um die Simulationen zu testen finden unterschiedliche Verfahren beim Testen Anwendung. Möglichkeiten für ein solches Verfahren sind Smoketests[63], Unittests[33], Modultests[63] oder im Speziellen Fingerprinttests[63]. Die Wahl des Testverfahrens ist dabei ausschlaggebend für das erwartete Ergebnis. Für Smoketests beispielsweise reicht eine Simulation aus, die über einen bestimmten Zeitraum ausgeführt wird mithin ohne Fehler zu produzieren. Dies können unter anderem Fehler sein, die zu einem weiteren Problem führen oder die Simulation im Zweifelsfall abstürzen lassen. Mit Unittests besteht die Möglichkeit einzelne Programmsegmente wie Funktionen und Klassen zu überprüfen. Diese Unittests werden in der Regel mit dem Ziel von Code Coverage eingesetzt um somit möglichst alle Fehlerquellen zu finden und beheben. Die Modultests sind vom Aufbau nicht zu vergleichen mit den zuvor

genannten Verfahren, da die Modultests jegliche Funktionsblöcke testen, die bereits vollständige Protokolle oder Anwendungen darstellen können. Dafür wird in der Regel das zu testende Modul in eine Simulation eingebaut und die anderen Kommunikationspartner durch MOCK Objekte ersetzt. MOCK Objekte sind simple Module, die ein Modul in ihrer inneren Funktionalität beschreiben, jedoch lediglich die für das Szenario benötigten Daten ausgeben oder empfangen können.

3.7 QUIC im INET Framework

Das QUIC Transportprotokoll ist, wie bereits grundlegend in Abschnitt 2.4 erläutert, ein Transportprotokoll, das auf Anwendungsebene arbeitet. Innerhalb der Transportschicht der OSI-/TCP-IP-Referenzmodelle verwendet QUIC das UDP Protokoll und lässt QUIC somit zu einem Sonderfall werden, da es als Transportprotokoll nicht direkt auf der Transportebene arbeitet. QUIC ist ein Protokoll, das Probleme aus den bestehenden Transportprotokollen wie unter anderem TCP, UDP versucht zu lösen und dabei Vorteile wie Latenzen, Multiplexing und TLS mit einbindet. Bestehend aus einer ersten Implementierung innerhalb des INET Repositories kann QUIC zusammen mit der Omnet++ Simulationsumgebung eingesetzt werden. Diese Implementierung basiert auf der Draft Version 10 des QUIC Drafts von der IETF. Die Implementierung enthält grundlegende Funktionalitäten für den Einsatz in der Simulationsumgebung. Im Bereich der Entwicklung wurden die nachfolgenden Funktionalitäten bereits implementiert. Zunächst wurde die im Draft beschriebene Long Header für die Implementierung definiert. Mit diesem Header Format kann daher der Stream 0 implementiert werden. Dieser Stream 0 beschreibt die bidirektionale Kommunikation von Klient und Server. Dabei wird diese Verbindung fortwährend vom Klienten initiiert. Der Stream beinhaltet bereits die beiden Frame Typen PADDING und STREAM und sowohl das initiale Paket, welches vom Klienten gesendet wird als auch das dazugehörige Handshake Paket, das der Server als Antwort sendet. Diese beschriebenen Pakete enthalten als erste Lösung lediglich einen pseudo-kryptografischen Handshake.

Eine Version Negotiation wurde nicht in die weitere Entwicklung eingeplant, da diese von QUIC ausschließlich auf dem Draft Version 10 beruht und es vorerst nicht vorgesehen ist, eine weitere zu implementierten. Insgesamt beschreiben die Funktionalitäten zusammengenommen den 1-RTT Handshake für das QUIC Protokoll. Eine Funktionalität von QUIC, das Verwenden von mehreren Streams für eine Verbindung, wurde ebenfalls bei

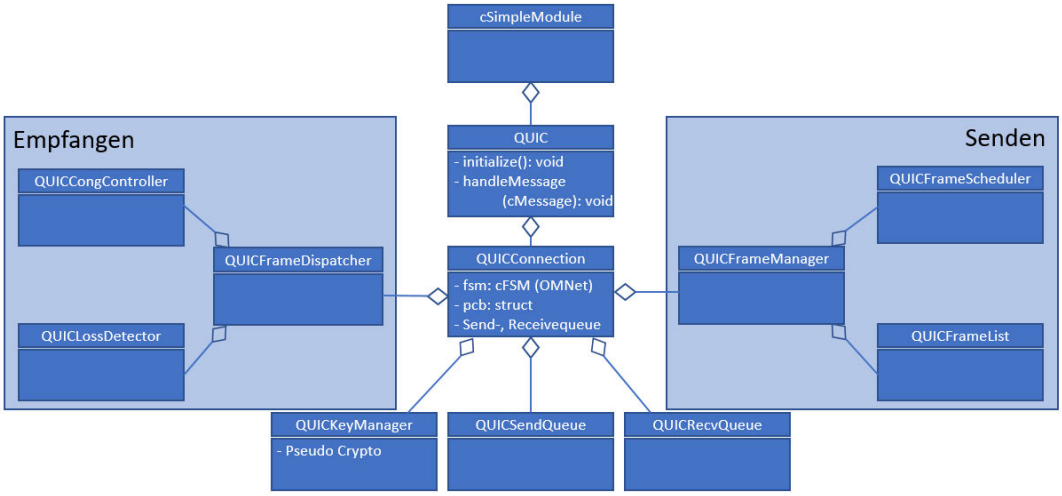

Abbildung 3.3: Das Klassendiagramm wie es für die Implementierung von QUIC in das INET Framework erstellt wurde.[88]

der Implementierung nicht eingebaut. Als Grund hierfür ist zu sagen, dass für eine erste Funktion des Transportprotokolls ein einzelner Stream ausreichend ist. Enthaltende Funktionalitäten sind dabei ein Idle Timeout, das Short Header Paket und Sender/Empfänger Queues. Das Klassendiagramm in Abbildung 3.3 zeigt den aktuellen Aufbau der QUIC Implementierung und in welchem Umfang dieser zum Zeitpunkt dieser Arbeit auf dem Entwicklungsbranch *FEATURE/DENISLUG/Quic-develop* zu finden ist.

Aufgrund der Tatsache, dass das QUIC Transportprotokoll zustandsbasiert ist, wurde die Implementierung mit Hilfe eines Zustandsautomaten erstellt. Dieser Zustandsautomat befindet sich ebenfalls in dem QUIC Draft und ist in Abbildung 3.4 dargestellt. Die Abbildung ist angelehnt an Lugowski[88]. In der Abbildung werden nur die für diese Arbeit relevanten Bereiche dargestellt. Die Abbildung zeigt die Veränderung der Zustände von Klienten und Server, die während einer regulären Verbindung durchlaufen werden. Diese Zustände zeigen die Zustände in denen ein Austausch von Datenpaketen stattfindet, wie dem initialen Handshake Paket oder dem ACK Pakaten. Es werden Änderungen von Zuständen beschrieben, die in der Simulation durch Events (Nachrichten) hervorgerufen werden. Der Klient hat dabei nach der Initialisierung und einem gesendeten initialen Paket den Zustand CLIENT_INI_SENT. Durch das eintreffende ACK Packet des Servers kann der Klient in den Zustand ESTABLISHED wechseln und mit dem Datenaustausch beginnen. Auf der Serverseite wird nach der Initialisierung im Zustand SERVER_LISTEN auf ein eingehendes Paket gewartet. Durch das Eintreffen eines Handshakepakets

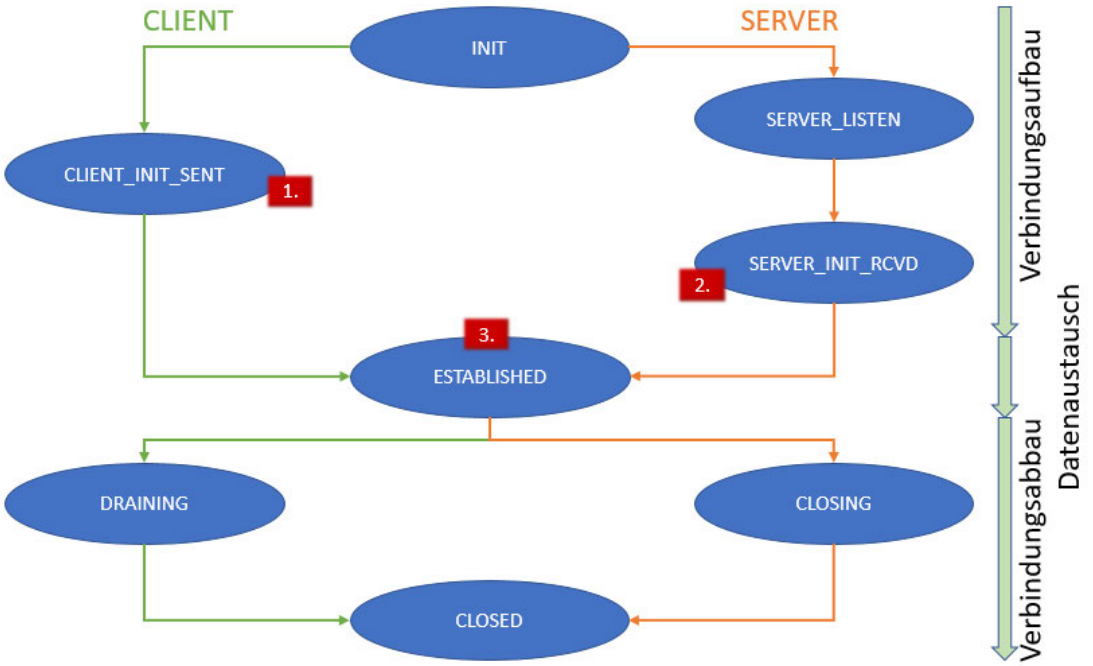

Abbildung 3.4: Der Zustandsautomaten für eine QUIC Verbindung von einem Klienten (initiator) und einem Server angelehnt an der Basis von Denis Lugowski[88]

wechselt der Server in einen Zustand namens SERVER_INIT_RCVD. Akzeptiert der Server die Verbindung und sendet eine Bestätigung in Form eines ACK Paketes zurück, wechselt dieser damit in den Zustand ESTABLISHED und ist folglich bereit für den Datenaustausch.

Das Beenden einer QUIC Verbindung kann durch drei Wege erfolgen. Die erste Variante ist ein Leerlauf der Verbindung über einen bestimmten Zeitraum, sodass ein Timer die Verbindung beendet. Die zweite Möglichkeit ist ein direktes Beenden der Verbindung durch einen der beiden Endpunkte. Dabei wird von einem Endpunkt entweder ein CONNECTION_CLOSE oder ein APPLICATION_CLOSE gesendet, das alle offenen Verbindungen unmittelbar beendet. Die letzte Variante ist ein zustandsloser Reset der Verbindung. Das bedeutet dieser Reset ist reserviert für Server, die keine Möglichkeit besitzen in den Zustand einer aufgebauten Verbindung zu wechseln. Schließlich ist dadurch möglich, dass das Senden von Daten vom Klienten beim Server einen Absturz hervorruft.

Die Implementierung von QUIC ist vollständig in der Programmiersprache C++ und ist in C++-Klassen unterteilt. Diese Klassen bauen auf den bereits in Abschnitt erwähnten 2.5 von Omnet++ bereitgestellten Basisklassen wie Simple Module auf. Die Abbildung 3.3 zeigt anhand eines Klassendiagramms die unterschiedlichen Bereiche, die im Zusammenhang mit der einer Kommunikation des Protokolls stehen. Der zentrale Bereich ist die QUIC Klasse, welche die Basis für das Protokoll darstellt. Darauf aufbauend befindet sich die QUIC Verbindung zusammen mit den Sender- und Empfangs-Puffern. Diese Puffer arbeiten als eine Queue, die im Rahmen der ersten Implementierung nach dem FIFO[2] Prinzip arbeiten. Zusätzlich zu der beschriebenen Funktionalität gibt es einen KeyManager, der einen pseudo Crypto bereitstellt, die die komplette Implementierung für TLS vorerst umgeht. Weiterhin gibt es innerhalb einer QUIC Verbindung zwei vergleichsweise größere Bereiche: Frame Manager und Frame Dispatcher. Ein Frame ist dabei ein Datenpaket, das durch Omnet++ in der Simulation als Nachricht versendet wird. Der Frame Manager ist für sämtliche ausgehenden Frames zuständig, die zum Senden vorbereitet werden. Dazu benötigt dieser sowohl einen Scheduler als auch eine Frame Liste um das Senden der Frames zu koordinieren. Der Dispatcher hingegen hat die Verantwortung für alle empfangenen Frames. Somit behandelt dieser die unterschiedlichen Frames abhängig von ihrem Frame Typ und leitet die Frames an richtige Module weiter. Dieser Aufbau sieht vor, dass jede Klasse ausschließlich für eine Aufgabe zuständig ist.

3.8 QUIC Handshake

Wie bereits in Abschnitt 2.4 erwähnt, befindet sich zum Zeitpunkt dieser Arbeit die Implementierung in der Entwicklung. Dies hat zur Konsequenz, dass nicht alle Funktionalitäten vorhanden sind. Aus diesem Grund ist keine Version Negotiation implementiert. Somit ist auf die Untersuchung der Version Negotiation verzichtet worden. Zudem ist die Implementierung für das TLS im Handshake zu dem jetzigen Zeitpunkt ausschließlich durch die Erstellung eines pseudo Crypo Schlüssels zu realisieren. Daher findet in dieser Arbeit lediglich das Testen des Handshakes mit den benötigten Handshake Paketen Anwendung. Für die QUIC Handshake Pakete werden Long Header verwendet mit dem Wert *0x7F* zur Identifizierung. Dementsprechend verwendet ein Server als Antwort auf das Handshake Pakete in dem Paket den Wert *0x7D*. Der QUIC Draft beschreibt für ein

[2] FIFO steht für *first in first out* und beschreibt dabei eine Queue, bei der neue Daten immer hinten angehängt werden und vorne entnommen werden.

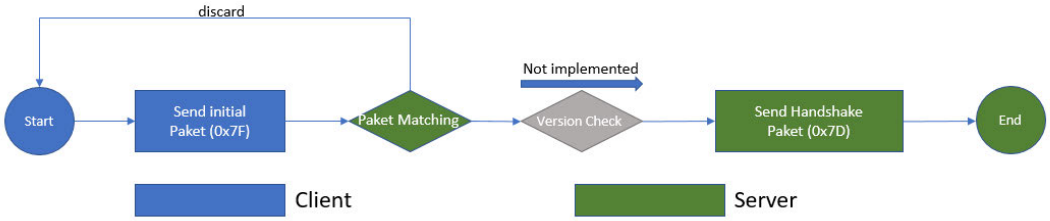

Abbildung 3.5: Der Ablauf eines QUIC Handshakes ohne die Version Negotiation und TLS wie QUIC aktuell im INET Framework Implementiert ist. Aus den Arbeiten von Lugowski[88]

Handshake Paket folgendes Verhalten: Der erste Versuch eines Verbindungsaufbaus geschieht mit einem bislang unbekannten Endpunkt (Server). Die Abbildung 3.5 erklärt den Ablauf eines QUIC Handshakes zwischen einem Klienten und einem Server. Dabei werden die blau unterlegten Bereiche auf der Klientseite und die grün hinterlegten Bereiche auf der Seite des Servers ausgeführt. Die Überprüfung der Version des QUIC Protokolls wird dabei übersprungen da die Version Negotiation wie bereits erwähnt noch nicht im INET Framework implementiert ist.

Zitat aus dem QUIC Draft der Version 10:

> A Handshake packet uses long headers with a type value of 0x7D. It is used to carry acknowledgments and cryptographic handshake messages from the server and client. [62]

Eine Besonderheit von QUIC ist die Möglichkeit anhand des initialen Pakets bereits erste Nutzdaten mitzusenden. Für diese Möglichkeit existiert ein 0-RTT Handshake[39]. Der 0-RTT Handshake findet ausschließlich dann Verwendung, wenn mit einer bekannten Gegenstelle kommuniziert wird. Um die erstmalige Verbindung zu einem Server aufzubauen wird der 1-RTT Handshake zusammen mit dem Long Header eingesetzt. Dieser verhandelt zudem die Version Negotation und den Kryptografischen Schlüssel (TLS) für die Verbindung. Der 0-RTT Handshake wird eingesetzt um erneute Verbindungen zu bereits bekannten Endpunkten erneut aufzubauen. In diesem 0-RTT Handshake Paket können aufgrund der bereits bekannten Gegenstelle erste Nutzdaten mitgesendet werden. Die Abbildung 3.6 zeigt beide Abläufe für den Handshake von einem 1-RTT, dem erstmaligen Aufbau einer Verbindung und dem 0-RTT, bei dem bereits Daten mitgesendet werden können.

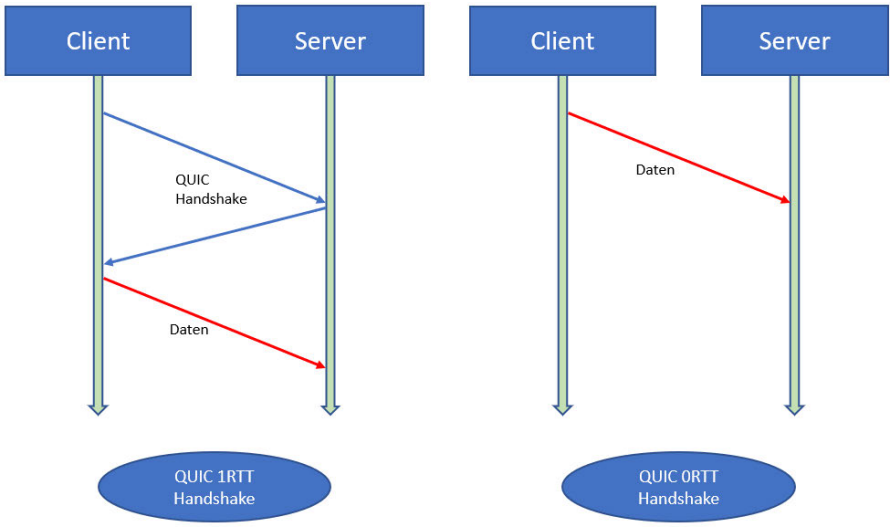

Abbildung 3.6: Die Gegenüberstellung von einem 1-RTT und einem 0-RTT Handshake und die Möglichkeiten wann Daten gesendet werden können.

Aufgrund des Umstandes, dass Omnett++ ohne zusätzliches Einfügen von Programmcodes für das Auswerten von Tests auskommt, können lediglich die durch Implementierung erzeugten Daten ausgewertet werden. Um einen vollständigen Handshake abzubilden ist jedwede Implementierung zu untersuchen, sodass die benötigten Transitionen identifiziert werden. Sobald die notwendigen Klassen für die QUIC Implementierung innerhalb des Klassendiagramms 3.3 analysiert werden, lassen sich die benötigten Schritte für einen Handshake daraus folgern. Bei der Abfolge einzelner Schritte findet zudem der zuvor beschriebene Zustandsautomat in Abbildung 3.4 Anwendung. Anhand dessen können die erzeugten Daten einem Handshake zugeordnet werden. Bei der Auswertung werden die erzeugten Daten der Simulation auf den Ablauf des Handshakes in Abbildung 3.5 referenziert und den einzelnen Schritten zugeordnet, beginnend mit dem Senden des initialen Pakets mit dem Wert *(0x7F)* im Header. Innerhalb des Zustandsautomatens ist diese Vorgehensweise mit Schritt 1. gekennzeichnet. In den von der Simulation generierten Daten lässt sich das Senden dieses initialen Pakets als Ausgabe in folgendem Format wiederfinden: *quic_basic: QUIC_S_PCB_INIT -> QUIC_S_CLIENT_INIT_SENT.* Diese Transition lässt sich weiterhin in dem Zustandsautomaten durch das CLIENT_-INI_SENT validieren. Im nächsten Schritt antwortet der Server auf diese Nachricht mit einem Paket und dem Wert *(0x7D)* im Header und schafft damit die Grundlage für den Verbindungsaufbau. Dies ist der zweite Schritt, der für den Ablauf des Handshakes im Zustandsautomaten markiert ist. Auf der Seite des Klienten ist, wie im Zustandsautomaten

beschrieben, die anschließende Transition aus dem Zustand CLIENT_INI_SENT in den ESTABLISHED. Dieser Vorgang ist mit Schritt 3. gekennzeichnet. Diese Transition lässt sich ebenfalls in den erzeugten Daten der Simulation durch folgende Ausgabe wiederfinden: *quic_basic: QUIC_S_CLIENT_INIT_SENT –> QUIC_S_ESTABLISHED*. Mit der Transition von beiden Seiten in den Zustand ESTABLISHED ist der Handshake vollständig, sodass der Datenaustausch letztendlich beginnen kann.

Der zuvor aufgeführte Handshake eignet sich deshalb optimal als ein Demotestfall, da hier die Funktionalität bereits vollständig für eine Kommunikation zwischen einem Klienten und einem Server ist. Eine QUIC Verbindung benötigt stets einen Handshake um eine Funktionsfähigkeit sicherzustellen. Daher ist der Handshake der Startpunkt für ein Transportprotokoll und zugleich die zuerst verwendete Funktionalität im Rahmen der Implementierung. Ohne einen funktionierenden Handshake ist ein Verbindungsaufbau unmöglich, mit der Folge, dass keine anderen Protokolle auf den Referenzmodellen arbeiten. Zusätzlich wird durch Veränderungen an der Implementierung weiterhin sichergestellt, dass der Verbindungsaufbau mit dem Protokoll funktioniert und nicht zerstört wird.

3.9 Gesteuerte Testverfahren im Forschungskontext

Das Testen bildet sowohl ein zentrales Mittel in der Praxis als auch in der Forschung. Abhängig von Ziel und Umfang des Testens benötigt dies eine gewisse Fachkompetenz der zugrundeliegenden Technologie. Somit können Tests in der Praxis eingesetzt werden um bestimmte Grundlagen oder Attribute zu überprüfen. In der Forschung sollen Tests nicht nur zur Überprüfung, sondern für eine Belegung bzw. Widerlegung von aufgestellten Hypothesen eingesetzt werden. Dazu ist der Testbereich exakt zu spezifizieren, zumal in der Forschung regelmäßig unter bestimmten Vorbedingungen und Voraussetzungen ein konkretes Ergebnis erwartet wird. In diesem Sinne belegt der Forschungsbereich anhand von Tests Argumente anhand derer aufgestellte Schlussfolgerungen getroffen werden. Ebenfalls können im gleichen Verfahren auf Argumente widerlegt werden. Das Testen von komplexen Systemen bedarf einer Methodik, welche den Umfang und die Spezifikation der verwendeten Testverfahren einschließt.

Durch eine sinnvoll gewählte Methodik können weitere Aussagen über das System getroffen werden. Dafür wird zwischen statischen und dynamischen Testmethoden unterschieden. Statische Testmethoden beschäftigen sich damit bestimmte Artefakte in der

Implementierung zu finden ohne diese auszuführen. Bei den dynamischen Methoden hingegen wird nach dem Auftreten von Fehlern in einem laufenden und zusammengesetzten System gesucht.

3.10 Strukturierung & Aufbau

Die Struktur dieses Konzepts für ein gesteuertes Testverfahren wird durch Rahmenbedingungen beeinflusst. Diese erfordern nicht nur den Einsatz von besonderer Methodik sondern helfen gleichzeitig die Systemgrenzen exakt zu definieren und aufzuzeigen. Im ersten Schritt gibt die Wahl der Testmethodik indirekt die Simulationsumgebung Omnet++ zusammen mit dem *opp_test* Tool vor. Indirekt bedeutet hierbei die Verwendung des Testtools von Omnet++, da insofern ebenfalls andere Testframeworks eingesetzt werden können, diese jedoch anschließend zusätzlichen Arbeitsaufwand erfordern im Vergleich zu dem bereits integrierten Testtool. Dieses Testtool beschreibt in welcher Weise die Testszenarien aufgebaut, ausgeführt und überprüft werden können. Welche Schritte dieses Testtool im Detail durchläuft wurde bereits in Abschnitt 3.6 erläutert.

Eine weitere Basis für die Erstellung eines gesteuerten Testverfahrens stellt der Server dar, welcher hinter einer Firewall in der Hochschule steht. In diesem Fall ist eine weitere Technologie erforderlich, um die Webhooks durch die Firewall an den Server hindurch zu bringen. Webhooks bedeuten in diesem Zusammenhang Benachrichtigungen über eine Änderungen an der Codebasis. Da dieses Vorgehen ein häufig verwendetes Szenario darstellt, bietet GitHub eine solche Technologie an. Auf Github wird zudem das INET Repository gehostet. Diese Software wird anhand der Website Smee.io[26] zur Verfügung gestellt, wohinter sich ein Publish Subscribe Modell verbirgt. Auf dieser Technologie können sich Klienten mit einer vorher generierten URL verbinden und erhalten im Anschluss sämtliche Nachrichten, die auf diese URL geleitet werden.

Das Ergebnis dieser Analyse und die geschilderten Grundlagen dienen insgesamt dazu ein Konzept hervorzubringen, welches auf die Problemstellung, die zuvor in Abschnitt 3.1 erläutert wurde, zugeschnitten ist. Dieses Ergebnis umfasst die Erstellung eines gesteuerten Testverfahrens mithilfe dessen die Ressourcen und die Testmethodik effizienter eingesetzt werden können.

4 Konzept und Realisierung

Aufbauend auf den Grundlagen und der Analyse wird im Folgenden aus den vorgestellten Technologien das Konzept entwickelt. Somit setzt sich dieses Kapitel grundsätzlich mit Technologien auseinander, welche im Rahmen des gesteuerten Testsystems eingesetzt und in den Abschnitten 2 und 3 beschrieben wurden. Eine Darstellung des Konzepts gibt Aufschluss über die Realisierung eines solchen gesteuerten Testverfahrens. Zunächst werden Entscheidungen und Annahmen diskutiert, die getroffen wurden, um das System zu realisieren. Als Annahmen zählt sowohl eine eingesetzte Version als auch eine notwendige Technologie um das gesteuerte Testverfahren zu realisieren.

Darauf folgt eine Beschreibung der Testumgebung, welche sowohl den Aufbau als auch den Einsatz von Tests innerhalb des Testverfahrens darlegt. Diese Testumgebung ist ein Bestandteil der Simulationsumgebung und wird mithin zum Ausführen der Simulationen für die Testfälle benötigt. Weiterhin wird die Omnet++ Simulationsumgebung zusammen mit dem exakten Workflow in das Testsystem eingegliedert. Anhand dieser Eingliederung werden die Tests als ein Teil des gesteuerten Testverfahrens ausgeführt. Im nächsten Schritt folgt ein Einblick in das Testsystem mit Hinblick auf die Unterstützung im Entwicklungsprozess. Dabei werden bereits Aspekte vorgestellt, welche für die Bestätigung eines effektiveren Einsatz von Ressourcen herangezogen werden können. Dabei wird sowohl der resultierende Nutzen als auch der zusätzliche Aufwand, der mit einem solchen Testverfahren einhergeht, beschrieben. Abschließend fasst eine Auflistung der erwarteten Ergebnisse und eine darauf aufbauende Interpretation und Zuordnung den Abschnitt zusammen.

4.1 Gesteuertes Testverfahren

Das Ziel eines gesteuerten Testverfahren ist die Unterstützung der Entwicklung und das Reproduzieren von Testdurchläufen in einem möglichst kurzen Entwicklungszyklus. Ab-

hängig von der Beschaffenheit eines Systems muss das Testverfahren hinreichende Merkmale aufweisen, um möglichst für die Entwicklung unterstützend zu funktionieren. Für diese Arbeit wird dafür der aktuell verfügbare Stand des INET Repositories[68][1] als Grundlage verwendet. Der aktuelle Stand des QUIC Transportprotokolls ist aus dem QUIC Draft der Version 10 und beinhaltet lediglich die in Abschnitt 3 beschriebenen Funktionalitäten und Erweiterungen. Das Testsystem ist folglich auf das QUIC Protokoll in der Simulationsumgebung Omnet++ abzustimmen.

Zunächst sind es äußere Faktoren, wie ein Jenkins Server[40], ein Webhook[17] oder ein Docker Server[55], die die eingesetzten Technologien bereitstellen. Der Jenkins Server ist eine Open Source Lösung für einen Automatisierungsserver. Der Docker Server beschreibt in diesem Zusammenhang einen Hardware Server, auf dem die Docker Engine arbeitet. Die Docker Engine erlaubt das Starten von Docker Containern wie beispielsweise dem Jenkins Server. Da Jenkins als Automatisierungsserver lediglich ein Startsignal für definierte Arbeitsfolgen benötigt, wird innerhalb dieser Arbeit dieser Schritt mittels Webhook durchgeführt. Der Webhook ist, wie bereits in Kapitel 2 beschrieben, eine Technologie um Änderungen an einer Codebasis zu verfolgen. Dabei wird von GitHub ein solcher Webhook generiert, sobald eine Änderung mit dem Repository zusammengeführt wird. Diese Technologie ist für ein funktionierendes Testverfahren notwendig, da anderenfalls kein definierter Startpunkt für einen erneuten Testdurchlauf existiert. Um jederzeit auf einen Webhook reagieren zu können, sollte die Software für das Testsystem auf einem Server laufen, der eine möglichst hohe Verfügbarkeit ausweist. Ausschließlich unter diesen Voraussetzungen kann sichergestellt werden, dass zu jeder Zeit Änderungen im Repository anhand eines Testlaufs überprüft werden.

Als weiterer Bereich ist die Software für das QUIC Transportprotokoll zu nennen. Diese liegt im INET Repository[68] und ist für die Simulationsumgebung Omnet++ verfügbar. Der größte Vorteil, den Omnet++ gegenüber NS2/NS3 hat, ist die deutlich fundierte und übersichtliche Visualisierung. Da die Tests lediglich in einem Docker Container in der Jenkins Pipeline ausgeführt werden, ist dieses größte Argument für Omnet++, die Visualisierung, in diesem Fall nicht ausschlaggebend. Vielmehr spricht für Omnet++, dass eine Vielzahl von vordefinierte Modulen in einzelnen Frameworks gekapselt sind, wie zum Beispiel das verwendete INET Framework. Die primäre Entwicklungssprache für Omnet++/INET ist in diesem Fall C++ sowie die eigene NED Sprache. Somit besteht die

[1] Es wird der vom CaDS verwendete Fork des INET Repository herangezogen. Auf diesem Stand befindet sich die aktuelle Version der QUIC Implementierung. Dabei wird der Branch FEATURE/DENISLUG/Quic-develop[88] mit der letzten Änderung von November 28. 2018

Möglichkeit mit einfachen Mitteln effektivere Netzwerke bzw. Subnetzwerke aufzustellen und zu simulieren. Das QUIC Transportprotokoll ist bereits Teil der im INET Framework verfügbaren Funktionalität.s.

Die Bereitstellung eines gesteuerten Testverfahrens ist im kompletten Umfang mit keiner aktuell verfügbare Software realisierbar. Aus diesem Grund muss ein solches Testverfahren aus unterschiedlichen Technologien zusammengebracht werden. Dabei erhält jede Funktionalität ihre Aufgaben um einen vollständigen Durchlauf zu ermöglichen. Für die einzelnen Bereiche des Testverfahrens ist erforderlich, dass Änderungen vom System erkannt werden und entsprechend reagiert wird. Die Abbildung 4.1 beschreibt alle notwendigen Komponenten, die für das Konzept benötigt werden und zusammen das Testverfahren darstellen. Dort wird ebenfalls gezeigt auf welchen Wegen bei diesen Komponenten eine Kommunikation erfolgt. Zusätzlich zeigt diese Abbildung die zwei Bereiche von Docker Swarm[75] und Extern. Docker Swarm beschreibt an dieser Stelle die Softwarelösung der Containerisierung auf dem Server, auf welchem das gesteuerte Testverfahren erstellt wird. Extern beschreibt Dienste, die außerhalb des Docker Swarm Servers liegen, jedoch für das Testverfahren benötigt werden. Diese beiden Technologien sind ein GitHub Repository und ein Smee Server[26]. Innerhalb der Docker Swarm Umgebung werden drei Komponenten erstellt. Die erste und größte Komponente ist der Automatisierungsserver Jenkins. Dieser erhält die Webhooks aus dem Repository, die durch den lokalen Smee Client weitergegeben werden. Als dritte Komponente existiert ein Docker Image für die Omnet++ Umgebung, die von Jenkins innerhalb einer Pipeline gestartet werden kann. Eine detaillierte Beschreibung des vollständigen Durchlaufs eines Testzyklus wird im Abschnitt 4.7 zusammen mit den Einzelheiten der Kommunikation gezeigt.

Die Abbildung 4.2 zeigt die Architektur der Software. Das bedeutet in welcher Form das Testsystem aufgebaut und zusammengesetzt wird. Zu sehen sind hierfür in der Abbildung die Docker Swarm Umgebung zusammen mit der Docker Engine, welche innerhalb von Docker Swarm läuft. Innerhalb der Docker Engine werden die drei Komponenten als einzelne Blöcke dargestellt. Dabei beschreibt jeweils einer der Blöcke einen Docker Container. Die Container heißen Smee Client, Jenkins und den in der Pipeline benötigten Container mit der Omnet++/INET Umgebung wird das gesteuerte Testverfahren realisiert.

Die Erstellung und die Konfiguration der einzelnen Docker Images wird in den folgenden Abschnitten erläutert. Aus den zusammengebauten Docker Images können anschließend

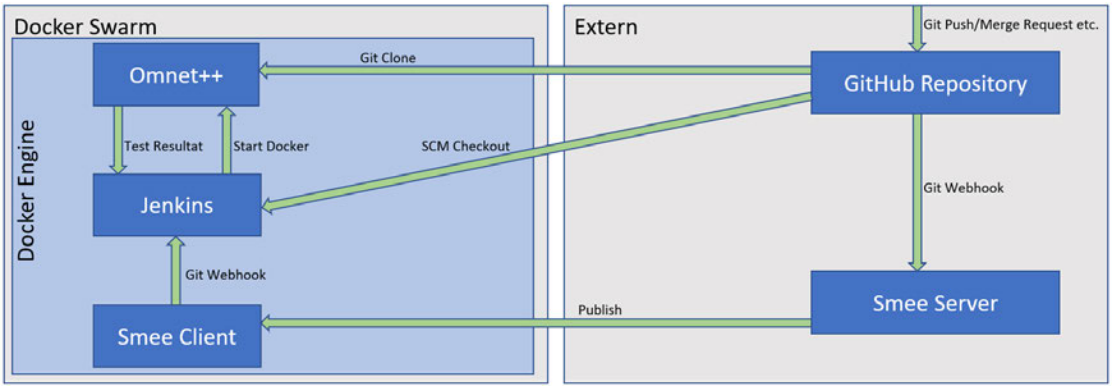

Abbildung 4.1: Die benötigten Komponenten für das Testsystem und wie sie in Abhängigkeit zur Ablauf stehen.

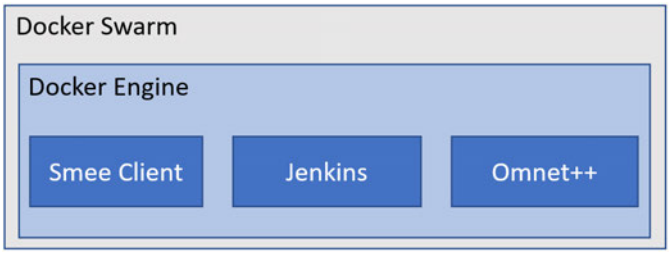

Abbildung 4.2: Diese Abbildung beschreibt die Architektur wie sie auf dem Server eingerichtet wird für das Testverfahren

die Docker Container gestartet werden, welche die für das Testverfahren benötigte Funktionalität ausführen. Alle erstellten Ressourcen und Scripte für das Testverfahren sind in dem GitHub Repository CADS_jenkins_tools[20] gespeichert. Von hier aus hat der Jenkins Server Zugriff auf die Konfiguration für die Pipeline und Testressourcen. Dafür stehen die für den Demotestfall benötigte NED-Datei und die einzelnen Testdateien ebenfalls innerhalb dieses Repositories zur Verfügung.

4.2 Machbarkeitsuntersuchung mit dem QUIC Handshake

Im Rahmen einer Machbarkeitsuntersuchung werden Projekte bzw. komplette Projektpläne auf ihre technische Umsetzbarkeit überprüft. Dabei wird der mögliche Lösungsansatz für ein gesteuertes Testverfahren untersucht. Anhand dieser Untersuchung werden die Erfolgsaussichten abgeschätzt und mögliche Risiken identifiziert. Die Machbarkeitsuntersuchung liefert Projektergebnisse, welche unter dem betrachteten Lösungsansatz und

den vorgegebenen Rahmenbedingungen, erzielt werden können.

Die Untersuchung eines Arbeitsablaufs zielt darauf ab ob ein zeitlicher Gewinn durch das kontinuierliche Feedback erzielt werden kann unter dem Einsatz eines gesteuerten Testverfahrens. Zusätzlich zu dem zeitlichen Gewinn ist eine Verbesserung in der Qualität der Implementierung erstrebenswert. Zusätzlich soll ein effizienterer Einsatz für sowohl Ressourcen als auch das Testverfahren erreicht werden. Für einen effektiven Einsatz läuft dieses Testverfahren im Hintergrund und dient ausschließlich als Rückmeldung und Informationsquelle. Es ist kein Starten des Prozesses seitens des Entwicklers erforderlich. Abgesehen von der initialen Konfiguration des Testverfahrens, wird dieses Continuous Testing System bei allen Änderungen im Repository verwendet. Dieses System erkennt aktuelle Änderungen an der Codebasis und integriert demnach selbständig die Software, in der im weiteren Verlauf alle Tests ausführt werden. Für die vollständige Präsentation eines Durchlaufs des gesteuerten Testverfahrens wird ein Demotestfall benötigt um die Schritte zu beschreiben. Der erste Testfall demonstriert einen vollständigen Arbeitsdurchlauf des Testverfahrens und untersucht damit einhergehend die Unterschiede im Arbeitsprozess von Entwicklern. Die Software, die als Basis für das gesteuerte Testverfahren verwendet wird, ist das Transportprotokoll QUIC. QUIC ist in der Omnet++ Simulationsumgebung mittels INET Framework verfügbar. Da sich dieses Transportprotokoll sowohl bei der IETF im Status der Standardisierung als auch in der Entwicklungsphase für INET befindet ist auszuschließen, dass jede Funktionalität getestet werden kann. Zum Zeitpunkt der Erstellung dieser Arbeit existieren in der Implementierung ausschließlich die Funktionalitäten für einen Handshake. Die Version Negotioation wurde vorerst nicht implementiert, da lediglich eine Version des QUIC Transportprotokolls im INET Framework implementiert ist. Als erster Testfall wird ein initialer Handshake zwischen einem Klienten und einem Server verwendet. Der Ablauf und die genauen Schritte des QUIC Handshakes wurden bereits in Abschnitt 3.8 dargestellt. Die Rückmeldung über den Erfolg oder Fehlschlag dieses Testfalls kann unmittelbar in der Übersicht des Automatisierungsservers entnommen werden.

Die Erfolgsaussichten beim Einsatz dieses Testverfahrens sind relativ hoch, da der Zeitgewinn eines Entwicklers geschaffen werden kann. Dazu wird benötigte Zeit zum Ausführen der Tests auf das gesteuerte Testverfahren ausgelagert. Mögliche Risiken für den Einsatz eines solchen gesteuerten Testverfahrens sind ein steigender Konfigurationsaufwand in Bezug auf die verwendeten Technologien bei wachsenden Projekten.

4.3 Entscheidungen & Annahmen

Der nachfolgende Abschnitt diskutiert die Entscheidungen und Annahmen, welche im Zusammenhang mit der Wahl und der benötigten Funktionalität zu treffen sind. Ziel dabei ist ein Testverfahren zu erstellen, dass ein effizienten Einsatz der Ressourcen ermöglicht. Dieses Ziel wird grundsätzlich dadurch erreicht, indem Arbeiten automatisiert ausgeführt werden können ohne das ein Entwickler eingreifen muss.

Die Simulationsumgebung Omnet++ unterliegt einer andauernden Weiterentwicklung und Erweiterung in ihrer Funktionalität. Da die Implementierung für QUIC in dem INET Repository für den QUIC Draft in der Version 10 festgesetzt wurde, können nicht alle Versionen der Omnet++ Simulationsumgebung für dieses Testverfahren eingesetzt werden. Das Problem dabei ist, dass sich Datentypen von einer zur nächsten Version innerhalb von Omnet++ geändert haben, sodass die aktuelle Implementierung von QUIC überarbeitet werden muss. Auf dieses Problem wird nicht weiter eingegangen, da eine Überarbeitung der QUIC Implementierung nicht Bestandteil dieser Arbeit ist. Ein Update für diese Probleme ist innerhalb des INET Repositorys erforderlich. Zum Zeitpunkt dieser Arbeit ist allerdings kein Update erfolgt. Folglich nutzt diese Thesis die Omnet++ Simulationsumgebung in der Version 5.4.2. Dies ist in diesem Fall die aktuelle Version mit der Möglichkeit das INET Repository zu kompilieren und das QUIC Transportprotokoll zu verwenden.

Als Basis für die Automatisierung des Testverfahrens wird ein Server benötigt, der es ermöglicht automatisierte Arbeitsfolgen zu beschreiben. Jenkins ermöglicht das Ausführen von Programmcodes in einer Pipeline. Innerhalb dieser Pipeline können somit unter anderem die Schritte CI, CT, CD erfolgen. Dazu gehören Versionierung von Releases, Uploads und Downloads von Daten auf Servern und Funktionen wie beispielsweise das Sichern von Artefakten auf einem Artifactory Server[78] innerhalb des Firmennetzwerk.

Jenkins wurde als Hauptbestandteil dieser Arbeit ausgewählt, da es bereits eine breite Anwendung im Fachgebiet der Informatik erlangt hat. Weiterhin bietet es Flexibilität, um alle denkbaren Szenarien abbilden zu können. Eine erweiterte Funktionalität wird dabei durch Plugins erreicht, welche direkt in Jenkins heruntergeladen, installiert und konfiguriert werden können. Die Tabelle 4.1 zeigt die verwendeten Plugins zusammen mit einer kurzen Beschreibung für die verwendete Funktion. Es ist deutlich zu machen, dass nicht für jede Pipeline zugleich alle Plugins verwendet werden. Jenkins ist im Vergleich zu anderen Technologien, die für CI, CD und CT eingesetzt werden, überaus flexibel und

bietet sowohl unterschiedliche Basiskonstrukte als auch Konfigurationen für eine Pipeline an.

Plugin Name	**Beschreibung**
Build Name and Description Setter	Wird verwendet um anstatt einfacher fortlaufender Zahlen den Pipeline Durchläufen einen konkreten Namen zuzuweisen. Dadurch lässt sich in der Übersicht bereits am Namen erkennen zu welcher Pipeline ein Prozess gehört.
Docker Pipeline	Dieses Plugin beinhaltet alle Funktionalität um innerhalb von Pipelines Images zu bauen, pushen oder auszuführen. Zusätzlich bietet dieses Plugin einen Wrapper der es erlaubt Pipeline Befehle aufzurufen, welche dann in einem Container ausgeführt werden.
Git plugin	Jenkins wird um die Funktionalität von Git erweitert. Dieses Plugin beinhaltet die Grundlegende Implementierung von Git ohne auf eine spezifische Versionierungsplattform zu verweisen.
Git Parameter Plug-In	Mit diesem Plugin ist man in der Lage innerhalb von Pipelines spezielle Branches, Tags, oder Revisions eines Repositories anzugeben und zu verwenden.
GitHub API Plugin	Stellt die GitHub API innerhalb von Jenkins und für andere Plugins zur Verfügung.
GitHub plugin	Durch dieses Plugin wird GitHub in Jenkins integriert. Dieses Plugin benötigt das GitHub API Plugin
SSH Credentials Plugin	Dieses Plugin wird verwendet um Zugangsdaten innerhalb von Jenkins zu verwalten und innerhalb von Pipelines zur Verfügung zu stellen. Die Pipeline nutzt dieses zum Beispiel für die GitHub Zugangsdaten um Repositories zu clonen.
Workspace Cleanup	Löscht nach einem Pipeline Durchlauf den Workspace damit keine Dateien in den Ordnern verbleiben und Festplattenspeicher verbrauchen

Tabelle 4.1: Auflistung der im Jenkins Server verwendeten Plugins.

Die nächste Software ist die bereits ausführlich beschriebene Simulationsumgebung Omnet++. Die Installation von Omnet++ und das Einrichten von INET bedürfen einiger Konfiguration. Um diesen Prozess lediglich einmal anzustoßen und damit im Endeffekt beliebig viele Testdurchläufe durchführen zu können, wurden sämtliche Schritte in ein Docker Image gepackt. Aus diesem Docker Image können beliebig viele Docker Container gestartet werden, welche dieselben Einstellungen und Konfiguration beinhalten. Wie bereits in Kapitel 2 beschrieben ist dieser Fakt einer der großen Vorteile einer Virtuali-

sierung mit einem Containerdienst wie Docker.

Das im Rahmen dieser Arbeit entwickelte gesteuerte Testverfahren befindet sich auf einem Server der Hochschule für Angewandte Wissenschaften Hamburg (HAW). Eine an der HAW arbeitenden Forschungsgruppe CaDS verwendet diesen Server und betreibt somit bereits ein Docker Swarm Server[2]. Für diesen Server kann Jenkins und der Smee Client unmittelbar für die Docker Engine entwickelt werden. Die einzelnen Dockerfiles befinden sich in den Anlage dieser Arbeit und werden an den benötigten Stellen ausführlich erläutert. Für diese Arbeit wurden drei verschiedene Docker Images erstellt: Jenkins, Smee Client und die Omnet++ Simulationsumgebung.

Zur Untersuchung eines effektiven Einsatzes von Ressourcen wird im letzten Schritt ein Arbeitsprozess eines Entwicklers definiert, anhand dessen ein zeitlicher Arbeitsaufwand untersucht werden kann. An einem Arbeitstag implementiert der Entwickler zwei unterschiedliche Erweiterungen für eine Software. In diesen Erweiterungen verbirgt sich jedoch ein Fehler, welcher die Tests der bestehenden Implementierung fehlschlagen lässt. Aus diesem Anlass sucht der jeweilige Entwickler nach dem Fehler in seinen Änderungen an der Codebasis. Sofern der Fehler identifiziert ist, implementiert er einen Hotfix für dieses Problem, sodass die Software anschließend einwandfrei funktioniert. Mit einem Hotfix laufen folglich alle vorhandenen Testfälle wieder fehlerfrei durch. Dieses Szenario beschreibt einen repräsentativen 8 Stunden Arbeitstag eines Entwicklers. Dabei soll mit diesem Szenario ein Vergleich zu dem Einsatz des gesteuerten Testverfahren ermöglicht werden. Ebenfalls kann durch den Einsatz des gesteuerten Testverfahrens untersucht werden in wie weit der effektivere Einsatz von Ressourcen erreicht wird.

4.4 Die Umsetzung der Testumgebung

Die Umsetzung des hier beschriebenen Konzeptes erfolgt auf Basis des in Abschnitt 3.8 beschriebenen QUIC Handshake. Dieser Handshake fungiert als Demotestfall um die Ausführungen der Simulationen mit dem *opp_ test* Tool in einem gesteuerten Testsystem ausführen zu können. Diese können im Anschluss untersucht und ausgewertet werden. Der QUIC Handshake beschäftigt sich mit den ersten Nachrichten, die zwischen einem Klienten und einem Server ausgetauscht werden und bilden damit die Grundlage für

[2]Ein Docker Swarm Server ist eine Erweiterung der Docker Engine für verteilte Systeme. Aus diesem Grund können direkte Cluster definiert werden, welche nicht auf derselben Hardware laufen müssen. Diese Technologie wird für viele größere bzw. über den Globus verteilte Anwendungen eingesetzt.

einen Verbindungsaufbau. Da QUIC, als bezeichneter Sonderfall, ebenfalls TLS und die Version Negotiation im Verbindungsaufbau direkt mit aushandelt, können diese zudem als mögliche Testfälle herangezogen werden. Zum Zeitpunkt dieser Arbeit ist die Version Negotiation und das TSL nicht vollständig für die Simulationsumgebung in Omnet/INET implementiert, sodass diese beim Handshake Demotestfall keine weitere Betrachtung finden.

Wie im Abschnitt 3.5 erläutert, werden in Omnett++ Simulationen durch *.ned*-Dateien und *.ini*-Dateien beschrieben. In den ini-Dateien werden die für Omnet++ benötigten Konfigurationen einer Simulation vorgenommen. In den ned-Files wird das Netzwerk aufgebaut und definiert. Dies geschieht Anhand von einzelnen Parametern, wie die verwendeten Endpunkte (hosts) und Knotenpunkte (router). Das Netzwerk, welches für den QUIC Handshake verwendet wird, ist im Listing 4.1 wiederzufinden.

```
1  network QuicTest
2  {
3      types:
4          channel pppdrc extends DatarateChannel
5          {
6              delay = 0.01ms;
7              datarate = 100Mbps;
8          }
9
10     submodules:
11         host1: NetPerfMeterHost {
12             gates:
13                 pppg[1];
14         }
15         host2: NetPerfMeterHost {
16             gates:
17                 pppg[1];
18         }
19         router1: Router {
20             gates:
21                 pppg[2];
22         }
23         router2: Router {
24             gates:
25                 pppg[2];
26         }
27
28     connections:
29         host1.pppg[0] <--> pppdrc <--> router1.pppg[0];
30         router1.pppg[1] <--> pppdrc <--> router2.pppg[0];
31         router2.pppg[1] <--> pppdrc <--> host2.pppg[0];
32 }
```

Listing 4.1: QUIC Network definition für die verwendeten Simulationen

Anhand des Listings 4.1 können die Definitionen, die in Omnet++ verwendetet werden, identifiziert werden. Diese Definitionen beschreiben einzelne Module, die die Funktionalität bereit stellen. Eine genaue Erläuterung dazu ist in Abschnitt 3.5 zu finden. Beide Router sowie Hosts bilden jeweils ein Compound Module, bei denen die Ein- und Ausgänge definiert sind (Gates). Zusätzlich beschreibt das Netzwerk die einzelnen Verbindungen zwischen den Modulen und gibt an in welche Richtungen diese miteinander kommunizieren können.

Das Listing 4.1 definiert das Netzwerk QuicTest. Das Netzwerk besteht aus zwei Compound Modulen für die Endpunkte (*host1* und *host2*) zwei Compound Modulen für Router (*router1* und *router2*). Zuerst wird in Zeile 4 unter der Kategorie *types* ein Channel definiert, der mit $0.01ms$ delay und einer Datenrate von $100Mbps$ initialisiert ist. Dieser Channel bildet die wesentliche Übertragung von Daten auf dem IP-Layer ab und verknüpft somit die einzelnen Module miteinander. Definiert werden die Verbindungen ab der Zeile 28 ab dem Keyword *connections*. In diesem Bereich werden alle Verbindungen zusammen mit den Richtungen der möglichen Signale beschrieben. Aus diesem Grund erreicht der *host1* über den DatarateChannel den *router1*. Der *router1* ist ab Zeile 19 mit zwei Anschlüssen(Gates) definiert. Dies ermöglicht dem Router sowohl mit den *host1* zu kommunizieren als auch die weitere Verbindung zu dem *router2*. Dieser Router besitzt ebenfalls die zwei Gates mit denen er zum einen mit *router1* und die Verbindung zu *host2* herstellt. Auf diese Weise wird eine Kommunikation der beiden Hosts über die Router in beide Richtungen ermöglicht. Mit diesem QuicTest Netzwerk können indes Simulationen ausgeführt werden, welche exakt diese Netzwerkinfrastruktur benötigen.

Ein weiterer Teil einer Simulation wird durch die bereits erwähnten *.ini*-Dateien beschrieben. An der Stelle, an der die ned-Files die Infrastruktur beschreiben und zusammensetzen, konfigurieren die ini-Files diese Infrastruktur. Als Grundlage für die Konfiguration einer QUIC Verbindung wird die sogenannte NetPerfMeterApp verwendet. Das bedeutet mithin, dass das Erstellen der Sockets und das Setzen aller Parameter für diese Verbindung anhand dieser App geschieht.

Um den beschrieben Vorgang gewährleisten zu können ist die Konfiguration für jede Simulationen speziell und separat an das Szenario anzupassen. Aus diesem Grund besitzt jeder Test eine eigene Konfiguration für die NetPerfMeterApp, da das Verhalten der Simulation an den Testfall angepasst ist. Diese Konfiguration der NetPerfMeterApp ist der erste von insgesamt drei Bereichen, die eine Simulation als Testfall vorzuweisen haben. Ein Teil ist der zuvor beschriebene Bereich der Netzwerkdefinition in den NED-Dateien. Unabhängig von einer optionalen Beschreibung benötigt ein Test weiterhin die Konfiguration und Auflistung von benötigten Constraints, welche die auftretenden und ausbleibenden Ereignisse definieren und über Erfolg oder Fehlschlag der Testfälle entscheiden. Diese drei Bereiche einer Simulation beschreiben damit einen vollständigen Test der in Omnet++ mit dem *opp_test* Tool ausgeführt werden kann. Eine detaillierte Definition zusammen mit der Implementierung dieser drei Bereiche erfolgt in den Abschnitten der Umsetzung für dieses Konzept.

Für die Überprüfung einzelner Ereignisse der Simulation können sogenannte Constraints eingesetzt werden. Die Verwendung von mehreren Constraints sind der Hauptteil eines Testfalls, welche das Verhalten beschreiben, das durch den Testfall abgebildet werden soll. Wie im Abschnitt 3.6 bereits erwähnt, sind die Constraints fester Bestandteil in Omnet++ und können durch das *opp_test* Tool ausgewertet werden. Die Verwendung der Constraints wird von dem *opp_test* Tool in Omnet++/INET umgesetzt, um die Ausgaben von durchlaufenden Simulationen zu parsen und nach den definierten Ereignissen zu suchen. Beinhalten diese Ausgaben Konstrukte, die in den Constraints beschrieben sind, kann ein Testfall verifiziert werden. Die Constraints können unmittelbar die Logdatei Ausgaben durchsuchen um das Auftreten bestimmter Ereignisse zu verifizieren. Diese Ereignisse ermöglichen beispielsweise den Wechsel eines Klienten in den Zustand einer erfolgreich aufgebauten Verbindung bzw. den Abbruch einer solchen.

Für den QUIC Handshake wurde der Quellcode einer QUIC Verbindung analysiert. (Für ein Klassendiagramm und weitere Informationen zur Implementation von QUIC siehe Abschnitt 2.4). Da eine QUIC Verbindung einen Zustandsautomaten darstellt, konnten die notwendigen Ereignisse für den QUIC Handshake identifiziert werden und durch die Log Ausgaben beim Ausführen des Testfalls11 verifiziert werden. Das Listing 4.2 zeigt die verwendeten Constraints für den QUIC Handshake Testfall. Die drei Ereignisse, welche einen vollständigen Handshake beschreiben, sind: Das Senden eines initialen Pakets, das Empfangen des Handshake Pakets auf der Serverseite bzw. das Empfangen der Antwort auf der Initiatorseite und dem Fertigstellen der Verbindung.

```
1
2  %#-------------------------------------------------------------------------------
3  %# ========================= starting handshake message ==========================
4  %contains: test.out
5   Transition: QUIC_S_PCB_INIT --> QUIC_S_CLIENT_INIT_SENT   \
6                                   (event was: QUIC_E_OPEN_ACTIVE)
7   quic_basic: QUIC_S_PCB_INIT --> QUIC_S_CLIENT_INIT_SENT   \
8                                   (on QUIC_E_OPEN_ACTIVE)
9
10 %# ========================= server received handshake message ====================
11 %contains: test.out
12  [QUICConnection.cc][processPacketInClientInitSent:176] Enter
13  [QUICConnection.cc][processPacketInClientInitSent:180] Received handshake packet
14
15 %# ========================= client received handshake -> established =============
16 %contains: test.out
17  Transition: QUIC_S_CLIENT_INIT_SENT --> QUIC_S_ESTABLISHED   \
18                                          (event was: QUIC_E_RCV_HANDSHAKE)
19  quic_basic: QUIC_S_CLIENT_INIT_SENT --> QUIC_S_ESTABLISHED   \
20                                          (on QUIC_E_RCV_HANDSHAKE)
21
22 %#-------------------------------------------------------------------------------
```

Listing 4.2: Die Definition der benötigten Ereignisse für einen erfolgreichen Handshake

Diese drei zuvor beschriebenen Teile für Beschreibung, Konfiguration und Definition von Constraints für Simulationen erlauben eine Überprüfung und Verifizierung und beschreiben dadurch einen Testfall. In Omnet++/INET werden diese drei Abschnitte in *.test* Dateien dargestellt und können schließlich anhand von *opp_test* Tool gebündelt werden. Das in Abschnitt 3.6 beschriebene *opp_test* Tool sammelt die angegebene *.test* Datei bzw. sämtliche vorhandenen *.test* Dateien ein, sofern kein Test spezifiziert wurde. Anschließend entsteht daraus eine sogenannte Executable. Diese Executable kann letztendlich ausgeführt werden um den Test durchzuführen.

Ein mögliches Problem, das im Rahmen dieser Auswertung von Tests innerhalb einer Simulationsumgebung auftreten kann, ist die Abhängigkeit von Log Ausgaben der ausgeführten Simulationen. Es können ausschließlich Ereignisse überprüft werden, die durch eine Simulation ausgegeben werden oder letztendlich ausbleiben. Dies benötigt eine vorangegangene Analyse des Protokolls wie in Kapitel 3 um die Ausgaben in die Log-Dateien zu identifizieren. Anhand der Szenarien für die ein Testfall konzipiert wird müssen die

auftretenden Ereignisse ausgesucht werden. Anderenfalls können die Tests nicht das benötigte Verhalten verifizieren. Das bedeutet, dass die Tests anhand der vom Entwickler gesetzten Log-Ausgaben überprüft werden und demnach fehleranfällig sein können. Ein Beispiel für solche Fehler könnte das Übersehen einzelner Transitionen der Kommunikation beider Endpunkte sein. Dies hat zur Konsequenz, dass die Testfälle auf dem Wissen der Entwickler über die Implementierung von QUIC aufgebaut werden. Die Annahme in diesem Zusammenhang liegt darin, dass die Entwickler ihre Software kennen und infolgedessen die benötigten Ereignisse für einen Testfälle definieren können.

4.5 Der Aufbau des Testverfahrens

Das Testverfahren besteht aus mehreren Komponenten, deren Interaktion nicht nur eine Automatisierung sicherstellt sondern zudem ermöglicht, dass die Tests kontinuierlich ausgeführt werden. Gestartet wird dieser Prozess anhand eines Mechanismus, welcher Änderungen an der Codebasis von QUIC erkennt. Auf diese Weise ist es möglich auf jede Änderung an der Codebasis reagieren zu können. Dieser Mechanismus wird Webhook genannt. Ein Webhook ist eine Technologie, welche von Anbietern eines Versionskontrollsystems bereitgestellt wird. Eine hinreichende Erklärung zu Versionskontrollsystem wurde bereits in Abschnitt 2.1 erläutert. Da jeder Anbieter diesen Mechanismus auf unterschiedliche Weise realisiert, ist die Konfiguration eines Webhooks komplett abhängig vom Anbieter.

In dieser Arbeit wird der Webhook des Anbieters GitHub verwendet. Dieser Webhook zeichnet sich dadurch aus, dass er beliebig konfiguriert werden und somit generell auf Ereignisse reagieren kann. Ereignisse sind an dieser Stelle unter anderem Optionen wie Push, Pull, Releases, Merge Requests, Pull Requests, Tags oder Änderungen im Wiki. Zur Konfiguration eines Webhooks wird ebenfalls eine öffentlich erreichbare URL benötigt, auf die ein HTTPS POST[3][71] gesendet werden kann, sobald die vorkonfigurierten Ereignisse eingetreten sind.

Die Ressourcen für das QUIC Transportprotokoll befinden sich im INET Framework, welches in GitHub erstellt wurde. Das INET Repository *inet_private* wird in Anspruch

[3]Das HTTPS beschreibt das Hypertext Transfer Protocol Secure und wird für die Auslieferung von Webseiten verwendet. Im Internet werden über HTTPS POST/GET Daten auf dem Protokoll verschlüsselt übertragen.

genommen um für die Simulationsumgebung Omnet++ Technologien im Netzwerkbereich zu erstellen und weiterzuentwickeln. Um Änderungen an dieser Implementierung zu erhalten wird für dieses Repository ein Webhook konfiguriert. Eine der wesentlichen Komponenten in einem Webhook ist eine öffentlich erreichbare URL auf die ein HTTP/HTTPS POST ausgeführt wird, sodass andere Systeme benachrichtigt werden. Im Rahmen dieser Arbeit sitzt der Docker Server hinter einer Firewall innerhalb des Netzwerkes der HAW. Die Kommunikation erfordert einen Zugang zum internen Netzwerk ohne von der Firewall blockiert zu werden.

Der zuvor genannte erforderliche Zugang kann mittels eines Services, namens Smee.io welche die Technologie Publish-Subscribe verwendet, erreicht werden. Dieser Service bietet eine Möglichkeit Nachrichten durch eine Firewall zu schleusen. Dies wird ermöglicht, da ein Klient von innerhalb der Firewall sich bei Smee.io anmeldet (subscribe). Daraufhin können Nachrichten an die angemeldeten Klienten weitergeleitet (gepublished) werden. Eine derartige Dienstleistung wird ebenfalls vom Anbieter des Versionskontrollsystems zur Verfügung gestellt. Aus diesem Grund funktioniert dieser Dienst zuverlässig mit den im INET Repository konfigurierten Webhook da sie vom selben Provider bereitgestellt werden. Das Prinzip des Smee.io Services besteht darin, dass sich ein Klient auf die konfigurierte URL von Smee.io verbindet und danach alle dort eintreffenden Nachrichten erhält. Die Firewall wird insofern umgangen, da der Klient innerhalb der Firewall einen Kommunikationskanal nach außen ermöglicht. Für die Konfiguration des Webhooks wird die URL *https://smee.io/jvYGzCZ6LLM7mI2Z* verwendet, an welche die POST Nachrichten sendet. Des Weiteren wird ein Inhalt dieser Webhooks auf das Format *json* hinzugefügt sowie die SSL[8] Verifikation ausgeschaltet.

Wie bereits im vorherigen Abschnitt erwähnt, ist eine öffentlich erreichbare URL für die Verwendung eines Webhooks notwendig. Damit einhergehend besteht eine Abhängigkeit zu dem jeweils verwendeten Server. Im Rahmen dieser Arbeit ist das Testverfahren auf einem Docker Swarm Server der HAW angesiedelt. Durch den Einsatz des Smee.io Service können die Webhooks aus dem Repository und das Testverfahren weitergeleitet werden.

Alle Webhooks, die von Github erzeugt werden, finden sich zusätzlich auf Smee.io als HTTP/HTTPS POST wieder. Dieses POST wird anschließend von Smee.io local auf dem definierten Topic als Nachricht gesendet. Der zweite Teil des Services besteht grundsätzlich aus Klienten, im Falle des Testverfahrens reicht jedoch ein Klient auf dem Docker Swarm Server aus. Zur Verbindung mit der Smee.io Webseite um die Webhooks weiterzuleiten registriert sich der Klient mittels der zuvor genannten generierten Zeichenfolge.

Dieser Klient benötigt zum Starten lediglich zwei Parameter: Zum einen die URL von der Smee.io Website mit der generierten Zeichenfolge und zum anderen eine weitere URL, auf die Webhooks weitergeleitet werden. Das Prinzip dieses *Smee-Clienten* besteht darin das Topic der gesetzten Zeichenfolge zu abonnieren und aus allen eintreffenden Nachrichten erneut ein HTTP POST zusammenzubauen. Daraufhin erfolgt eine Weiterleitung an die gestellte URL innerhalb des von einer Firewall geschützten Netzwerkes.

Für die Kommunikation innerhalb des geschützten Netzwerkes wird kein HTTPS verwendet, da hierfür eigene Zertifikate erstellt werden müssten um mit dem Jenkins Server zu kommunizieren. Da in der Konfiguration des Webhooks auf Github die SSL Verifikation ausgeschaltet wurde, ist die Verwendung von HTTPS nicht weiter notwendig. Somit besteht die Möglichkeit die Webhooks von GitHub durch die Firewall der HAW auf den Docker Swarm Server zu Jenkins zu senden, ohne weitere Konfigurationen an der Firewall vornehmen zu müssen. Der Smee-Client wird auf dem Docker Swarm Server in einem Docker Container ausgeführt in dem der Smee-Client installiert wurde. Das Image auf Dockerhub ist unter dem Namen *mikkoe/smee-client:latest* zu finden. Das Dockerfile für den Smee-Clienten befindet sich ebenfalls in dem Repository *MikkoE/CADS_jenkins_-tools* auf Github und in den beigefügten Anlagen dieser Arbeit.

Damit steht die Technologie zur Verfügung, mithilfe derer die Webhooks auf den in Abschnitt 2.1 vorgestellten Jenkins Server weitergeleitet werden. Für diesen Jenkins Server ist bereits ein vollständiges und funktionsfähiges Dockerimage vorhanden, jedoch benötigt das Testverfahren für das Ausführen der Simulationen zusätzlich die Docker Engine. Dafür ist notwendig, dass das Jenkins Dockerimage um die Funktionalität in Listing 4.3 erweitert wird, damit der laufende Container fähig ist Docker Container zu starten und konfigurieren. Dieses Dockerfile für Jenkins ist ebenfalls in dem Repostory *MikkoE/CADS_jenkins_tools* auf Github zu finden und als vollständig gebautes Image über Dockerhub verfügbar (*mikkoe/jenkins-docker:latest*). Besonders von Bedeutung bei Jenkins ist, dass das originale Docker Image auf Java basiert (verwendet: *openjdk:8-jdk-stretch*). Aus diesem Grund kann die Docker Engine nicht wie vergleichsweise bei einem vollwertigen Unix-System über die Paketverwaltung installiert werden. Nach einem Update und der Installation von benötigten Abhängigkeiten sowie eine direkten Download der Docker Sourcen wird in Zeile 7 des Listings 4.3 der Docker Deamon installiert. Ein wichtiger Schritt ist zugleich, dass der Jenkins Nutzer zur Gruppe Docker hinzugefügt wird, siehe Zeile 10. Dieser Zustand verhindert ein notwendiges *sudo* vor den verwendeten Docker Befehlen innerhalb des Containers. Zuletzt wird der Nutzer erneut auf Jenkins in Zeile 13 geändert, da der Jenkins Nutzer sodann im laufenden Container verwendet wird

um den Jenkins Server zu verwalten und an dieser Stelle die benötigten Docker Container zu starten.

```
1  FROM jenkins/jenkins:lts
2
3  # adding docker
4  USER root
5  RUN apt-get update
6  RUN apt-get -y install $DEPENDENCIES
7  RUN apt-get update && $INSTALL_DOCKER
8
9  # add jenkins USER to docker group
10 RUN usermod -a -G docker jenkins
11
12 # changing back to original jenkins USER
13 USER jenkins
```

Listing 4.3: Das Dockerfile, das den Jenkins Server um die Installation von Docker erweitert.

Schließlich sind alle erforderlichen Docker Container für das gesteuerte Testverfahren vorhanden. Gestartet werden diese Container auf dem Docker Server der HAW. Der Vorteil ist nicht nur, dass alle Container in einem zentralen System ausgeführt und gewartet werden können sondern zusätzlich, dass dieser Server zugleich permanent online ist und somit eine hohe Verfügbarkeit bietet. Gleichzeitig ermöglicht dieser Server eine vergleichsweise stabile Sicherheit, da die Firewall der HAW einen hohen Schutz bietet. Zudem ist die Anbindung und Erreichbarkeit an das Netz deutlich stärker als beispielsweise im Vergleich zu einem Internetanschluss, der in üblichen Haushalten vorzufinden ist. Zur besseren Handhabung und Organisation der Dockerumgebung und Container wird auf diesem Server Portainer[65] eingesetzt. Portainer ist eine grafische Webanwendung um über einen Browser die Docker Engine, Container, Images, Volumes und jegliche zur Konfiguration betreffend, die zur Dockerumgebung gehören, zu überwachen und konfigurieren. Über Portainer werden in diesem Fall der Smee Client und Jenkins gestartet.

Der Smee Client wird als einfacher Service innerhalb von Portainer gestartet und bekommt ausschließlich die beiden Parameter als Startargumente beim Hochfahren übergeben. Ein Service besteht aus einem Docker Image, einem Namen, die Häufigkeit der Replikation und der notwendigen Konfiguration. Die Konfiguration kann bestimmte Startparameter definieren, persistente Verweise zum Datenspeicher setzen, separate Netzwerke definieren oder ein Portmapping vornehmen. Diese und alle weiteren Konfigurationen

können in der Weboberfläche in Portainer eingestellt und gespeichert werden. Diese Konfiguration wird nicht selbstständig gesichert und sollte der Service gelöscht werden, ist dementsprechend die zugehörige Konfiguration entfernt. Sind die Anwendungen komplexer oder benötigen mehrere Images mit zugehöriger Konfiguration werden sogenannte Stacks für die Beschreibung der Anwendungen verwendet.

Eine Möglichkeit bei Docker ist die Verwendung des Kommandos *docker-compose*. Dieses Kommando ermöglicht das Starten von mehreren Services und Einrichten der jeweiligen Konfiguration mit lediglich einem Befehl. Der Befehl verwendet eine *docker-compose.yml* Datei um somit eine Anwendung mit allen Abhängigkeiten und Konfigurationen zu beschreiben. In einer Docker Swarm Umgebung wird der gleiche Mechanismus mit der *docker-compose.yml* Datei verwendet um Anwendungen zu starten. Im Docker Swarm heist das Starten einer solchen Datei, einen Stack starten. Dabei werden vom Docker Stack die Keywords, welche lediglich für Docker Compose verfügbar sind, ignoriert und umgekehrt. Dieses Verhalten ermöglicht eine Kompatibilität zwischen den beiden Befehlen beim Verwenden der gleichen Source Datei. Für Jenkins existiert eine derartige Datei, welche die Konfiguration und die Startbefehle enthält. Das Listing 4.4 zeigt den Ausschnitt der den Jenkins Service für den Stack definiert. Diese Datei kann direkt über Portainer hochgeladen und als eigener Stack gestartet werden. Die Datei ist ebenfalls im Hauptordner des CADS_jenkins_tools Repository und bei den angehängten Ressourcen dieser Arbeit zu finden.

```
1   jenkins:
2     image: mikkoe/jenkins-docker:latest
3     environment:
4       - DOCKER_HOST=http://cads-docker.cpt.haw-hamburg.de:80
5     ports:
6       - '11001:8080'
7       - '50000:50000'
8     volumes:
9       - jenkins-data:/var/jenkins_home
10      - /var/run/docker.sock:/var/run/docker.sock
```

Listing 4.4: Die *docker-compose.yml* Datei, welche benötigt wird um einen Stack mit vollständiger Konfiguration zu starten.

Das Keyword *jenkins* in Zeile 1 leitet einen neuen Serviceabschnitt ein und beschreibt den Namen des Services für den zu startenden Jenkins Server. Dieser benutzt das in Zeile 2 definierte Docker Image. Hier wird das bereits zuvor erwähnte erweiterte Jenkins

Image vom Dockerhub verwendet. Der Abschnitt *environment* beinhaltet alle zusätzlichen Parameter und Konstanten, die innerhalb des laufenden Containers in Gebrauch sind. In diesem Abschnitt wird das Portmapping definiert, das anhand von Docker Engine an die Container weitergereicht werden sollen. Das Prinzip dahinter lautet wie folgt: Mehrere Anwendungen, die mit Docker Engine verbunden sind, können einen Webservice zur Verfügung stellen, der den Port 80 oder 8080 verwendet. Somit definiert der Befehl nach dem Schema *<öffentlicherPort>:<containerPort>* das Portmapping. Am Beispiel von Jenkins wird für das Webinterface (standardmäßig über den Port 8080 zu erreichen) der öffentliche Port 11001 definiert und durch die Docker Engine schließlich intern für den Jenkins Container auf 8080 gemapped. Das zweite Portmapping (5000 : 5000) wird für die Kommunikation zwischen dem Jenkins Master und möglichen hinzugefügten Slaves benötigt. Slaves werden in dieser Arbeit nicht verwendet, da die Arbeitsbelastung von Jenkins durch das gesteuerte Testverfahren mit einer Pipeline nicht ausgelastet ist. Zusätzlich wird für die Erstellung eines Jenkins Slaves mehr administrativer Aufwand benötigt als für die Demonstration des gesteuerten Testverfahrens notwendig ist.

Der bedeutendste Abschnitt ist der Bereich ab Zeile 8, die Volumes. Ein Volume ist eine Referenz auf einen persistenten Bereich des Host-Speichers um Dateien zugleich nach Beendigung eines Containers nicht zu verlieren. Im Falle von Jenkins liegt auf diesem Speicher die komplette Konfiguration von der Benutzersteuerung über die installierten Plugins bis hin zu den erstellten Pipeline Jobs. Das Volume *jenkins-data:/var/jenkins_home* beschreibt das von Portainer verwaltete Volume mit dem Namen jenkins-data und den Pfad an dem dieser persistente Speicherbereich innerhalb des Container Dateisystems gemountet werden soll. Innerhalb des laufenden Jenkins Containers erscheint letztendlich der Ordner unter */var/jenkins_home* als reguläre Datei im vorhandenen Dateisystem. Folglich hat der Container keine Kenntnis über seinen Betriebssystemkontext hinaus.

Die letzte Zeile (Zeile 10) zeigt eine Besonderheit von Docker. Für die Identifikation, das Starten von Containern und dem Errichten von Images wird mittels Docker Engine eine Datei mit dem Namen *docker.sock* verwendet. Unterschiedliche Dateien haben eine differenzierte Dockerumgebungen zur Folge. Dadurch würden beide Dockerumgebungen getrennt voneinander agieren und können deshalb nicht untereinander kommunizieren. Folglich kann keine Kommunikation zwischen Containern hergestellt werden die auf unterschiedlichen Dockerumgebungen gestartet wurden.

Sofern der Fall eintritt, dass Anwendungen innerhalb eines Containers wie zum Beispiel Jenkins zugleich Docker ausführen, ist im nächsten Schritt die Überlegung, welche Funk-

tionen in diesem Zusammenhang benötigt werden. Diese Lock Datei wird dafür in den Container gemounted um Container auf der Docker Engine des Host ausführen zu können. In dem Jenkins Container ist Docker installiert und kann somit innerhalb einer Pipeline ein Docker Container gestartet werden. Die Docker Engine im Container hat allerdings keinerlei Informationen darüber, dass diese selbst innerhalb eines Containers arbeitet. Startet Jenkins innerhalb der Pipeline einen Container wird auf dem Docker Swarm Server ein Container gestartet. Ohne ein gemoutete Lock Datei laufen im Jenkins Container mithin ebenfalls ein oder gegebenenfalls mehrere Container mit den Ressourcen des Jenkins Containers. Um das Docker im Docker zu unterbinden kann die *docker.sock* Datei in den Jenkins Container hilfreich sein. Dabei werden die beiden Docker Engines als eine Instanz betrachtet. Container, die Jenkins startet, werden letztendlich in diesem Sinne nicht mehr innerhalb des Jenkins Containers, sondern durch die *docker.sock* auf dem Docker Server und somit parallel zu Jenkins gestartet. Die Abbildung 4.3 zeigt eine Gegenüberstellung der Docker in Docker Variante zu der mit einer gemounteten *docker.sock* Datei. Dies zeichnet einen weiteren Vorteil von einer geteilten *docker.sock* Datei aus, bei dem unterschiedliche Systeme parallel auf der selben Dockerumgebung gestartet werden können.

Grundsätzlich zeigt die Abbildung 4.3 auf der rechten Seite eine Möglichkeit Docker Container zu starten, welche wiederum Docker Container starten. Zu sehen ist das an dem Jenkins Servers, welcher intern einen weiteren Container für die Omnet++ Umgebung startet. Hierbei existiert innerhalb des Jenkins Servers eine separat arbeitende Docker Engine. Auf der linken Seite in der Abbildung ist dieselbe Kombination der Container für Smee CLient, Jenkins und Omnet++ beschrieben. Ein Vorteil auf der linken Seite ist, dass alle Container die selbe *.lock* Datei verwenden. Dadurch erhält die Docker Engine innerhalb des Jenkins Containers Zugriff auf die Docker Engine des Hosts.

4.6 Die Jenkins Pipeline

Der laufende Container bietet zusammen mit dem Automatisierungsserver Jenkins eine eigene Webumgebung, in der - nach einem Login - Pipelines konfiguriert und gestartet werden können. Eine derartige Pipeline ist eine formale Beschreibung von Arbeitsschritten, mit der es möglich ist, diese nacheinander bzw. parallel auszuführen. Jenkins besitzt für unterschiedliche Arbeiten und Arbeitsabläufe mithin andere Definitionen für Pipelines. Dafür gibt es die sogenannten Rubriken “Free Style“, Maven, Pipeline, Folder und

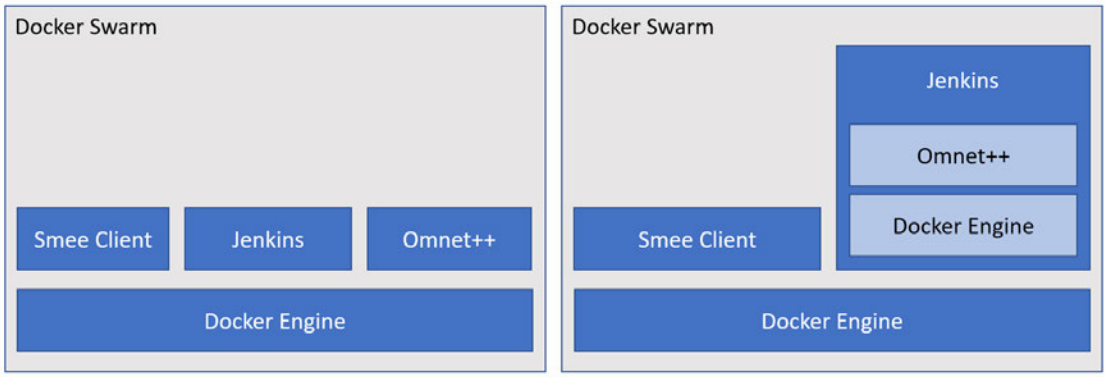

Abbildung 4.3: Die Docker Engine als Docker im Docker und mit geteilter *.lock* Datei.

Multibranch, welche anhand einer differenzierten Basis einen Ablauf von Arbeitsschritten beschreiben. Für diese Arbeit hat das Standard Pipeline Projekt Verwendung gefunden. Dieses wird als GitHub Projekt Pipeline konfiguriert. Das bedeutet, dass die Dateien für die Pipeline in einem Repository auf GitHub aufgenommen und von dieser Stelle administriert werden ohne unmittelbar Konfigurationen in Jenkins vornehmen zu müssen. Die Dateien befinden sich dementsprechend in dem bereits vorgestellten CADS_jenkins_tools Repository. Der Name dieser Datei ist frei wählbar, allerdings verwendet die Pipeline die Groovy Sprache und benötigt aus diesem Grund eine Pipeline Datei mit der Endung *.groovy*. Das Listing 4.5 zeigt die in dieser Arbeit verwendeten Arbeitsschritte in der Pipeline. Innerhalb dieser Pipeline wird zudem der Docker Container gestartet und die Tests ausgeführt. Das vollständige Pipeline Skript befindet sich in dem erwähnten Repository und bei den Ressourcen dieser Arbeit.

```
1  steps{
2    echo 'clone das Repository in unterordner'
3        dir('inet-private'){
4          git checkout INET REPOSITORY
5
6          init benoetigte Submodule
7        }
8  }
9  steps{
10   echo 'omnetpp im docker container'
11   script{
12       def image = docker.image('mikkoe/omnetpp-inet-docker')
13       image.pull()
14       image.inside{
15           echo 'Arbeitsschritte innerhalb des Docker Containers'
16           sh 'omnetpp downloaden und installieren'
17           sh 'omnetpp konfigurieren'
18
19           sh 'das inet repository compilieren'
20
21           sh 'die Ordner fuer die Tests vorbereiten'
22
23           sh 'die Tests an die richtigen Stellen kopieren'
24
25           sh 'Tests ausfuehren'
26       }
27   }
28 }
```

Listing 4.5: Das Pipeline Script welches den Arbeitsschritt zum Ausführen der Tests in Form eines Pseudocodes beinhaltet.

Um die einzelnen Arbeitsschritte der Pipeline durchlaufen zu können, erledigt Jenkins einen Checkout auf das Repository CADS_jenkins_tools. In diesem Repository befinden sich die Ressourcen für das Ausführen der Pipeline. Das Pipeline Script aus Listing 4.5 wird damit geladen und die Arbeitsschritte in der Pipelinedatei nacheinander ausgeführt.

Der erste Arbeitsschritt im Listing 4.5 zeigt das Clonen des aktuellen Standes des INET Repositories in den Unterordner *inet-private*. Dafür wird in der Konfiguration beim Git Plugin der Branch angegeben, der geclont werden soll. Weiterhin veranlasst eine credentialId, welche für das Git Plugin benötigt wird, die in Jenkins hinterlegten Nutzernamen

und Passwörter zu benutzen, das Repository von der genannten URL zu clonen. Dies ist notwendig, da das INET Repository ein privates Repository ist und ausschließlich befugten Nutzern Zugriff auf die Daten gewährt. Da in dem INET Repository bestimmte Bereiche der Software als Submodule definiert sind, werden diese zuletzt initialisiert. Damit ist letztendlich der Arbeitsschritt beendet und alle erforderlichen Ressourcen vorhanden.

Die Zeile 13 definiert das erstellte Image für die Omnet++ Umgebung. Dieses ermöglicht anschließend einen Download in Zeile 14 vom Dockerhub, sofern eine neuere Version existiert. Die Docker Engine hält zuvor gestartete Images von Containern lokal gespeichert und überprüft vor jedem Start ob eine aktuellere Version verfügbar ist. Der Befehl *inside* beschreibt den Start eines sogenannten Blocks. Dieser Block ist ein Abschnitt innerhalb dessen alle definierten Kommandos in dem Omnet++ Docker Containers ausgeführt werden.

In diesem Zusammenhang bewirkt dieser Block namens *inside*, dass Befehle, für den Docker Container gewrapped werden. Also auf Befehle für die Docker Engine übersetzt und damit innerhalb der Umgebung des Docker Containers ausgeführt. Damit sind diese Befehle losgelöst vom eigentlichen Jenkins Workspace. Eine Besonderheit, die Jenkins verwendet, ist, den lokalen Workspace als ein Volume beim Starten des Docker Containers mit anzugeben, sodass dieser innerhalb des Containers verfügbar ist.

Die folgenden Befehle sind ausschlaggebend für die Vorbereitungen und das Ausführen der Tests. Obwohl in einem Docker Container beim Starten standardmäßig *root* (Administrator) der Nutzer ist, wird jedoch innerhalb der Jenkins Umgebung für Docker standardmäßig der Jenkins Nutzer verwendet. Dadurch ist Omnet++ innerhalb des Docker Containers für diesen Nutzer zu installieren und konfigurieren. Die Befehle hinter der Zeile 19 wechseln in das INET Repository, welches im vorherigen Arbeitsschritt der Pipeline in den Workspace geclont wurde. Dort werden die Ressourcen gebündelt und das INET Framework kann kompiliert werden. Damit sind sämtliche Abhängigkeiten konfiguriert und die Omnet++/INET Funktionalität findet folglich Anwendung.

In den Zeilen 21 und 23 wird im Anschluss der Ordner und alle Abhängigkeiten für die Tests vorbereitet. Dazu gehört darüber hinaus, dass eine nicht benötigte Datei gelöscht wird, welche Probleme mit den Namen für das Netzwerk erzeugen kann. Diese Datei stammt von der Implementierung des QUIC Transportprotokolls und beschreibt ein Basis Netzwerk für QUIC. In der folgenden Zeile 23 wird die verwendete .ned Datei an die richtige Stelle im INET Repository kopiert. Danach wird der Testfall für den

QUIC Handshake in den Testordner kopiert. Die Testdatei liegt wie in Abschnitt 4.5 beschrieben, ebenfalls in dem CADS_jenkins_tools Repository. In Zeile 25 steht somit der Aufruf für das *opp_test* Tool, welches durch das Skript *./runtest* ausgeführt wird und schlussendlich den spezifizierten Test ausführt. Damit ist der für die Pipeline relevante Abschnitt, welche die Tests in der Simulationsumgebung ausführt beendet. Das Testtool generiert einer Auflistung über erfolgreiche oder fehlgeschlagene Tests mithilfe dessen schließlich eine Aussage über den Zustand des Programmcodes getroffen werden kann. Das Resultat zeigt sich in einer erfolgreichen oder fehlschlagenden Pipeline in der Jenkins Weboberfläche.

4.7 Der Arbeitsablauf des Testverfahrens

Der kontinuierliche Arbeitsdurchlauf des Testverfahrens verwendet die Webhooks als Startsignal für einen Testdurchlauf. Die Abbildung 4.4 zeigt den Arbeitsablauf des gesteuerten Testverfahrens nachdem ein Webhook gesendet wurde. Der komplette Arbeitsablauf stellt das Senden des Webhooks durch die Firewall mittels des Smee Clients dar und das Senden an den Jenkins Server. Der Jenkins Server startet daraufhin eine neue Pipeline, die den aktuellen Stand des Repositories verwendet, um sowohl das INET Repository zu bauen, zu installieren als auch die Tests auszuführen. Sollte ein Testfall fehlschlagen, wird die komplette Pipeline als Fehlschlag gewertet. Unabhängig von der Anzahl jener Entwickler, die zu derselben Zeit an dem Repository arbeiten, können zugleich mehrere Testdurchläufe hintereinander gestartet werden. Unter der Voraussetzung, dass die Testdurchläufe schneller gestartet werden als die Pipeline fertig gestellt ist, landen die nachfolgenden Testdurchläufe in einer von Jenkins bereitgestellten Warteschlange und werden in dieser Reihenfolge abgearbeitet. Eine detaillierte Beschreibung der einzelnen Schritte im Arbeitsablauf befindet sich in der Tabelle 4.2.

Der in Abbildung 4.4 dargestellte Arbeitsablauf beschreibt ausschließlich die für das Testverfahren relevante Kommunikation. Es grenzt folglich eine zusätzlich erwähnenswerte Kommunikation für Clonen der Repository aus. Ebenfalls findet die Kommunikation für das Starten, Überwachen und Updaten der Services keine Berücksichtigung. Da diese Faktoren keinen direkten Einfluss auf das gesteuerte Testverfahren nehmen, sondern lediglich zusätzliche Daten bereitstellen, wurde diese Kommunikation nicht im Arbeitsablauf berücksichtigt.

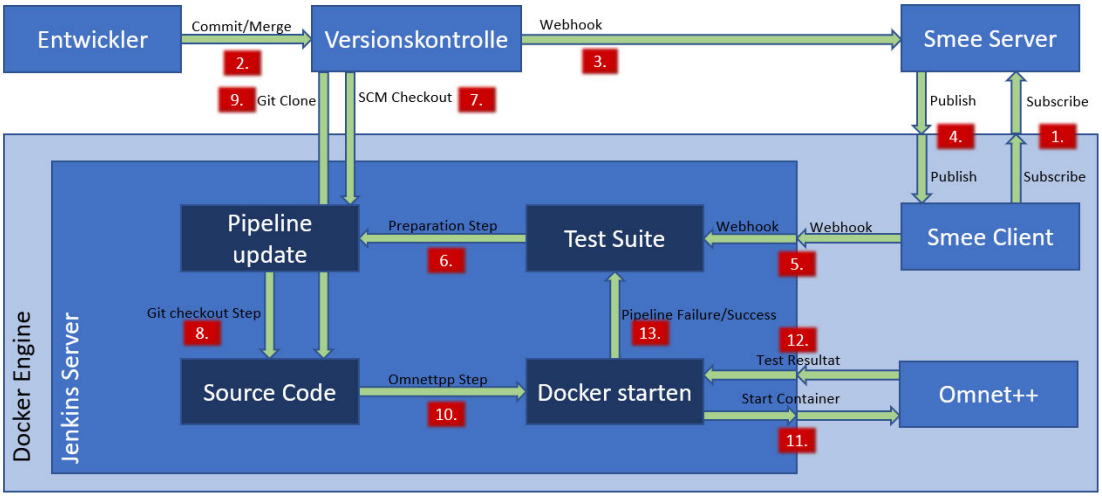

Abbildung 4.4: Der Arbeitsablauf für das gesteuerte Testverfahren mit dem auftretenden Informationsaustausch zwischen den Technologien.

Ein Entwickler stößt diesen Arbeitsdurchlauf an, indem er Änderungen an der Software mit einem Branch im Versionskontrollsystem zusammenführt. Daraufhin wird ein Webhook generiert, welcher an den konfigurierten Smee Server gesendet wird. Dieser sendet den Webhook an alle Klienten, die sich zuvor mittels eines Subscribes auf diesen Server angemeldet haben. Der Smee Client, der bereits hinter der Firewall gestartet wurde, kann an dieser Stelle den Webhook auf den in der Docker Engine laufenden Jenkins Server weiterleiten. Dieser Jenkins Server startet somit innerhalb eines Pipeline Durchlaufs einen Omnet++ Container, in welchem die Tests ausgeführt werden und über Fehlschlag oder Erfolg des Arbeitsdurchlaufs entscheiden. Innerhalb des Jenkins Servers werden dafür zunächst die Pipeline mit dem Namen Test Suite gestartet und ein aktueller Softwarestand aus dem Versionskontrollsystem kopiert. Danach sind alle benötigten Ressourcen und der Source Code vorhanden, sodass Jenkins den Omnet++ Docker Container startet. Ist dieser Container schließlich beendet wird der Arbeitsdurchlauf abhängig von dem Resultat mit Erfolg oder Fehlschlag markiert.

Schritt	Eventbeschreibung
1.	Der Smee Client verbindet sich mit dem Smee Server außerhalb der Firewall. Die Verbindung verwendet das Publish/Subscribe verfahren.
2.	Ein Entwickler veröffentlicht eine Änderung mittels Commit oder Merge auf den Hauptbranch.
3.	GitHub produziert aus dem Commit/Merge einen Webhook der auf dem Smee Server konfiguriert ist.
4.	Der Smee Server veröffentlicht den Webhook an alle bekannten Empfänger.
5.	Der Smee CLient baut die Nachricht vom Smee Server wieder zusammen und leitet den Webhook an den Jenkins Server weiter. Jenkins weiß zu welcher Pipeline dieser Webhook gehört und startet die Test Suite.
6.	Zum Start der Pipeline muss die Konfiguration geladen sowie der temporäre Workspace für den aktuellen Durchlauf erstellt werden.
7.	Da die Pipeline in einer Pipeline.groovy Datei in dem CADS_jenkins_-tools Repository liegt wird zunächst ein SCM Checkout durchgeführt um die Konfiguration der Pipeline zu laden.
8.	Der Git Checkout Schritt soll den Arbeitsbereich für INET und Omnet++ vorbereiten.
9.	Hier werden die Daten des INET Repositories in den Arbeitsbereich geclont und die Submodule initialisiert.
10.	Der Omnetpp Schritt ist der eigentliche Zusammenbau aller Ressourcen und das Ausführen der Tests.
11.	Da die Simulationsumgebung innerhalb eines separaten Docker Containers abläuft wird hier der Omnet++ Docker Container gestartet und bekommt die notwendigen Befehle für die Installation und Ausführung der Tests (siehe 4.5).
12.	Dieser Schritt beinhaltet den Abschluss der Tests, das Beenden des Containers und das Überbringen der Testresultate an die Pipeline.
13.	Das Beenden der Pipeline zusammen mit der Auswertung der Pipeline. Ein Erfolg oder Misserfolg einer Pipeline wird im Jenkins Server visuell dargestellt zusammen mit den weiteren Informationen zu den einzelnen Steps der Pipeline.

Tabelle 4.2: Eine detaillierte Beschreibung der Events die innerhalb des Testverfahrens auftreten.

5 Evaluation

Im nachstehenden Kapitel erfolgt die Evaluation zum Thema der Thesis. Grundsätzlich setzt sich eine Evaluation mit der aufgestellten Hypothese, dem aufgebauten, gesteuerten Testverfahren und letztendlich mit den daraus resultierenden Daten auseinander. Darauf soll im weiteren Verlauf eingegangen werden. Zusätzlich tragen Antworten einer Umfrage zum Einsatz eines gesteuerten Testverfahrens zur Auswertung bei. Dies beinhaltet darüber hinaus die Auswertung der Szenarien zu betrachten, die für den Vergleich zum gesteuerten Testverfahren erzeugt wurden. Im Anschluss daran werden sämtliche Ergebnisse und Beobachtungen, die im Rahmen dieser Arbeit gesammelt wurden, erläutert und mithin in den Aufbau eingegliedert. Als letztes werden die gruppierten und zugeordneten Daten ausgewertet und mögliche Argumentationen daraus abgeleitet. Diese Evaluation verknüpft die Daten aus dem Konzept und der Umsetzung zusammen mit der Analyse.

5.1 Vorgehensweise

Als Unterstützung der Arbeit von Entwicklern bei der Implementierung von komplexer Software kann ein gesteuertes Testverfahren eingesetzt werden. Das hier genannte Testverfahren unterstützt die Entwickler in ihrer täglichen Arbeit, indem diese ein verhältnismäßig schnelles Feedback zu der Implementierung erhalten. Die Idee für ein solches Testverfahren besteht in einer einfachen Visualisierung für den jeweiligen Entwickler ,bei dem dieser unmittelbar den Status erkennen kann. Der Vorteil entsteht dadurch, dass die Entwickler nicht selbständig den Programmcode überprüfen müssen, sondern dies parallel zu ihrer Arbeit vom gesteuerten Testverfahren übernommen wird. Die Virtualisierung über den Zustand des Programmcodes ist anhand der roten oder grünen Felder ersichtlich, die beschreiben, inwieweit die Software lauffähig implementiert und getestet ist. Jedoch bleibt festzuhalten, dass die Implementierung und die Tests weiterhin bei den Entwicklern liegen und ausschließlich die Auswertung von Tests und Ausführung der Software von dem gesteuerten Testverfahren durchgeführt werden können.

Für das vorliegende Testverfahren wurde zunächst analysiert welche Funktionalität erforderlich ist und an welcher Stelle eventuelle Grenzen für die Ausführung bestehen. Da ein derartiges System aus mehreren Schritten und unterschiedlichen Technologien zusammengesetzt wird, lag der Fokus zuerst darin, welche Arbeitsschritte mittels welcher Technologien abgebildet werden können. Nach einer Einführung von Begrifflichkeiten und deren Besonderheiten zum Verständnis des Kontextes dieser Arbeit wurde die Umsetzung mit möglichen Technologien analysiert. Dafür wurde die Problemstellung in Abschnitt 3.1 im Detail ausgeführt und mit Beispielen aus der Industriebranche unterlegt. Danach wurden die möglichen Technologien auf ihren Einsatz analysiert und beschrieben.

Neben der Auswahl und Analyse der verwendeten Technologien wurde in Abschnitt 4 ein Konzept erarbeitet, welches die Hypothese belegen soll. Unterstützend soll eine Umfrage zur Belegung beitragen. Das Konzept für ein gesteuertes Testverfahren soll einen effektiveren Einsatz von Ressourcen ermöglichen. Das Konzept basiert auf dem zuvor beschriebenen Szenario: In welcher Weise ein Entwickler Arbeiten an einem Repository durchführt und an einer bestehenden Software arbeitet.

5.2 Datengenerierung und Beobachtungen

Unter Zuhilfenahme des QUIC Handshakes wurde ein Demotestfall erstellt, welcher als Grundlage dient um den vollständig Ablauf des gesteuerten Testverfahrens zu untersuchen. Nach den Durchläufen des Testverfahrens sind die Daten in drei verschiedenen Kategorien zu differenzieren. Die erste Kategorie beschreibt das Feedback eines derartigen gesteuerten Testverfahrens und die damit entstehende Visualisierung über fehlschlagende Testdurchläufe. Dabei besteht zusätzlich die Ansicht über die Historie der durchlaufenen letzten Testläufe. Die zweite Kategorie beinhaltet sämtliche Daten, die während eines Testdurchlaufs gesammelt werden. Dazu gehört unter anderem der Log Output auf dem Jenkins Server. Hier wird die komplette Konfiguration der Pipeline und der Testfälle sowie der verwendete Stand aus dem Repository aufgelistet. Eventuelle Probleme oder fehlschlagende Tests sind ebenfalls in dieser Ausgabe dargestellt. Die dritte und letzte Kategorie beinhaltet sämtliche Daten, welche Omnet++ allein beim Ausführen der Testszenarien generiert. Diese liegen in einem Unterordner der jeweiligen Simulation, dem *out* Ordner.

Visualisierung

Dieser Abschnitt behandelt die Visualisierung und inwiefern diese mittels Jenkins Server umgesetzt und den Entwicklern zur Verfügung gestellt wird. Diese Visualisierung erfolgt anhand von Jenkins bereitgestellten Mitteln. Jenkins beschreibt die Historie von Pipeline Durchläufen auf zwei unterschiedliche Weisen. Die erste zeichnet sich durch eine prägnante Übersicht über die letzten Durchläufe der Pipeline, wie sie beispielhaft in Abbildung 5.1 dargestellt wird, aus. Die Anzahl der in Jenkins gespeicherten Durchläufe kann beliebig konfiguriert werden. Anhand der vorgehaltenen Nummer an Pipelines und eines zeitlichen Verfallsdatums für die einzelnen gespeicherten Durchgänge können die Daten von zu alten Durchgängen gelöscht werden. Eine Deaktivierung beider Einstellungen verursachen einen hohen Speicherverbrauch, indem alle getätigten Durchläufe auf dem Jenkins Server gespeichert bleiben. Zusätzlich zeigt Jenkins für eine Pipeline Symbole an, die an einen Wetterbericht erinnern. Diese beschreiben mittels Symbol die Historie der letzten Jenkins Durchläufe. Somit steht beispielsweise eine Sonne dafür, dass alle zuletzt durchgeführten Testdurchläufe erfolgreich waren und ohne Fehler abgeschlossen wurden. Entsprechend steht das Zeichen "bewölkt"für ein oder mehrere Testdurchläufe, die fehlgeschlagen sind. Weiterhin zeigt in diesem Sinne ein Gewitter-Symbol, dass keiner der letzten fünf Testdurchläufe fehlerfrei war.

Die Abbildung 5.1 zeigt blaue und rote Punkte, die für erfolgreiche bzw. fehlschlagende Durchläufe stehen. Zeigt die Anzeige einen grauen Punkt an, wurde dieser Durchlauf durch äußere Einflüsse beendet. Das kann als Beispiel ein Anwender sein, der seinen Durchlauf stoppt und damit die Pipeline nicht vollständig zu Ende gelaufen ist. Ein Grund hierfür könnte die eigene Prognose sein, dass dieser nicht erfolgreich sein wird und damit keine Ressourcen belegen soll. Auch das Stoppen von Docker Containern oder anderer Software, die für einen erfolgreichen Durchlauf benötigt wird, erzeugt einen Abbruch der Pipeline. Die Übersicht zeigt direkte Indizien über Erfolg oder Misserfolg der Pipeline. Weitere grundlegende Informationen zu den konkreten Durchläufen bietet Jenkins zum Beispiel anhand einer Übersicht an, die weitere Details zu einem Durchlauf darstellt. Diese detaillierte Ansicht ist die Erweiterung der einfachen Liste an Durchläufen, welche in Abbildung 5.1 dargestellt ist.

Die Abbildung 5.2 stellt eine detaillierte Aufstellung der letzten Durchläufe dar. In dieser Übersicht ist neben der Durchlaufnummer, dem Datum und dem Erfolg bzw. Misserfolg eine Beschreibung der einzelnen Schritte der Pipeline zu sehen. Jeder Schritt ist als Liste aufgeführt und mit einer Dauer versehen, die für die Ausführung des benötigten Schritts

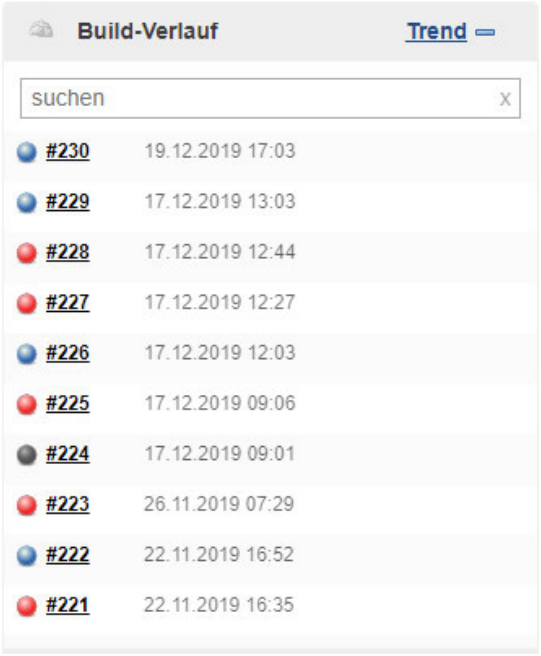

Abbildung 5.1: Kurze Übersicht der von Jenkins bereitgestellten Historie über die letzten Pipeline Durchläufe.

gebraucht wurde. Rot hervorgehobene Kästchen beschreiben dabei einen Arbeitsschritt in dem ein Fehler auftreten ist und der die Pipeline als Fehlschlag markiert. Aktuelle in der Ausführung befindliche Durchläufe werden mit einem blauen Fortschrittsbalken dargestellt. Dieser Fortschrittsbalken richtet sich nach der an den Zuständen ermittelten durchschnittlichen Laufzeit der Arbeitsschritte. Die dort gelisteten Arbeitsschritte der Pipeline decken sich mit dem in Abschnitt 4.7 beschriebenen Übergänge und spiegeln diese Schritte der Jenkins Pipeline aus der Abbildung 4.4 wieder.

Nicht nur das Datum und die Durchlaufnummer werden für jeden Durchlauf angezeigt, sondern auch die Anzahl der Änderungen, welche in dem Repository hinzugekommen sind. Dadurch wird jedem Nutzer anhand eines ersten Blicks ermöglicht alle relevanten Informationen zu erhalten. Die Anzahl der Änderungen beschreibt keineswegs funktionalen Änderungen und lässt keine weitere Schlüsse zu den tatsächlichen Änderungen zu. Es bietet sich lediglich ein kurzer Überblick über den Zustand des Systems. Dies trägt zu den Entscheidungen bei inwiefern der Anwender weiter agieren muss: Kann der Entwickler regulär weiterentwickeln oder tritt ein eventueller Fehler innerhalb des Testphase ein, der eine nähere Begutachtung bedarf? Somit ist die Visualisierung eine effiziente Darstellung, indem eine unmittelbare Rückmeldung für getätigte Änderungen einer Entwicklung

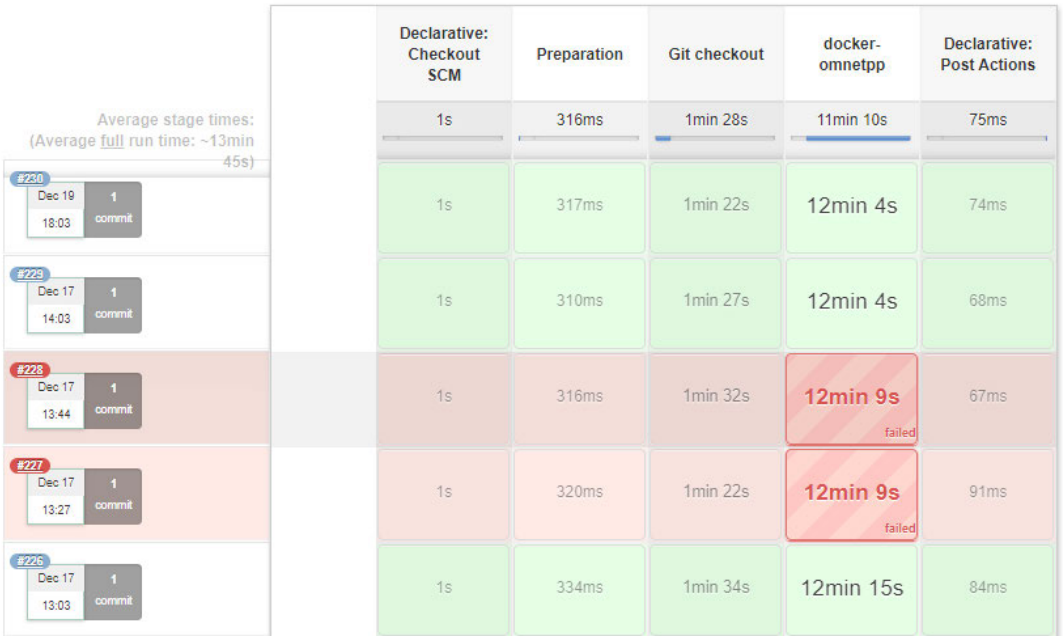

Abbildung 5.2: Detailierte Aufstellung der letzten Pipeline Durchläufe mit einer Zeitangabe und der Markierung eines Schrittes in der die Pipeline fehlschlägt.

erscheint.

Pipelinedaten

Zusätzlich zu der Visualisierung existieren weitere Daten, die während eines Durchlaufs der Pipeline generiert werden. Die Daten befinden sich innerhalb der einzelnen Testdurchläufe. Jede angestoßene Pipeline erzeugt auf der Kommandozeile in Jenkins eine Logausgabe. Indem sowohl die Ausgabe über den aktuellen Bereich der Pipeline als auch jene Ausgabe der ausgeführten Programmschritte innerhalb der Arbeitsschritte Dokumentation über die Ausführung bereitstellen werden hier Daten angesammelt. Wie bereits in Abschnitt 4.6 beschrieben, beinhaltet die Pipeline mehrere nacheinander ausgeführte Arbeitsschritte. Diese einzelnen Schritte werden in den Logausgaben der Pipeline in Jenkins sichtbar. Die Abbildung 5.3 beschreibt beispielhaft die Unterscheidung von Jenkins eigener Pipelineausgabe und dem im Arbeitsschritt ausgeführten Kommandos. Dies führt dazu, dass die kompletten Daten zu einem Durchlauf der Pipeline innerhalb dieser Übersicht für die Pipeline zusammengefasst sind. Abhängig von der ausgeführten Software und den Arbeitsschritten kann diese Logausgabe ausführlich werden. Bereits die Aus-

```
19:17:12  opp_test.py: running tests using work_dbg...
19:17:12  *** QuicHandshake.test: PASS
19:17:12  ======================================
19:17:12  PASS: 1   FAIL: 0   UNRESOLVED: 0
19:17:12
19:17:12
19:17:12  Results can be found in ./work
[Pipeline] }
19:17:12  $ docker stop --time=1 5af3e3328371d535a1396fab52e17c8af17133237aeee6a77399cf57ef2749e4
19:17:14  $ docker rm -f 5af3e3328371d535a1396fab52e17c8af17133237aeee6a77399cf57ef2749e4
[Pipeline] // withDockerContainer
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
```

Abbildung 5.3: Beispielhafte Ausgabe der Logdateien von Jenkins zur Unterscheidung von Ausgabe der Pipeline und der ausgeführten Kommandos

gabe innerhalb der Pipeline für die Kommandos im Docker Container sind lang, sodass Jenkins aus Performance-Gründen lediglich die letzten Zeilen der Logausgabe anzeigt.

Diese Logausgaben der Pipeline beschreiben das unmittelbare Verhalten des gesteuerten Testsystems. Fehlschlagende Pipelines erzeugen ein Fehlerprotokoll innerhalb der Logausgaben. Eine Fehleranalyse kann direkt in dieser Logausgabe gestartet werden. Es gibt zwei Varianten für fehlschlagende Pipelines. Die erste Variante sind implementierte Fehler in der Software. Dazu gehören zum Beispiel Inkompatibilitäten zwischen bestimmten Teilen der Software. Diese Teile der Software können beispielsweise die Änderung eines Datentyps sein der ältere Versionen der Software nicht mehr vollständig ausführen kann. Dieses Beispiel wurde bereits im Abschnitt 4.3 beschrieben und ist der Grund, das für diese Arbeit ausschließlich eine maximale Version für Omnet++ verwendet wird. Da das INET Framework innerhalb von Omnet++ agiert und in der Pipeline Omnet++ kompiliert wird, können ebenfalls Probleme mit neu hinzugefügten Änderungen entstehen. Diese Änderungen beschreiben Fehler, welche durch eine Erweiterung eingepflegt werden. Teile dieser Änderungen können bereits bestehende Funktionalität beschädigen oder das Verhalten der Software verändern.

Die zweite Variante für fehlschlagende Pipelines sind die Tests an sich. Ein fehlerhafter Testfall wird in den Logausgaben mit FAIL (siehe Aufstellung in Abbildung 5.3) markiert und das Omnet++ Testprogramm beendet sich automatisch mit einem Fehlercode. Anhand der Übersichten im zuvor beschriebenen Abschnitt kann der Status eines Durchgangs des gesteuerten Testverfahrens eingesehen werden. Die Probleme bei einem Durchlauf der Pipeline können in diesen Logausgaben ermittelt werden. Diese ermöglichen den

Einstieg im Programmcode für die Fehlersuche. Dabei sind die Fehlermeldungen ergiebig, basierend darauf, wie die Entwickler bei der Implementierung mit der Fehlerbehandlung umgehen. Zusammengefasst bedeutet dies, dass sich Entwickler in den Logausgaben kann ein Fehler effizienter gefunden und identifiziert werden kann, sofern eine ausgiebige Dokumentation des aktuellen Systemzustandes existiert.

Simulationsdaten

Der dritte Bereich zeigt sämtliche gesammelten Daten, die mittels Ausführung der Simulation in Omnet++/INET generiert werden. Jeder Testfall, der durch das in Omnet++/INET ausgeführte Tool innerhalb der Pipeline *opp_test* untersucht wird, ist eine eigene Simulation. Der detaillierte Aufbau einer derartigen Simulation wurde bereits im Abschnitt 4.4 beschrieben. Durch das *opp_test* Tool werden alle ausgeführten Tests zu einer Executable zusammengefasst. Dadurch sind selbst bei mehreren Tests sämtliche erzeugten Daten innerhalb des *out* Ordner der Simulationsumgebung. Sollten Tests fehlschlagen, bildet dieser Ordner die Basis für die erforderliche Problemsuche. Die Tests überprüfen mittels Logausgaben im *out* Ordner der Simulation potenzielle Ereignisse dahingehend, ob und wie Fehler behoben werden können. Aus diesem Grund befindet sich eine umfangreiche Dokumentation über den Ablauf der Simulationen für die Tests in diesem Ordner.

Die Simulationen erzeugen mithilfe des *opp_test* Tools beim Ausführen der Tests einen Datensatz. Die Daten selbst können für die Auswertung von Fehlern und Problemen in der Ausführung herangezogen werden. Für das gesteuerte Testverfahren bedeutet dies keinen unmittelbaren Mehrwert, da es kein direkter Bestandteil dieser Arbeit darstellt die Daten der Simulationen zu analysieren. Lediglich werden die Grundzüge verwendet für den Demotestfall dessen Verifikation auf diesen Simulationsdaten aufbaut. Diese Sammlung von Daten spielt lediglich für die einzelnen Entwickler eine Rolle, welche im Rahmen der Problemsuche dadurch entscheidende Kenntnisse erlangen können. Aus diesem Grund werden die enthaltenen Daten an dieser Stelle nicht weiter erläutert, da diese für die Demonstration des gesteuerten Testverfahrens lediglich im Demotestfall verwendet werden. Die Beschreibung über die Verifikation von Tests wurde bereits in Abschnitt 3.5 beschrieben. Dort werden für die Auswertung von Tests die durch die Simulation erzeugten Daten benötigt.

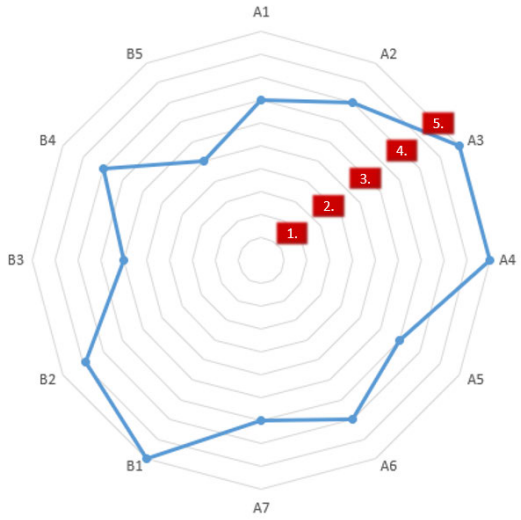

Abbildung 5.4: Eine Übersicht über die Antworten aus der Fragerunde zur Analyse

5.3 Zusatzdaten

Zusätzlich zu den gesammelten Daten aus dem gesteuerten Testverfahren sollen Daten aus einer Umfrage für die Bekräftigung der in dieser Arbeit diskutierten Hypothese beitragen. Aus diesem Grund wurden insgesamt 16 Personen zum Thema Umgang mit einem Feedback wie ein gesteuertes Testverfahren bieten kann befragt. Die in Abschnitt 3.4 aufgelisteten Fragen wurden an ausgewählte Personen gestellt. Der befragte Personenkreis umfasst Entwickler, welche im Umgang mit einem Versionskontrollsystem Erfahrungen aufweisen und dieses zudem regelmäßig nutzen. Das Ziel dieser Umfrage ist, eine Aussage zu generieren, die diskutierte Hypothese wie ein effektiverer Einsatz von Ressourcen und effektiverer Einsatz einer Testmethodik zu untermauern. Die Fragen können in zwei Kategorien unterteilt werden. Die erste Kategorie (A) befasst sich mit der Frage zur Problemlösung innerhalb einer Software welche mit einem Versionskontrollsystem entwickelt wird. Die zweite Kategorie (B) beschäftigt sich mit einer Schätzung zu dem erwarteten Aufwand den die Problemfindung mit und ohne ein, wie in dieser Arbeit beschriebenes gesteuertes Testverfahren in Anspruch nimmt. Jeder Befragte bewertet die Antwort anhand einer Skala von 1 - 5. Dabei bedeutet 1 - *stimme nicht zu* und 5 - *stimme voll zu*. In der Kategorie B beschreibt die Skala von $1 - 5$ den Aufwand, wobei 1 - *wenig Aufwand* und 5 - *viel Aufwand* bedeutet. Um eine allgemeine Aussage zu identifizieren wurden die Antworten über alle Aussagen der Befragten gemittelt und in die Abbildung 5.4 eingetragen. Im Folgenden wird auf die Auswertung der beschriebenen Umfrage eingegangen.

Fragen Kategorie A

Die Kategorie A befasst sich mit Problemen, welche innerhalb einer Software entstehen und eine Lösungsfindung durch einen Entwickler bedürfen. Hier soll insbesondere die Erfahrung der einzelnen Befragten ausschlaggebend für die Meinungsbildung sein. Die Antworten der Kategorie A sind zur Veranschaulichung in Abbildung 5.5 in einem Säulendiagramm dargestellt. Anhand der Säulen ist erkennbar, dass mehr als 50 Prozent (Skala 3.5) der Befragten täglich mit einem Versionskontrollsystem arbeiten. Damit ist sichergestellt, dass die befragten Personen mindestens ein Grundwissen gegebenenfalls sogar weitreichende Erfahrungen in Bezug auf Fehlersuche innerhalb eines Repositories besitzen. Aus diesem Grund bilden die ersten Fragen sowohl den Einstieg in das Thema als auch die Grundlage für die konstruktive Beurteilung der späteren Aussagen. Die Befragten bestätigen zusätzlich die Vermutung, dass ein fehlerhafter Code innerhalb des Repositories schnellstmöglich behoben werden sollte (Skala 4). Dazu gehören nicht nur die fehlschlagenden Tests sondern auch ein fehlerhafter Programmcode welcher innerhalb der Software entstanden ist. Weiterhin haben das ausnahmslos alle Befragten die beiden folgenden Fragen mit dem höchsten Wert 5 bewertet: Die erste Frage bezieht sich auf die Situation in der mehrere Entwickler zusammen an einer Software arbeiten. An dieser Stelle sollten die Befragten einschätzen in wie weit Maßnahmen ergriffen werden sollen um die Arbeiten der einzelnen Entwickler zu koordinieren. *Problemfindung dauert lange* lautet die zweite Aussage, anhand derer überprüfen werden soll wie die Erfahrungen der Entwickler zum Thema Zeitaufwand beim Finden von Problemen ausgeprägt ist.

Folglich sind alle befragten Anwender der gleichen Meinung: Projekte mit einer großen Anzahl von Entwicklern, die an einer Software arbeiten, benötigen eine spezielle Koordination für die Handhabung des Umgangs mit dem Versionskontrollsystem. Damit ist insbesondere gemeint, dass nicht alle Entwickler direkt ihre Änderungen auf den Master Branch pushen können sondern vorher nach bestimmten Überprüfungen Änderungen einpflegen. Ebenfalls sind die befragten Probanden sich darin einig, dass bei Problemen in einer Software, die Problemfindung von fehlerhaftem Programmcode den höchsten Zeitfaktor einnimmt. Allerdings ist diese Aussage in Relation zu sehen, denn lediglich knapp über 50 Prozent der Befragten (Skala 3.5) haben bereits Probleme lösen müssen, bei denen nicht von Beginn an problematische Änderung im Programmcode bekannt gewesen ist. Dies hat zur Folge, dass die Antworten dieser Entwickler nicht nur auf Erfahrungen basieren, sondern zum Teil persönlichen Schätzungen unterliegen.

Indes sind die beiden letzten Fragen der Kategorie A für diese Arbeit am wesentlichs-

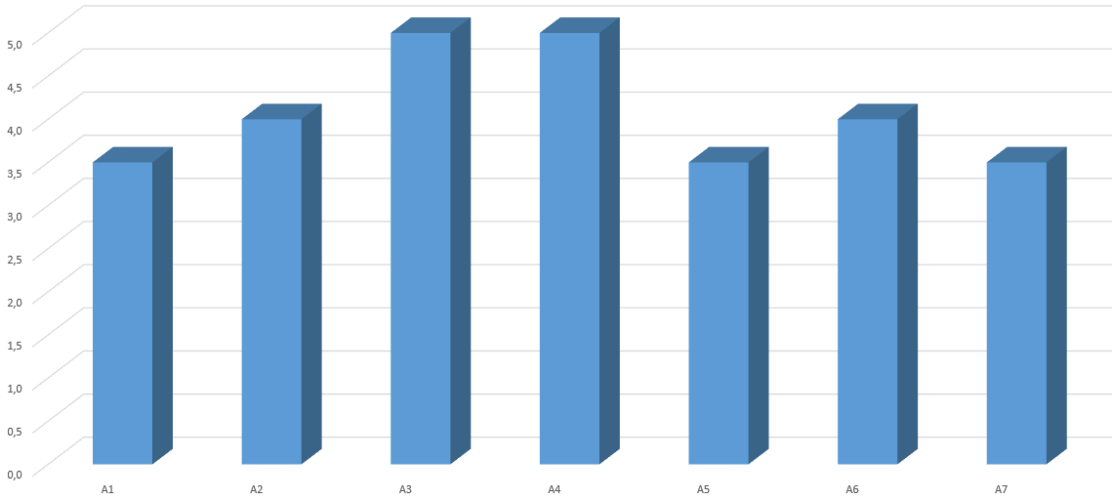

Abbildung 5.5: Die Auswertung der Antworten aus Kategorie A

ten in Bezug auf die herangeführte Hypothese. Diese Fragen zielen auf die Erfahrungen der jeweiligen Entwickler ab im Hinblick darauf, inwiefern der Einsatz eines gesteuerten Testverfahrens sinnvoll und gewinnbringend in ihrem individuellen Arbeitsalltag ist. Die erste Frage lautet: *Sie wären mit dem Feedback eines gesteuerten Testverfahrens schneller gewesen.* Die durchschnittliche Antwort liegt auf der Skala bei 4. Dieses Ergebnis ist vermutlich darauf zurückzuführen, dass nicht jeder der Befragten einen Fehler über eine große Anzahl an Änderungen hinweg untersuchen musste. Deuten lässt sich dies aus der zuvor erwähnten Frage bei der knapp 50 Prozent der Entwickler mit einer persönlichen Schätzung geantwortet haben. Dennoch kann aus der verhältnismäßig hohen Zustimmung von 80 Prozent auf die Frage eine Mutmaßung aufgestellt werden: Die Befragten empfinden ein gesteuertes Testverfahren im Zusammenhang mit unmittelbarem Feedback auf die getätigten Änderungen als Arbeitserleichterung.

Die letzte Frage dieser Kategorie richtet sich an die Motivation eine temporäre zusätzliche Leistung zu erbringen um für ihr Projekt ein gesteuertes Testverfahren einzusetzen. In diesem Fall ist deutlich geworden, dass die befragten Anwender nicht unbedingt bereit sind ein System in jedem Projekt einzusetzen. Möglicherweise liegt dieser Umstand daran, dass sich durchaus nicht jedes Projekt hierfür eignet, da es bei kleineren Projekten meisten einen zu großen Mehraufwand bedeutet. Die Vermutung basiert darauf, dass die Entwickler bei dieser Frage nicht ausschließlich größere Projekte im Sinn hatten und vielmehr im Allgemeinen basierend auf ihrer Erfahrung oder aus dem Bauchgefühl geantwortet haben. Festzuhalten bleibt, dass jeder der Entwickler durchaus einen Vorteil

und eine Arbeitserleichterung durch ein gesteuertes Testverfahren erkennen kann oder auch zu schätzen weiß.

Fragen Kategorie B

Die zweite Fragenkategorie für die Umfrage fokussiert sich gänzlich auf das Thema Aufwandsschätzung in Bezug auf bestimmte Aufgaben und deren Fertigstellung. Dabei handelt es sich erneut um den Bereich der Fehlersuche beziehungsweise die erhoffte Hilfe bei der Fehlersuche im Softwareprojekt. Hier sollen lediglich Schätzungen basierend auf Erfahrung und Bauchgefühl getroffen werden.

In der ersten zu bewertenden Frage sollten die Befragten anhand der bekannten Skalierung angeben, wie hoch jeder Einzelne den Aufwand einer Fehlersuche im einem Projekt einschätzt. Die soll unter der Bedingung, dass kein Hilfsmittel zur Verfügung steht erfolgen. Als Hilfsmittel ist in diesem Fall ein Feedback durch ein System wie Jenkins gemeint. Im Ergebnis haben alle Teilnehmer den Aufwand als sehr hoch empfunden, der Mittelwert beträgt eine 5 und damit den Höchstwert auf der Skala.

Dabei beschreibt die Suche nach dem Fehler, insbesondere die Suche des Teil des Programmcodes indem sich der Fehler, befindet den meisten Anteil des Zeitaufwandes. Belegt wird dies durch die Antworten der Frage zwei (B2 - 4.5) welche die Zeitersparnis schätzt, die erreicht werden kann, wenn die Ursache eines Fehlers bekannt ist. Mit dem Feedback eines gesteuerten Testverfahrens, wie in dieser Arbeit vorgestellt, reduziert sich die Schätzung des Aufwandes zur Fehlersuche auf die Hälfte (B3 - 3). Einer der Gründe für diese Ersparnis ist, dass die Fehlerquelle in der Regel deutlich früher und genauer erkannt werden kann. Dafür bieten die Daten aus dem Testverfahren einen detaillierteren Einstieg in die Fehlersuche.

Der zweite Teil der Fragen zielt darauf ab, wie hoch die Einflussnahme eines gesteuerten Testverfahrens innerhalb eines Teams ist. Im Wesentlich soll anhand dessen untersucht werden, inwieweit ein gesteuertes Testsystem das Team durch die zusätzlichen Arbeiten mit der Administration und dem Aufbau belastet. Dabei ist nicht die Teamgröße entscheidend. Es soll lediglich der zu erwartende Mehraufwand durch das Aufbauen eines gesteuerten Testverfahrens für ein Projekt darstellen. In vielen großen Projekten werden aus diesem Grund sogar einzelne Teams für diese Arbeit gegründet. Diese Teams beschäftigen sich mit den Aufgaben ein autonomes System zu betreiben und bearbeiten dort alle auftauchenden Probleme die solche ein Testverfahren mitbringen kann. Diese Tätigkeiten

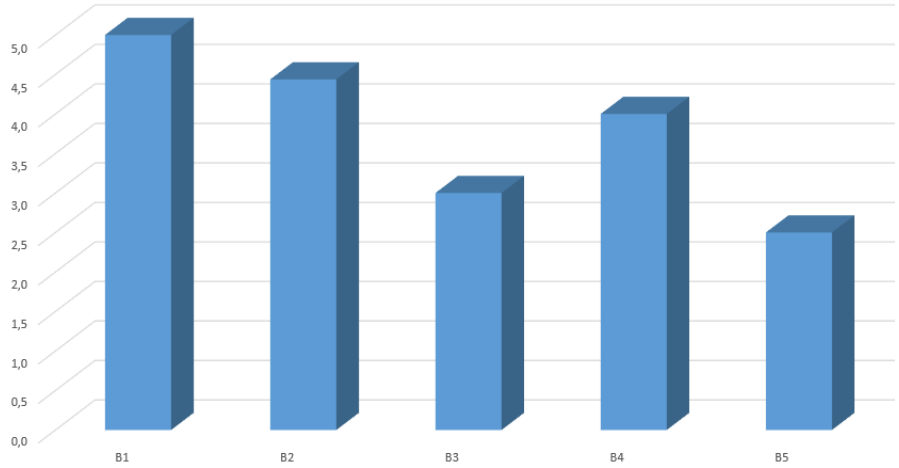

Abbildung 5.6: Die Auswertung der Antworten aus Kategorie B

fallen in den Bereich des bereits erwähnten Mindsets genannt Devops. Aus diesem Grund richten sich die beiden letzten Fragen auf diesen Bereich und sollen untersuchen wie die zusätzliche Arbeit durch den Einsatz eines solchen gesteuerten Testverfahren durch die Teams eingeschätzt wird. Die Frage wurde von Entwicklern, die in vergleichsweise kleinen Teams arbeiten mit einem deutlich höheren Aufwand eingeschätzt, nämlich (B4 - 3.5). Im Vergleich dazu haben Entwickler den zusätzlichen Aufwand für größere Teams auf (B5 - 2.5) bewertet. Als Grund für die geringere Aufwandsschätzung im Falle von vergleichsweise größeren Teams lässt sich der Nutzen durch ein solches Testverfahren aufführen. Wo bei kleinen Teams verhältnismäßig viel Arbeitsleistung in die Entwicklung und Wartung eines solchen Testverfahrens gesteckt wird, reduziert sich dieses Verhältnis bei großen Teams.

Somit untermauern die Antworten die Hypothese das mittels eines gesteuerten Testverfahrens der Aufwand für das Finden und Identifizieren von Fehlern reduziert werden kann. Zusätzlich schätzen alle Entwickler den direkten Aufwand für das Finden von Problemen als verhältnismäßig hoch ein, sodass in diesem Fall das gesteuerte Testverfahren unterstützend ansetzen kann.

5.4 Einordnung

Das Ziel dieses Kapitels ist die zuvor beschriebenen drei Bereiche zusammenzufassen, einzuordnen und einer Bewertung zu unterziehen. Die Daten aus dem gesteuerten Test-

verfahren sind über mehrere Wege einsehbar, jedoch wird zur Einsicht der Zugang zum Jenkins Server benötigt. Das gilt für die Daten der Visualisierung, der einzelnen Pipelinedurchläufe sowie für die Daten der ausgeführten Simulationen. Somit können sämtliche Daten, die gesammelt werden, über diesen einen Zugang zum Jenkins Server eingesehen, ausgewertet und zum Teil heruntergeladen werden. Alle drei Bereiche erheben unterschiedliche Daten und dennoch fügt lediglich die Kombination einen Mehrwert für die Entwicklung durch des gesteuerten Testverfahren zusammen. Ein Beispiel dafür ist, dass einer dieser Bereiche fehlt eine ausgelassene Visualisierung. Diese zeigt in der Regel mittels Eintragungen den Zustand der Software an. Fehlt indes diese Visualisierung, kann der einzelne Entwickler keinerlei Informationen über den Zustand einer Pipeline bekommen ohne die ausführlichen Logausgaben der Pipeline zu analysieren. Dieser Umstand hat zur Folge, dass jede einzelne Pipeline kontrolliert und die durch den Durchlauf generierten Daten untersucht werden müssen. Ausschließlich dadurch erhält der Entwickler eine Rückmeldung über den Erfolg oder Misserfolg seiner Änderungen in der Software durch die Pipeline.

Die Hypothese, dass das hier entwickelte Testverfahren einen Mehrwert für die Entwicklung von Software und deren Tests erbringen soll, wird im Folgenden anhand der verfügbaren Daten bewertet. Ein Mehrwert in diesem Zusammenhand beschreibt einen effektiveren Einsatz von Ressourcen und der effektive Einsatz einer Testmethodik. Dies beinhaltet eine Auflistung der Daten, die dem Entwickler zur Verfügung stehen, um seine Änderungen zu kontrollieren.

Zunächst ist eine simple Betrachtung der Visualisierung für den jeweiligen Entwickler ausreichend, um auf den ersten Blick zu erkennen, dass beispielsweise eine Pipeline fehlgeschlagen ist. Bereits bei fehlschlagenden Pipelines kann dieses Vorgehen einen immensen Zeitgewinn haben. Dabei ist die Visualisierung eine geeignete Möglichkeit den unmittelbaren Zustand abzubilden. Anhand dessen können durch die Entwickler Schlüsse auf das System zugelassen werden. Einen weiteren Vorteil schafft ein detaillierter Einblick in die Zustände der Visualisierung einzelner Pipelinedurchläufe. Anhand der Pipelinedaten werden bereits einzelne Bereiche eines Durchgangs analysiert und mittels visueller Darstellung angezeigt. Hier bietet sich für die Entwickler bereits ein erster Anhaltspunkt für einen möglichen problematischen Programmcode infolge der Darstellung einzelner Arbeitsschritte einer Pipeline und deren Erfolg oder Fehlschlag. Ist die Übersicht der Pipelinedaten bzw. die Logausgaben des zugehörigen Pipeline Durchlaufs unzureichend zu den bestehenden Problemen ist ein Blick in die Simulationsdaten zwingend notwendig. Innerhalb dieser Daten befinden sich sämtliche durch die Omnet++ Simulation generierten

Daten, die einen Aufschluss über das Problem geben können, da sie die direkte Ausgabe der Implementierung beisteuern. Die generierten Daten der einzelnen Simulationen sind abhängig von den definierten Testfällen. Da in dieser Arbeit eine Demonstration des gesteuerten Testverfahrens analysiert wird brauchen diese Daten nicht näher analysiert werden. Diese Daten sind in der Regel für die Entwickler notwendig, sofern Probleme auf die Software zurückzuführen sind, welche mit dem gesteuerten Testverfahren überprüft wird.

Das Ziel besteht darin ein gesteuertes Testverfahren zu erstellen, welches Softwareentwickler in ihrer Tätigkeit unterstützt indem der Einsatz von Ressourcen effektiver genutzt werden kann. Zusammen mit diesem Testverfahren und den in Abschnitt 4 beschriebenen Arbeitsprozessen kann die Arbeit eines Entwicklers beschleunigt werden.

Die entscheidendste Erkenntnis im Zusammenhang mit diesem Testverfahren liegt in der Tatsache, dass die Entwickler ein schnelleres Feedback über den Zustand ihrer Implementierung erhalten. Als Beispiel dafür dient das folgende Szenario: Die Entwicklung einer Software kann durch die simple Visualisierung von erfolgreichen Pipeline Durchgängen beschleunigt werden. Es bleibt festzuhalten, dass dadurch eine weitere Untersuchung des Problems nicht ausbleibt im Fall eines angezeigten Fehlers. Dennoch ist ein Entwickler nicht länger zeitlich blockiert den vollständigen Testdurchlauf auf seinem Arbeitsrechner durchzuführen. Mit dem im Abschnitt 4 beschriebenen Szenario wird die Arbeit eines Entwicklers dargestellt.

Bei einer erfolgreichen Pipeline kann der Entwickler beruhigt seine nächsten Arbeitsaufgaben angehen. Bei einem Fehler nutzt der Entwickler die drei Kategorien der Visualisierung, Pipelinedaten und Simulationsdaten. Beginnend mit der ersten Visualisierung um das Problem zu untersuchen und den zugehörigen Arbeitsschritt zu identifizieren. Anschließend werden die Pipelinedaten betrachtet um das Problem weiter einzugrenzen. Um die direkte Problemstelle auszumachen können aus den Simulationsdaten exakte Details zur Software und deren Ausführung erhalten werden.

Anhand des Szenarios sind in direkter Gegenüberstellung mit der Arbeit eines Entwicklers ohne ein Testverfahren die Vorteile des Testsystems sichtbar. Für die Visualisierung mit einem anschaulichen Szenario wurde ein Master Branch gewählt. Von dem Master Branch zweigt sich ein Entwickler einen Branch ab und tätigt auf diesem seine Entwicklungen und Erweiterungen sowie die Behebung von Bugs. Von diesem Branch merged der Entwickler in regelmäßigen Abständen seine Entwicklung zurück auf den Master

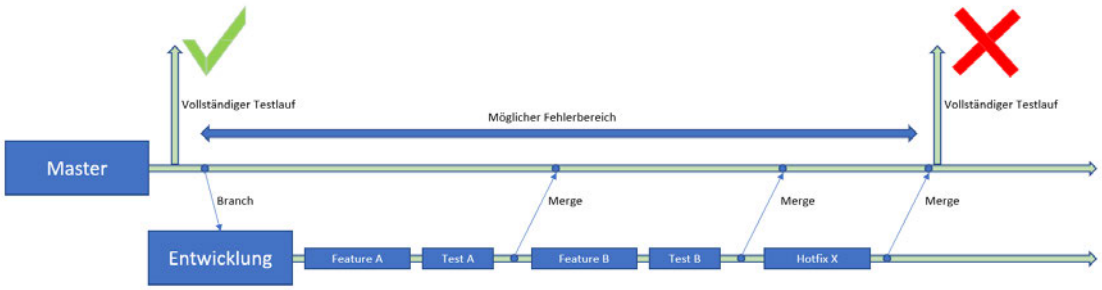

Abbildung 5.7: Ein beispielhafter Arbeitsprozess eines Entwicklers bei dem die Suche nach dem Fehlschlagen von bestehenden Test erschwert wird.

Branch. Dieses Zusammenführen geschieht nach der Fertigstellung und nach dem Durchführen der jeweiligen notwendigen Tests für diese Erweiterungen. Es ist hinreichend bekannt, dass der einzelne Entwickler zunächst die Tests ausführt um seine Erweiterung zu überprüfen bevor diese in den Master Branch zurückgeführt werden. Die Abbildung 5.7 visualisiert den zuvor beschriebenen Vorgang anhand eines zeitlichen Ablaufs. Da das Ausführen aller Tests für die komplette Software Zeit in Anspruch nimmt und folglich den Arbeitsbereich des Entwicklers in diesem Zeitabschnitt blockiert, werden während der Entwicklung ausschließlich jene Tests für die neu entwickelte Erweiterung ausgeführt. Am Ende eines Tages wird in diesem Szenario ein einmaliger Testlauf getätigt um das entstehende Problem gut darstellen zu können. Bei diesem Komplettdurchlauf fällt auf, dass innerhalb der Software einige bestehende Tests fehlschlagen. Ein potenzielles Problem besteht darin die ausschlaggebende Änderung zu finden, aufgrund dessen die für das Fehlschlagen der Tests verantwortlich sind. Folglich sind sämtliche Änderungen im Master Branch zu durchsuchen um die entsprechende Zusammenführung mit der problematischen Änderung zu finden und nachzuvollziehen. Diese Fehlersuche ist notwendig um den fehlerhaften Programmcode zu identifizieren und zu beheben.

Existiert ein wie in dieser Arbeit vorgestelltes Testverfahren, ändert sich die Tätigkeit des einzelnen Entwicklers zunächst nicht unmittelbar. Dieser arbeitet weiterhin auf einer Kopie des Master Branch und entwickelt darin die Erweiterungen inklusive der zugehörigen Tests. Ein Vorteil eines solchen Testverfahrens ergibt sich erst, sobald der Entwickler einen ersten Teil seiner Aufgaben erledigt hat. Das bedeutet er führt diese Erweiterung mit dem Master Branch zusammen. Der Entwickler hat die Möglichkeit - wie zuvor - ebenfalls unmittelbar weiter zu arbeiten und eine neue Erweiterung bzw. einen Hotfix zu implementieren. Das in dieser Arbeit vorgestellte Testverfahren fungiert gänzlich im Hin-

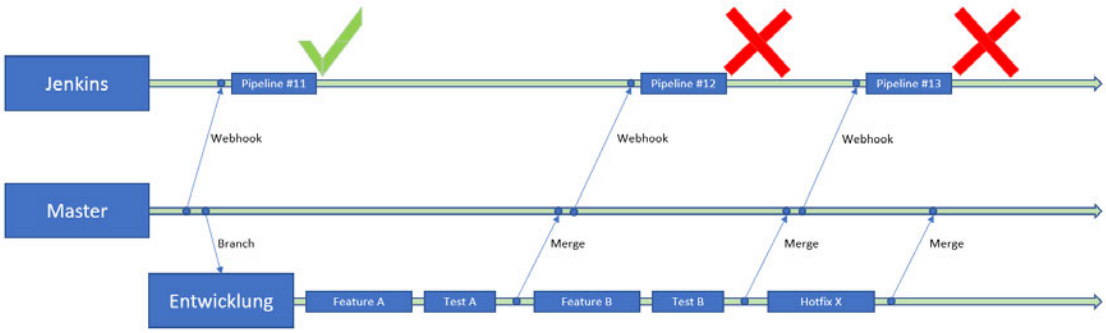

Abbildung 5.8: Ein beispielhafter Arbeitsprozess bei dem ein Entwickler direkt auf das Fehlschlagen von Tests durch ein Fehlschlagen der Pipeline hingewiesen wird.

tergrund und startet automatisch bei einem Merge oder Push auf dem Master Branch. Das Testverfahren beschreibt das Zusammentragen der einzelnen Daten, die für die Simulationsumgebung und Tests benötigt werden sowie die letztendliche Ausführung aller im Master Branch vorhandenen Tests. Das Ergebnis der Pipeline ermöglicht dem Entwickler den genauen Programmcode im Repository zu zeigen, bei dem eine Änderung eingepflegt wurde, welcher die bestehende Tests fehlschlagen lässt. Dabei werden sowohl die zugehörige Änderung im Master Branch bekannt als auch die fehlschlagenden Testfälle im Log angezeigt. Die Abbildung 5.8 zeigt wie schon in der Abbildung 5.7 die Aufgaben eines Entwicklers. Der Unterschied besteht in dem unmittelbaren Feedback. Durch das gesteuerte Testverfahren, welches durch den Webhook beim Zusammenführen mit den Master Branch gestartet wird.

Unter Betrachtung eines größeren Repositories, in dem 50 - 100 Entwickler an der Software entwickeln, wird deutlich, dass die Arbeitsprozesse nicht immer nachvollziehbar bleiben und die Fehlersuche in der Regel über unterschiedliche Änderungen hinweg betrieben werden muss. Bei einer derartigen Größe eines Repositories wird der Nutzen eines solchen Testverfahrens erst konkret sichtbar aufgrund des unmittelbaren Feedback über Probleme in der Codebasis. Zusammen mit den Logdateien der Pipeline kann die Fehlersuche in diesem Sinne effizient eingegrenzt werden. Zusätzlich spiegelt die Pipeline den aktuellen Teststand des betrachteten Branches wieder, da über erfolgreiche bzw. fehlgeschlagene Testdurchläufe direkt und visuell Feedback gegeben werden kann.

Grundlage für eine Beurteilung bildet die Annahme, wie bereits in Abschnitt 4.3 vorgestellt, dass die Arbeit eine Entwicklers unterstützt werden kann. Anhand der Abbildungen

5.8 und 5.7 kann visuell erläutert werden durch welche Bereiche die Entwickler Feedback erhalten und eine Ersparnis der Arbeitszeit haben. In den Abbildungen wird beispielhaft ein Arbeitstag des Entwicklers betrachtet, dort ergibt sich mithin ein potenzieller Zeitgewinn von bis zu zwei Stunden. Diese Berechnung setzt sich wie folgt zusammen: Die Zeit, welche für das Implementieren des ersten Features und die dafür benötigten Tests benötigt wird sind zeitlich fix. Sobald das erste Feature mit dem Master zusammengeführt wurde, kann der Entwickler direkt mit der Implementierung für das zweite Feature und den zugehörigen Tests beginnen. Das Überprüfen des ersten Features zusammen mit dem Ausführen aller vorhandenen Tests übernimmt dabei das Testverfahren. Betrachtet der Entwickler das Testverfahren und erkennt eine fehlgeschlagene Pipeline bekommt er unmittelbar weitere Informationen durch den Jenkins Server bereitgestellt. Zusätzlich erhält dieser die Information welche Tests fehlschlagen und kann damit eine Eingrenzung für die Fehlersuche durchführen. Die Logausgaben der Pipeline und die Logdateien der Simulationen des Testdurchlaufs ermöglichen einen Einblick in die fehlgeschlagenen Tests. Dadurch ermittelt der jeweilige Entwickler letztendlich die Änderung, welche die bestehenden Tests fehlschlagen lässt.

Eine erste Möglichkeit für die Behebung der fehlschlagenden Tests wäre, dass der Entwickler das zweite Feature zurück in den Master führt und danach mit einem Hotfix für die fehlgeschlagene Pipeline beginnt. Die zweite Variante wäre ein direkter Hotfix für die fehlgeschlagenen Tests. Um die bessere Identifizierung zu einzelnen Änderungen im Programmcode zu verdeutlichen entscheidet sich der Entwickler für die erste Möglichkeit. Da der Entwickler zuerst das zweiten Features mit dem Master Branch zusammenführt wird die Pipeline erneut fehlschlagen. Unter Umständen können dadurch sogar weitere Probleme entstehen. Erst durch den Hotfix, welchen der Entwickler wieder auf den Master zusammenführt um die Probleme zu beheben sollte die Pipeline wieder erfolgreich durchlaufen.

Eine etwas aufwendigere Möglichkeit für den Entwickler ist, dass dieser auf einem temporären Branch das Hotfix für das erste Feature implementiert und diese zurück in den Master führt. Das hat zur Folge, dass die Pipeline erneut erfolgreich durchlaufen wird bevor die Änderungen für das zweite Feature auf den Master geführt werden. Die Abbildung 5.9 zeigt die Gegenüberstellung von der Entwicklung mit und ohne eine Rückmeldung durch ein Testverfahren. Mit dem Testverfahren und dem Feedback einer solchen Pipeline lässt sich eine geschätzte Zeitersparnis von bis zu 1/4 eines Arbeitstages des Entwicklers berechnen. Diese Schätzung folgt aus der Reduzierung der Arbeitsleistung und der direkten Präsentation von Daten aus einem Durchlauf einer Pipeline. Diese Daten können

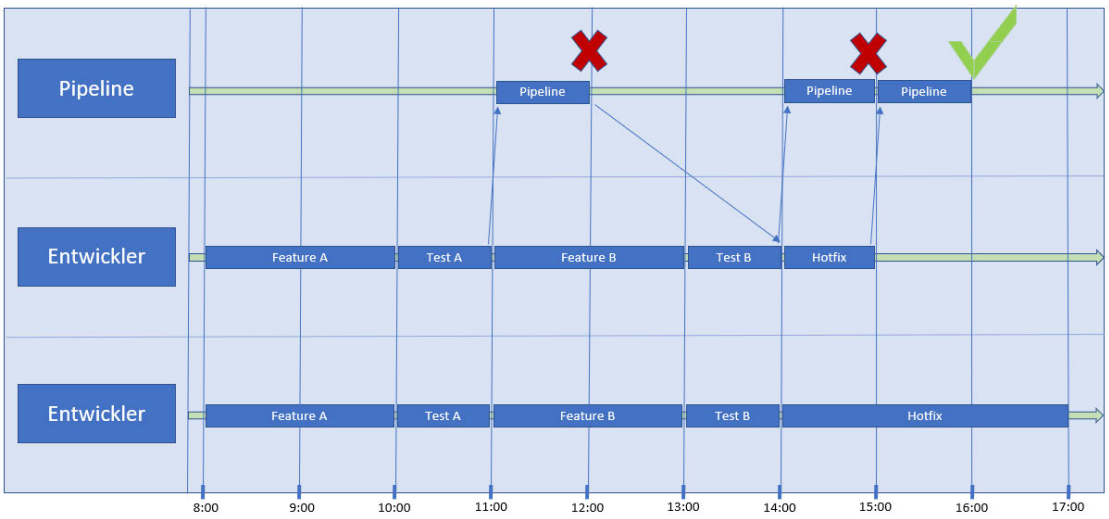

Abbildung 5.9: Ein beispielhafter Ablauf einer Entwicklung mit und ohne das Feedback eines Testverfahren.

auch immer zu einem direkten Versionsstand innerhalb des Repositories zurückverfolgt werden.

6 Ausblick & Schluss

In diesem Kapitel werden abschließend die in dieser Arbeit gesammelten Ergebnisse strukturiert und kritisch hinterfragt. Dabei steht die Frage im Vordergrund welche Auswirkung das vorgestellte gesteuerte Testverfahren auf die Methodik hat um letztendlich einen effektiven Einsatz der Ressourcen für die Entwicklung einer Software zu leisten. Prinzipiell soll der effiziente Einsatz von Ressourcen im Zusammenhang mit dem Testverfahren betrachtet werden. Als Teil der kritischen Betrachtung soll das Testsystem untersucht und eventuell problematische Bereiche hervorgehoben werden. Um die Forschungsfrage zu beantworten, wird das Ergebnis des Testverfahrens kritisch analysiert. Schließlich rundet ein Ausblick für weitere bzw. zukünftige Forschungen und Erweiterungen für ein gesteuertes Testverfahren das Kapitel ab.

6.1 Ergebniszusammenfassung

Die Untersuchung der Daten hat gezeigt, dass durch dieses gesteuerte Testverfahren keine neue Daten generiert werden können. Die Daten werden übersichtlicher und strukturierter dargestellt. Die Anzahl der unterschiedlichen Daten bezieht sich auf die Bereiche: Visualisierung, Pipelinedaten und Simulationsdaten. Damit sind die in Abschnitt 5.2 vorgestellten und diskutierten Daten gemeint. Die Visualisierung ermöglicht dabei, dass sichtbare Elemente in Jenkins dem einzelnen Entwickler den Zustand des Testdurchgangs beschreiben. Somit ist für die Nutzer jederzeit eine Statusabfrage zu den eigenen Aufgaben einzusehen. Voraussetzung hierfür ist lediglich ein Zugang zu Jenkins, welches die Daten sammelt und speichert.

Es ist nicht abzustreiten, dass jeder Entwickler die gleichen Informationen erhält, indem ein vollständiger Testdurchgang auf seinem Arbeitsrechner durchgeführt und die ausgegebenen Logdateien überprüft. Aus diesem Grund stellt sich die Frage, welchen Nutzen das vorgestellte Verfahren mit sich bringt. Letztendlich stellt das System Jenkins eine Strukturierung und Zuordnung der Daten zur Verfügung. Weiterhin werden die Logausgaben

in der zweiten Kategorie, die Pipelinedaten, mittels Jenkins an einer zentralen Stelle gesammelt und präsentiert. Die Änderungen an der Software können dabei unmittelbar einem Stand im Repository zugeordnet werden. Somit übernimmt Jenkins die Arbeitsleistung von den Entwicklern den Testdurchlauf auszuführen und präsentiert zusätzlich die Daten in den drei Bereichen (Visualisierung, Pipelinedaten und Simulationsdaten) in strukturierter und übersichtlicher Form. Somit automatisiert Jenkins die zuvor beschriebenen Tätigkeiten, die zuvor von dem einzelnen Entwickler selbständig auszuführen ist und stellt die generierten Informationen zur Verfügung.

Die in dieser Arbeit zusammengestellten Daten sind für den Entwickler sehr hilfreich, da ein Feedback über den aktuellen Status bereitgestellt wird. Dieses gesteuerte Testverfahren bietet den Entwicklern somit einen alternativen Weg um an Informationen über den Erfolg oder Fehlschlag der Tests zu gelangen. Eine effizientere Arbeitsweise ergibt sich hingegen, da die Entwickler weiter arbeiten können und den Durchlauf der Tests nicht mehr selbst ausführen müssen. Zusätzlich erhalten sie die Informationen über Probleme in der Software direkt nach Beendigung der Pipeline. Somit wird die einleitende Fragestellung dieser Arbeit belegt und ein effektiverer Einsatz der Ressourcen erreicht. Diese Ressourcen beschreiben sowohl eine Reduzierung eines benötigten Zeitaufwandes als auch einen monetären Gewinn der unter anderem durch die Reduzierung der Arbeitszeit der Entwickler erreicht wird.

Durch die Möglichkeit mit diesem Testverfahren jegliche Änderung an einer Codebasis direkt zu überprüfen und damit genau auf getätigte Änderungen in Falle eines Fehlers zu verweisen wird die Testmethodik effizient eingesetzt. Somit zeigt dieses gesteuerte Testverfahren mit dem Beispiel des QUIC Handshakes das die Ressourcen und Testmetodik effektiv eingesetzt werden kann. Dabei begleitet die Testmethodik die vorhandene Teststruktur der Omnet++ Simulationsumgebung, welche als Basis für dieses Testverfahren gewählt wurde.

6.2 Erweiterungen

In diesem Abschnitt werden Erweiterungen und Änderungen vorgestellt mit denen das gesteuerte Testverfahren verbessert werden kann. Zur Übersicht über mögliche Erweiterungen dient die folgende Aufstellung. Die Erläuterung und der geschätzte Einfluss auf das System wird im Folgenden erklärt.

- Variable Testfälle
- Erweiterung des Docker Images mit der kompilierten Software
- Externe Visualisierung
- Weitere Überprüfungen
- Begrenzung auf Merge Requests als Pipeline Trigger

Variable Testfälle

Die Ausführung eines Durchlaufs des gesteuerten Testverfahrens wurde explizit für einen Demotestfall erstellt. Eine notwendige Erweiterung für dieses gesteuerte Testverfahren ist in diesem Fall eine Erweiterung des Skriptes der Pipeline, sodass diese nicht nur den Demotestfall in die laufende Pipeline kopiert sondern auch alle zusätzlich verfügbaren Testdateien. Diese könnte auch variabel gestaltet werden in dem hier ein Parameter eingesetzt wird, der ein Regex enthält und mit dem bestimmte Kriterien für die Auswahl spezieller Tests eingesetzt werden kann. Somit könnte nach bestimmten Schlüsselworten in den Dateinamen der Tests gefiltert werden um nur diese Tests auszuführen.

Erweiterung des Docker Images mit der compilierten Software

Eine weitere sinnvolle Erweiterung des Testverfahrens wäre die Erweiterung des Docker Images um eine bereits fertig gepackte Omnet++ Software. Dies würde zwar keine funktionale Änderung bewirken, allerdings würde dadurch die Durchlaufzeit einer Pipeline stark verringert. Der Grund hierfür ist, dass aktuell das Kompilieren der Software der größte Posten in der benötigten Zeit eines Durchlaufs der Pipeline belegt. Dabei muss sichergestellt werden, das die Zugriffe des Jenkins Benutzer in der Pipeline verwendet wird und dort alle nötigen Rechte besitzt um die Bibliotheken und Programme auszuführen. Zusätzlich müssen die korrekten PATH-Variablen eingetragen sein.

Externe Visualisierung

Eine externe Visualisierung der einzelnen Pipeline Durchläufe dient der Übersicht und verhindert, dass sich ein Entwickler in Jenkins einloggen muss um die aktuellen Zustände und Durchläufe zu überprüfen. Jenkins stellt dafür sowohl Plugins als auch eine mögliche API zur Verfügung um Informationen auszulesen. Als eine Alternative kann das Pipeline Skript erweitert werden um Statusinformationen zu exportieren. Zum einen können Kommentare und Statusmeldungen an GitHub gesendet werden. Beispielsweise kann Jenkins Kommentare innerhalb eines auf GitHub erstellten Merge Request posten und über Erfolg oder Misserfolg einer Pipeline zu berichten. Zum anderen kann eine Datenbank bestimmt werden, welche die Daten für ein übersichtliches Dashboard bereitstellt und speichert.

Weitere Überprüfungen

Die Pipeline führt aktuell lediglich einen Demotestfall aus. Dieser überprüft zusätzlich zum QUIC Handshake die erfolgreiche Ausführung der Simulation. Es können jedoch weitere Schritte zur Überprüfung des Programmcodes herangezogen werden. Das betrifft unter anderem statische Überprüfungen mit Tools wie ClangFormat, Bauhaus. Zugleich können bestimmte Artefakte extrahiert und archiviert werden. Die Möglichkeiten sind weitreichend, was die Überprüfung der Funktionalität der Software anbelangt.

Begrenzung auf Merge Requests als Pipeline Trigger

Weiterhin gehört zu den nichtfunktionalen Erweiterungen das Verwenden des Sourcecode Managements. Hier können verschiedene Verhalten konfiguriert werden. Aktuell reagiert der Git Trigger (Webhook) auf jedes Ereignis des Repositories. Die meisten Ereignisse sind dabei das Pushen von einzelnen Änderungen zurück auf den Master Branch. Eine Möglichkeit diese zu bündeln und zu überprüfen sind sogenannte Merge Requests. Diese fassen sämtliche Änderungen auf einem separaten Branch zusammen auf dem die Entwickler arbeiten. Mit Abschluss der Änderungen werden diese in Form eines Merge Requests an den Masterbranch gestellt. Die Pipeline kann konfiguriert werden, dass ausschließlich beim Erstellen eines Merge Requests die Pipeline startet. Somit sind Änderungen an einem Repository deutlich strukturierter und im Verhältnis einfacher nach-

zuvollziehen. Zugleich werden nicht viele kleine sondern größere, funktional gruppierte Änderungen überprüft.

Literaturverzeichnis

[1] : *Quick UDP Internet Connections*. 2013. – Online erhältlich unter https://www.portainer.io/; abgerufen am 1. Oktober 2018.

[2] Abrahams, Beman Dawes D.: *Boost provides free peer-reviewed portable C++ source libraries*. Website. – Online erhältlich unter https://www.boost.org/; abgerufen am 11. Juni 2020.

[3] Atzori, Luigi ; Iera, Antonio ; Morabito, Giacomo: The internet of things: A survey. In: *Computer networks* 54 (2010), Nr. 15, S. 2787–2805

[4] axivion: *axivion stopping software erosion*. Website. – Online erhältlich unter https://www.axivion.com/de; abgerufen am 11. Juni 2020.

[5] Balaji, S ; Murugaiyan, M S.: Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. In: *International Journal of Information Technology and Business Management* 2 (2012), Nr. 1, S. 26–30

[6] Bamboo, Atlassian: *Erstellen, testen, deployen*. Website. – Online erhältlich unter https://www.atlassian.com/de/software/bamboo; abgerufen am 11. Juni 2020.

[7] Bass, Len ; Weber, Ingo ; Zhu, Liming: *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015

[8] Bates, Adam ; Pletcher, Joe ; Nichols, Tyler ; Hollembaek, Braden ; Tian, Dave ; Butler, Kevin R. ; Alkhelaifi, Abdulrahman: Securing SSL certificate verification through dynamic linking. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, S. 394–405

[9] Baun, Christian: Protokolle und Schichtenmodelle. In: *Computer Networks/Computernetze*. Springer, 2019, S. 35–44

[10] Baun, Christian ; Baun: *Computernetze kompakt*. Springer, 2012

[11] BITRISE.IO: *Build better mobile applications, faster.* Website. – Online erhältlich unter https://www.bitrise.io/; abgerufen am 11. Juni 2020.

[12] BROMAN, David ; SANDAHL, Kristian ; BAKER, Mohamed A.: The company approach to software engineering project courses. In: *IEEE Transactions on Education* 55 (2012), Nr. 4, S. 445–452

[13] CARDWELL, Neal ; CHENG, Yuchung ; BRAKMO, Lawrence ; MATHIS, Matt ; RAGHAVAN, Barath ; DUKKIPATI, Nandita ; JERRY CHU, Hsiao keng ; TERZIS, Andreas ; HERBERT, Tom: *packetdrill: Scriptable Network Stack Testing, from Sockets to Packets.* Website. 2013. – Online erhältlich unter https://www.usenix.org/conference/atc13/technical-sessions/presentation/cardwell; abgerufen am 13. August 2018.

[14] CI, Circle: *The most powerful, scalable CI/CD for Enterprise teams.* Website. – Online erhältlich unter https://circleci.com/; abgerufen am 11. Juni 2020.

[15] CI, Travis: *The simplest way to test and deploy your projects.* Website. – Online erhältlich unter https://travis-ci.com/; abgerufen am 11. Juni 2020.

[16] COACH, Atlassian A.: *Was ist Scrum?* Website. – Online erhältlich unter https://www.atlassian.com/de/agile/scrum; abgerufen am 27. September 2018.

[17] DAVIS, Tony: What is a WebHook? In: *webhooks. pbworks. com, https://webhooks. pbworks. com/w/page/13385124/FrontPage, retrieved Mar* 28 (2017)

[18] DIERKS, Tim ; ALLEN, Christopher u. a.: *The TLS protocol version 1.0.* 1999

[19] DONNER, Dipl.-Ing. (FH) Stefan Luber / Dipl.-Ing. (FH) A.: *Was ist eine Proxy Firewall?* Website. 2018. – Online erhältlich unter https://www.ip-insider.de/was-ist-eine-proxy-firewall-a-621987/; abgerufen am 24. April 2020.

[20] EBERHARDT, M.: *Tools und Ressourcen die für dieses Thesis erstellt wurden.* Website. – Online erhältlich unter https://github.com/MikkoE/CADS_jenkins_tools; abgerufen am 13. Juni 2020.

[21] EBERHARDT, M: Automatisiertes System zum Ausführen von Tests in einer Pipeline. In: *Grundprojekt im Rahmen des Masterstudiengangs an der HAW Hamburg* (2018)

[22] Elektronik-Kompendium.de: *NAT - Network Address Translation*. Website. 2020. – Online erhältlich unter https://www.elektronik-kompendium.de/sites/net/0812111.htm; abgerufen am 24. April 2020.

[23] Eronen, Pasi ; Zitting, Jukka: An expert system for analyzing firewall rules. In: *Proceedings of the 6th Nordic Workshop on Secure IT Systems (NordSec 2001)*, 2001, S. 100–107

[24] Everett, Gerald D. ; McLeod Jr, Raymond: *Software testing: testing across the entire software development life cycle*. John Wiley & Sons, 2007

[25] Fürst, Simon ; Bechter, Markus: AUTOSAR for connected and autonomous vehicles: The AUTOSAR adaptive platform. In: *2016 46th annual IEEE/IFIP international conference on Dependable Systems and Networks Workshop (DSN-W)* IEEE (Veranst.), 2016, S. 215–217

[26] Github smee.io von: *Webhook payload delivery service*. Website. – Online erhältlich unter https://smee.io/; abgerufen am 12. Juni 2020.

[27] github.com: *Build for developers*. https://github.com. – Accessed: 2020-04-28

[28] gitlab: *Simplify your workflows with GitLab*. Website. – Online erhältlich unter https://about.gitlab.com/; abgerufen am 11. Juni 2020.

[29] GmbH, QA S.: *Beschleunigen sie ihre Unit- und Integrationstests*. Website. – Online erhältlich unter https://www.qa-systems.de/tools/cantata/; abgerufen am 11. Juni 2020.

[30] Goldratt, Eliyahu M.: *Theory of constraints*. North River Croton-on-Hudson, 1990

[31] Gujrathi, Siddharth: Heartbleed bug: Anopenssl heartbeat vulnerability. In: *International Journal of Computer Science and Engine ter Science and Engineering* 2 (2014), Nr. 5, S. 61–64

[32] H, König: *Fallbeispiel: Der Internet-Protokollstack*. Vieweg+Teubner Verlag, 2003. – URL https://link.springer.com/chapter/10.1007%2F978-3-322-80066-4_6. – ISBN 978-3-519-00454-7

[33] Hamill, Paul: *Unit test frameworks: tools for high-quality software development*. Ö'Reilly Media, Inc.", 2004

[34] Helander, Tuukka: *Zehn Tipps für effektives URL-Filtering*. Website. 2014. – Online erhältlich unter https://www.security-insider.de/zehn-tipps-fuer-effektives-url-filtering-a-457173/; abgerufen am 24. April 2020.

[35] info, ITWissen: *Versionskontrolle*. https://www.itwissen.info/Versionskontrolle-version-control-VCS.html. 2017. – Accessed: 2020-04-28

[36] Info, ITWissen: *DPI (deep packet inspection)*. Website. 2019. – Online erhältlich unter https://www.itwissen.info/DPI-deep-packet-inspection.html; abgerufen am 24. April 2020.

[37] Info, ITWissen: *SPI (stateful packet inspection)*. Website. 2019. – Online erhältlich unter https://www.itwissen.info/SPI-stateful-packet-inspection.html; abgerufen am 24. April 2020.

[38] Insider, Datacenter: *Virtualisierung und/oder Containerisierung? Argumente aus der Praxis*. Website. – Online erhältlich unter https://www.datacenter-insider.de/virtualisierung-undoder-containerisierung-argumente-aus-der-praxis-a-907816/; abgerufen am 09. Juni 2020.

[39] Iyengar, J. ; Thomson, M.: *QUIC: A UDP-Based Multiplexed and Secure Transport draft-ietf-quic-transport-14*. Website. 2018. – Online erhältlich unter https://datatracker.ietf.org/doc/draft-ietf-quic-transport/?include_text=1; abgerufen am 30. August 2018.

[40] Jenkins: *Jenkins, build great things at any scale*. Website. 2018. – Online erhältlich unter https://jenkins.io/; abgerufen am 10. September 2018.

[41] Kerschagl, Daniel: *Container on Azure – Teil 1: Basics*. Website. – Online erhältlich unter https://whiteduck.de/container-on-azure-teil-1-basics/; abgerufen am 11. Juni 2020.

[42] Kraft, Peter ; Weyert, Andreas: *Network Hacking: Professionelle Angriffs-und Verteidigungstechniken gegen Hacker und Datendiebe*. Franzis Verlag, 2017

[43] Kruse, Jörn: Internet-Überlast, Netzneutralität und Service-Qualität / Diskussionspapier. 2008. – Forschungsbericht

[44] Küveler, Gerd ; Schwoch, Dietrich: Das ISO/OSI—Schichtenmodell der Datenkommunikation. In: *Informatik für Ingenieure*. Springer, 2003, S. 446–451

[45] Laghari, Asif A. ; He, Hui ; Zardari, Shehnila ; Shafiq, Muhammad: Systematic analysis of quality of experience (QoE) frameworks for multimedia services. In: *IJCSNS* 17 (2017), Nr. 5, S. 121

[46] Larisch, Dirk ; Lienemann, Gerhard: *TCP/IP-Grundlagen und Praxis: Protokolle, Routing, Dienste, Sicherheit*. Heise Verlag, 2013

[47] Lienemann, Gerhard ; Larisch, Dirk: *TCP/IP-Grundlagen und Praxis: Protokolle, Routing, Dienste, Sicherheit*. Heise Verlag, 2013

[48] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media, 2009

[49] Lili Qiu ; Varghese, G. ; Suri, S.: Fast firewall implementations for software and hardware-based routers. In: *Proceedings Ninth International Conference on Network Protocols. ICNP 2001*, 2001, S. 241–250

[50] Loeliger, Jon: *Versionskontrolle mit git*. O'Reilly Germany, 2009

[51] Make, Gnu: *GNU Operating System*. Website. – Online erhältlich unter https://www.gnu.org/software/make/; abgerufen am 12. Juni 2020.

[52] Meinel, Christoph ; Sack, Harald: Die Grundlage des Internets: TCP/IP-Referenzmodell. In: *Internetworking* Springer (Veranst.), 2012, S. 31–67

[53] Meinel, Christoph ; Sack, Harald: Die Grundlage des Internets: TCP/IP-Referenzmodell. In: *Internetworking* Springer (Veranst.), 2012, S. 31–67

[54] Meinel, Christoph ; Willems, Christian ; Roschke, Sebastian ; Schnjakin, Maxim: *Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht*. Universitätsverlag Potsdam, 2011 (44)

[55] Merkel, Dirk: Docker: lightweight linux containers for consistent development and deployment. In: *Linux journal* 2014 (2014), Nr. 239, S. 2

[56] Miegler, M. ; Wolz, W.: Development of test programs in a virtual test environment. In: *Proceedings of 14th VLSI Test Symposium*, 1996, S. 99–103

[57] Minnich, Sebastian: *Anonym surfen mit VPN*. Website. 2020. – Online erhältlich unter https://www.heise.de/download/specials/Anonym-surfen-mit-VPN-Die-besten-VPN-Anbieter-im-Vergleich-3798036; abgerufen am 24. April 2020.

[58] Nidhra, Srinivas ; Dondeti, Jagruthi: Black box and white box testing techniques-a literature review. In: *International Journal of Embedded Systems and Applications (IJESA)* 2 (2012), Nr. 2, S. 29–50

[59] NS2/NS3: *ns-3 is a discrete-event network simulator for Internet systems.* Website. 2018. – Online erhältlich unter https://www.nsnam.org/; abgerufen am 10. September 2018.

[60] Omnet++: *OMNeT++ is an extensible, modular, component-based C++ simulation library and framework, primarily for building network simulators.* Website. 2020. – Online erhältlich unter https://omnetpp.org/; abgerufen am 24. April 2020.

[61] omnetpp: *An open-source OMNeT++ model suite for wired, wireless and mobile networks.* Website. – Online erhältlich unter https://inet.omnetpp.org/; abgerufen am 11. Juni 2020.

[62] omnetpp: *Simulation Manual.* Website. – Online erhältlich unter https://doc.omnetpp.org/omnetpp/manual/; abgerufen am 12. Juni 2020.

[63] Pilorget, Lionel: *Testen von Informationssystemen.* Springer, 2012

[64] Pohlmann, Norbert: *Firewall-Systeme.* mitp, 2003

[65] Portainer.io: *MAKING DOCKER MANAGEMENT EASY.* Website. – Online erhältlich unter https://www.atlassian.com/de/agile/scrum; abgerufen am 27. September 2018.

[66] Radivilova, Tamara ; Kirichenko, Lyudmyla ; Ageyev, Dmytro ; Tawalbeh, Maxim ; Bulakh, Vitalii: Decrypting SSL/TLS traffic for hidden threats detection. In: *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)* IEEE (Veranst.), 2018, S. 143–146

[67] Rash, Michael: *Linux Firewalls: Attack Detection and Response with iptables, psad, and fwsnort.* No Starch Press, 2007

[68] Repositories, Inet F. des: *Inet-private Repository.* Website. – Online erhältlich unter https://github.com/Transport-Protocol/inet-private; abgerufen am 13. Juni 2020.

[69] Rouse, Margaret: *Content-Filter (Informationsfilter).* Website. 2016. – Online erhältlich unter http://weidner.in-bad-schmiedeberg.de/computer/netz/firewall/rfcs-fuer-paketfilter/; abgerufen am 24. April 2020.

[70] Scharf, M.: *Quantifying Head-of-Line Blocking in TCP and SCTP*. Website. 2013. – Online erhältlich unter https://tools.ietf.org/id/draft-scharf-tcpm-reordering-00.html; abgerufen am 27. September 2018.

[71] Scheller, Martin ; Boden, Klaus-Peter ; Geenen, Andreas ; Kampermann, Joachim: *Das Internet*. S. 5–32. In: *Internet Werkzeuge und Dienste: Von „Archie“ bis „World Wide Web“*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1994. – URL https://doi.org/10.1007/978-3-642-85137-7_2. – ISBN 978-3-642-85137-7

[72] Schmitz, Dr. Götz Güttich / P.: *Grundlagen der Next Generation Firewall (NGFW)*. https://www.security-insider.de/grundlagen-der-next-generation-firewall-ngfw-a-648098/. 2017. – Accessed: 2020-04-28

[73] Seufert, M. ; Schatz, R. ; Wehner, N. ; Gardlo, B. ; Casas, P.: Is QUIC becoming the New TCP? On the Potential Impact of a New Protocol on Networked Multimedia QoE. In: *2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX)*, 2019, S. 1–6

[74] Soni, M.: End to End Automation on Cloud with Build Pipeline: The Case for DevOps in Insurance Industry, Continuous Integration, Continuous Testing, and Continuous Delivery. In: *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2015, S. 85–89

[75] Soppelsa, Fabrizio ; Kaewkasi, Chanwit: *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016

[76] Southern California, Information Sciences Institute U. of: *Transmission Control Protocol*. Website. 1981. – Online erhältlich unter https://tools.ietf.org/html/rfc793; abgerufen am 13. August 2018.

[77] Stewart, R.: *Stream Control Transmission Protocol*. Website. 2007. – Online erhältlich unter https://tools.ietf.org/html/rfc4960; abgerufen am 13. August 2018.

[78] Syed, Shahbaz A. ; Soomro, Tariq R.: Achieving Software Release Management and Continuous Integration using Maven, Jenkins and Artifactory. In: *International Journal of Experiential Learning & Case Studies* 3 (2018), Nr. 2, S. 236–245

[79] Tanenbaum, Andrew S.: *Moderne Betriebssysteme*. Pearson Deutschland GmbH, 2009

[80] TaskForce, Internet E.: *Internet Engineering TaskForce*. Website. – Online erhältlich unter https://www.ietf.org/; abgerufen am 11. Juni 2020.

[81] Team, Clang: *ClangFormat*. Website. 2018. – Online erhältlich unter https://clang.llvm.org/docs/ClangFormat.html; abgerufen am 27. September 2018.

[82] Team, The J.: *The new major version of the programmer-friendly testing framework for Java*. Website. – Online erhältlich unter https://junit.org/junit5/; abgerufen am 11. Juni 2020.

[83] TeamCity: *Leistungsfähige, sofort einsatzbereite kontinuierliche Integration*. Website. – Online erhältlich unter https://www.jetbrains.com/de-de/teamcity/; abgerufen am 11. Juni 2020.

[84] test google: *Welcome to Google Test, Google's C++ test framework!* Website. – Online erhältlich unter https://github.com/google/googletest; abgerufen am 11. Juni 2020.

[85] VectorCAST: *Software-Testautomatisierung für qualitativ hochwertige Software*. Website. – Online erhältlich unter https://www.vector.com/de/de/produkte/produkte-a-z/software/vectorcast/; abgerufen am 11. Juni 2020.

[86] Warentest, Stiftung: *Stiftung Warentest*. https://www.test.de. 2017. – Accessed: 2020-04-24

[87] Weidner, Mathias: *RFCs für Paketfilter*. Website. 2015. – Online erhältlich unter https://www.computerweekly.com/de/definition/Content-Filter-Informationsfilter; abgerufen am 24. April 2020.

[88] weitere, D. L. und: *Transport-Protocol/inet-private*. Website. – Online erhältlich unter https://github.com/Transport-Protocol/inet-private/wiki; abgerufen am 12. Juni 2020.

[89] Wong, W E. ; Qi, Yu ; Zhao, Lei ; Cai, Kai-Yuan: Effective fault localization using code coverage. In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)* Bd. 1 IEEE (Veranst.), 2007, S. 449–456

[90] Zhang, Xinyou ; Li, Chengzhong ; Zheng, Wenbin: Intrusion prevention system design. In: *The Fourth International Conference onComputer and Information Technology, 2004. CIT'04.* IEEE (Veranst.), 2004, S. 386–390

[91] Zhaojuan Yue ; Yongmao Ren ; Jun Li: Performance evaluation of UDP-based high-speed transport protocols. In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*, 2011, S. 69–73

[92] Zimmerer, P.: Strategy for Continuous Testing in iDevOps. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, S. 532–533

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Masterarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Gesteuertes Testverfahren in einer Simulationsumgebung am Beispiel des QUIC Handshakes

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

Ort	Datum	Unterschrift im Original