



Universität Passau  
Fakultät für Informatik und Mathematik  
Juniorprofessur für Sicherheit in Informationssystemen  
Prof. Dr. Hans P. Reiser

Bachelorarbeit  
in  
Internet Computing

**Automatisierte Erzeugung von spezifizierbaren  
Client Anfragen zum Testen von  
Netzwerkapplikationen**

Marc Wendelborn

Datum: 09.05.2019  
Prüfer: Prof. Dr. Hans P. Reiser  
Betreuer: Johannes Köstler

# Erklärung

Nachname, Vorname:  
Matrikelnummer:

Wendelborn, Marc  
77281

Hiermit erkläre ich, dass ich die Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate auch als solche gekennzeichnet habe.

.....  
(Datum)

.....  
(Unterschrift des Studenten)

**Kontakt des Prüfers:**

Prof. Dr. Hans P. Reiser

Juniorprofessur für Sicherheit in Informationssystemen

Universität Passau

Email: [hr@sec.uni-passau.de](mailto:hr@sec.uni-passau.de)

Web: <https://www.fim.uni-passau.de/en/sis/>

## Abstract

Das Testen von Netzwerkanwendungen (wie z.B. einfache Webanwendungen oder verteilte Systeme), ist elementar, um eine hohe Verfügbarkeit und Sicherheit zu gewährleisten. Dazu gehört es, dass Anfragen von Clients an das System simuliert werden. Die manuelle Simulation einer Vielzahl solcher Anfragen würde zu viel Zeit in Anspruch nehmen. Deswegen wird eine Anwendung benötigt, mit der spezifizierbare Client Anfragen automatisiert erzeugt werden können. In dieser Arbeit wird eine Anwendung vorgestellt, mit der ein Nutzer eine Folge von Client Anfragen zum Testen von Netzwerkanwendungen konfigurieren und ausführen kann. Darüber hinaus bietet das Programm eine Auswertung der in Folge der Anfragen enthaltenen Antworten an. Um die Anwendung flexibel einsetzbar und erweiterbar zu machen, wird die Ausführung der je nach Protokoll unterschiedlichen Client Anfragen unabhängig von protokollspezifischen Eigenschaften entwickelt. Somit kann die Anwendung bei Bedarf um weitere Protokolle erweitert werden.

The testing of web systems (e.g. web applications or distributed systems) is an indispensable task to guarantee a high availability and security of the system. Therefore client requests to the system have to be simulated. Creating a big amount of these requests by hand would be a costly work, so there is a need for an application that is able to automatically create specific client requests. This thesis will introduce such an application that allows a user to create and execute a specific sequence of client requests. Furthermore the program will give the user a feedback about the received responses. To meet the expectation of a flexible and expandable use, the core functionality of the application (the execution of the requests) will be independent from the particularities of the different protocols. This allows further researches to add new protocols to the application and create specific requests for these protocols.

## Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Vergleichbare Arbeiten</b>	<b>3</b>
2.1	Programme für Lasttests . . . . .	3
2.2	Ansätze für das Testen von Netzwerkanwendungen . . . . .	5
<b>3</b>	<b>Hintergrund</b>	<b>8</b>
3.1	Begriffserklärungen . . . . .	8
3.2	Datenübertragung im Internet . . . . .	11
3.2.1	Referenzmodelle . . . . .	12
3.2.2	Internetprotokolle . . . . .	17
<b>4</b>	<b>Eigener Versuch</b>	<b>20</b>
4.1	Anforderungen . . . . .	21
4.2	Ausführung der Anwendung . . . . .	21
4.3	Konfigurationsdatei . . . . .	22
4.3.1	TargetGroups . . . . .	22
4.3.2	Requests . . . . .	23
4.3.3	Schedule . . . . .	27
4.4	Architektur . . . . .	30
4.4.1	Parsen des Workloads . . . . .	31
4.4.2	Ausführen des Workloads . . . . .	33
4.4.3	Ergebnisauswertung . . . . .	36
4.5	Verwendete Technologien . . . . .	38
<b>5</b>	<b>Validierung</b>	<b>40</b>
5.1	Umgang mit Fehlern in der Konfiguration . . . . .	40
5.2	Korrektheit der Ausführung . . . . .	44
5.3	Erweiterbarkeit . . . . .	47
<b>6</b>	<b>Fazit</b>	<b>49</b>
<b>A</b>	<b>Anhang</b>	<b>51</b>

Im Zeitalter der Digitalisierung nimmt die Vernetzung von Rechnern eine immer größere Rolle ein. Zunehmend mehr Aktivitäten des alltäglichen Lebens, wie die soziale Interaktion oder das in Anspruch nehmen von Dienstleistungen (z.B. Bankgeschäfte tätigen oder Einkaufen gehen), verlagern sich in die Onlinewelt. Soziale Netzwerke, Online-Banking oder E-Commerce Systeme sind ein etablierter Teil unserer Gesellschaft und erleichtern uns das Leben. Auch in anderen Bereichen, die weniger im Fokus der öffentlichen Aufmerksamkeit stehen (wie z.B. der Strom- oder Wasserversorgung, der Landwirtschaft oder in der öffentlichen Verwaltung) erhält die Digitalisierung Einzug.

Hinter all diesen Systemen stehen unterschiedliche Netzwerkarchitekturen. Diese reichen von einfachen Client-Server Systemen bis hin zu komplexeren verteilten Systemen. Um eine maximal hohe Verfügbarkeit und Sicherheit dieser Systeme zu gewährleisten, ist es unabdingbar, sie intensiv zu testen. Nur so kann geprüft werden, ob ein System sich auch unter hoher Belastung wie gewünscht verhält. Auch werden dadurch mögliche Schwachstellen oder Fehlfunktionen aufgedeckt. Damit wird das Risiko eines unkontrollierten Fehlers im Normalbetrieb des Systems reduziert und das ganze System somit sicherer und stabiler. Um solche Systeme realistisch zu testen, ist es wichtig, die Alltagsbedingungen so gut wie möglich nachzustellen. Das erfordert das Simulieren von Datenverkehr, also von Paketen, die verschiedene Daten zwischen den vernetzten Rechnern transportieren. Netzwerkapplikationen bieten dabei meistens einen Service an, der von anderen Rechnern genutzt werden kann. Der Datenverkehr besteht aus Anfragen an die Anwendung und den daraus resultierenden Antworten. Diese Art von Datenverkehr wird im Kontext von Netzwerkapplikationen oft als „Workload“ bezeichnet. Um das Verhalten einer Anwendung bei hoher Auslastung zu testen, müssen viele dieser Anfragen erzeugt werden. Da eine manuelle Erzeugung dieses Datenverkehrs sehr aufwändig ist, wird ein Tool benötigt, welches den Datenverkehr nach den Anforderungen des Systems generiert und automatisiert an das zu testende System ausliefert. Die Herausforderung besteht darin, dass es nicht eine feste Art von Workload gibt, sondern dieser sehr spezifisch sein kann. Es gibt

## 1 Einleitung

heutzutage eine Vielzahl an unterschiedlichen Netzwerkprotokollen. Diese geben bestimmte Regeln und Formate für das Kommunikationsverhalten von verschiedenen Rechnern vor. Dementsprechend sehen Workloads für jedes Protokoll anders aus. Ebenso müssen automatisch generierte Workloads möglichst authentisch sein, also das Verhalten von realen Nutzern so gut wie möglich simulieren. Ist dies nicht der Fall, verfehlen die auf den Workloads basierenden Tests ihre Wirkung.

Im Rahmen dieser Abschlussarbeit soll ein Workload Generator entwickelt werden, der zum Testen von Netzwerkanwendungen eingesetzt werden kann. Der Workload Generator soll ein unkompliziertes Erzeugen von Client Anfragen ermöglichen, und dabei die oben genannten Herausforderungen berücksichtigen. Der Fokus der Anwendung liegt auf dem Testen von Systemen, die auf der BFT-SMaRt Bibliothek aufbauen. Aber auch für andere Protokolle sollen Client Anfragen erzeugt und ausgeliefert werden können. Um das Tool flexibel und anpassbar zu machen, soll die Kernfunktionalität unabhängig von den Verschiedenheiten der Protokolle sein. Somit kann die Anwendung leicht um weitere Protokolle erweitert werden und wird damit der Schnellebigkeit des Themenbereichs gerecht.

Im zweiten Kapitel der Arbeit werden vergleichbare Arbeiten betrachtet. Dazu wird zwischen wissenschaftlichen Ansätzen zum Testen von Netzwerkanwendungen und kommerziellen Anwendungen zum Ausführen solcher Tests unterschieden. Das folgende Kapitel erläutert den Hintergrund der Arbeit. Dabei werden für die Arbeit wichtige Begriffe vorgestellt, bevor die Funktionsweise der Datenübertragung im Internet erörtert wird. Dafür wird zunächst der Aufbau von Netzwerkprotokollen im Allgemeinen erklärt, bevor auf die für diese Arbeit relevanten Protokolle eingegangen wird. Im vierten Kapitel wird der im Zuge dieser Arbeit entwickelte Workload Generator vorgestellt. Nach einer kurzen Definition der Anforderungen und Ziele wird die Ausführung der Anwendung sowie deren Aufbau und Ablauf erklärt. Im anschließenden Kapitel wird die Anwendung evaluiert und geprüft, ob die definierten Ziele erreicht wurden.

## Vergleichbare Arbeiten

Dieses Kapitel widmet sich vergleichbaren Arbeiten. Dabei werden zuerst Lasttest Programme vorgestellt, die, ähnlich wie die in dieser Arbeit entwickelte Anwendung, Tests für Netzwerkkapplikationen durchführen und auswerten können. Anschließend werden wissenschaftliche Ansätze für das Testen von Netzwerkkapplikationen erläutert.

### 2.1 Programme für Lasttests

Workload Generatoren haben eine lange Historie. Bereits Ende der 90er Jahre begann man, sich mit solchen Software-Werkzeugen zu beschäftigen. Zur damaligen Zeit war das „World Wide Web“ in kommerzieller Hinsicht noch in den Kinderschuhen. Als im Jahr 1993 mit „Mosaic“ der erste graphische Browser entwickelt wurde, war das „World Wide Web“ für die breite Masse leichter zugänglich und die Nutzerzahlen stiegen [1].

Dieser Nutzeranstieg machte es notwendig, Lasttests für Webserver durchzuführen, um ihr korrektes Verhalten bei Nutzeranfragen sicherzustellen. Die ersten bekannten Programme für solche Tests waren „httpref“ [2], „SPECweb99“ [3] (der Nachfolger von SPECweb96) und „WebSTONE“ [4]. Diese Tools sind aufgrund ihres Alters aus heutiger Sicht sehr rudimentär in ihrer Funktionalität und ermöglichen lediglich das Erzeugen von HTTP Requests, die an einen zu testenden Webserver geschickt werden.

Heute gibt es eine sehr große Auswahl an Workload Generatoren. Monika Sharma, Vaishnavi S. Iyer, Sugandhi Subramanian und Abhinandhan Shetty bieten in ihrem Artikel „Comparison of Load Testing Tools“ einen Überblick über weit verbreitete Workload Generatoren und vergleichen diese anhand verschiedener Parameter wie z.B. Nutzerfreundlichkeit, Kosten und der Möglichkeit, unbegrenzt Workload zu generieren [5].

Im Folgenden werden die Tools „Gatling“, „WebLOAD“ und „Apache JMeter“ detailliert vorgestellt. Diese Tools gehören zu den am weitesten verbreiteten Workload



## 2 Vergleichbare Arbeiten

Generatoren und werden kommerziell vertrieben. Aus technischer Sicht stellen sie mit das höchste Level im Bereich der Erzeugung von Lasttests dar.

### ***Gatling***

Gatling ist ein open source Tool zum Ausführen von Lasttests für Webserver. Es wird hauptsächlich von dem Unternehmen „Gatling Corp“ weiterentwickelt. Um einen Lasttest mit Gatling auszuführen, werden zuerst sog. *Szenarios* aufgenommen. Hierfür simuliert man eine Nutzerinteraktion mit der Website, während im Hintergrund ein Rekorder diese mitschneidet und in eine Experimentbeschreibung (domänenspezifische Sprache (DSL)) umwandelt. Diese Szenarios werden gespeichert und lassen sich nun beliebig miteinander kombinieren. Dies ermöglicht eine modulare Zusammenstellung komplizierterer Nutzerinteraktionen und eine einfache Wiederverwendung von bereits erzeugten Szenarien. Nach der erfolgten Automation der Nutzerinteraktion wird als nächstes das Ausführverhalten des Lasttests festgelegt. Dabei können eine Vielzahl an Parametern, wie z.B. die Nutzeranzahl oder die Maximallaufzeit, definiert werden. Ist dies abgeschlossen, wird die Simulation ausgeführt. Hierbei erhält der Nutzer über die Konsole Informationen zum Ausführungsstatus (für eine detaillierteres Live-Monitoring wird die kostenpflichtige Erweiterung „Gatling Frontline“ benötigt).

Nach Abschluss der Simulation, generiert Gatling ein HTML-Dokument mit den Ergebnissen des Tests. In diesem sind verschiedene Graphen und Statistiken zu den Antwortzeiten, ausgeführten Aufrufen und weiteren Werten enthalten.

### ***WebLOAD***

WebLOAD wird von der Firma RadView entwickelt und ist ein von Unternehmen wie eBay oder Ebix Inc genutztes Tool für Lasttests [6]. Der Ablauf eines Lasttests mit WebLOAD ähnelt dem vom Gatling. Zuerst wird eine Experimentbeschreibung erstellt. Dies geschieht, wie in Gatling auch, mit einem Rekorder, der die Nutzerinteraktion aufzeichnet und in eine Experimentbeschreibung übersetzt. Dabei bietet WebLOAD die Möglichkeit, Selenium als Rekorder zu integrieren. Die Experimentbeschreibung wird in eine andere DSL übersetzt als bei Gatling. Diese ist aufgrund einer großen Auswahl an Tools auch ohne der Benutzung eines Rekorders leicht zu erstellen bzw. zu bearbeiten. Zum Ausführen des Lasttests wird mit Hilfe der WebLOAD Console eine Vorlage erstellt, in der die Eigenschaften des Lasttests spezifiziert werden können.

Nach der Ausführung des Tests bietet WebLOAD Analytics eine detaillierte Auswertung des Lasttests. Dabei sind eine Vielzahl an Werten abrufbar und die Aufbereitung der Statistiken durch Graphiken ermöglicht ein einfaches Verständnis.

### ***Apache JMeter***

Apache JMeter ist ein weiteres open source Tool zur Erzeugung von Last-

tests, welches hauptsächlich von der Apache Software Foundation entwickelt wird. Im Vergleich zu den anderen beiden vorgestellten Tools, kann ein Lasttest anders aufgebaut werden. Anstatt zuerst eine Experimentbeschreibung mittels eines Rekorders zu erstellen, kann man den Lasttest auch modular zusammenbauen. Für einen einfachen Lasttest mit einem HTTP Request heißt das, dass zunächst ein Lasttest Objekt erstellt wird. Diesem wird dann ein Thread Objekt zugeordnet, in dem spezifiziert wird, wie viele Nutzer die Anfrage stellen sollen und in welcher Zeit diese Threads initialisiert werden. Als nächstes wird diesem Thread Objekt ein HTTP Request Objekt hinzugefügt, dass den Request spezifiziert. Um das kurze Beispiel abzurunden, ergänzt man noch ein Ergebnis Objekt, dass die Auswertung des Tests ermöglicht. In diesem Objekt kann sich die gesendete Anfrage sowie die erhaltene Antwort nochmal angeschaut werden.

Neben dieser modularen Zusammenstellung ermöglicht JMeter jedoch auch das Aufzeichnen einer Nutzerinteraktion mit einem Rekorder. Diese Nutzerinteraktion wird dann in eine Objektstruktur übersetzt. Durch diese modulare Zusammenstellung der Lasttests lassen sich solche einfach und schnell verändern.

Die drei beschriebenen Tools legen ihren Fokus auf eine detaillierte Erzeugung von authentischen Workloads, sowie auf eine sehr präzise Auswertung der Lasttests. Im Vergleich zu dem in dieser Arbeit beschriebenen Workload Generator bietet keines dieser Tools die Möglichkeit, auf unkomplizierte Weise Workloads für das Testen von BFTSMaRt Anwendungen zu generieren.

## 2.2 Ansätze für das Testen von Netzwerkanwendungen

Im Laufe der letzten Jahre wurden verschiedene Methoden vorgestellt, mit denen Netzwerkanwendungen getestet werden können.

2006 stellten Vahid Garousi, Lionel C. Briand und Yvan Labiche einen Ansatz vor, wie Stresstests für verteilte Systeme auf Grundlage von UML Modellen durchgeführt werden können [7]. Dieser Ansatz setzt voraus, dass ein Design Modell des verteilten Systems in Form eines Sequenzdiagramms vorliegt. Auf Grundlage der in diesem Diagramm enthaltenen Timing- und Architekturinformationen können Voraussetzungen für den Aufbau von Lasttests für das System abgeleitet werden. Das Ziel dieses Ansatz ist es, basierend auf UML Modellen, automatisiert Strategien und Voraussetzungen für Stresstests abzuleiten, mit denen Fehler im Nachrichtenfluss des Systems entdeckt werden können.

Eine weitere Methodik für das modell-basierte Testen wurde von Jian Zhang und Shing Chi Cheung vorgestellt [8]. Die Autoren stellen einen Ansatz vor, mit dem automatisiert Lasttests für Multimedia Systeme generiert werden können. Dabei werden die Systeme mit Petri-Netzen modelliert, in denen der Datenfluss des Systems dargestellt wird. Basierend auf diesen Modellen können Lasttest Szenarien erzeugt werden. Im Gegensatz zu der UML-basierten Lasttest Erzeugung zielen

## 2 Vergleichbare Arbeiten

die mit diesem Ansatz erzeugten Tests nicht auf das Auslasten des Datenflusses innerhalb des Netzwerkes, sondern auf das Auslasten der CPUs der beteiligten Komponenten ab.

Ein älterer Ansatz aus dem Jahr 1995 stellt drei Algorithmen vor, mit denen automatisiert Testszenarien für das Testen von Telekommunikationssystemen erzeugt werden können [9]. Die vorgestellten Technologien sind zwar primär auf das Testen von Telekommunikationssystemen ausgelegt, können aber auch für andere Systeme, die mittels einer Markov-Kette modelliert werden können, angewandt werden. Die drei vorgestellten Algorithmen (*Deterministic State Testing (DST)*), *Deterministic State Testing With State Transition Checking (TDS7)* und *Deterministic State Testing with Length N Cycles (CDST)*) basieren alle auf dem stochastischem Modell einer Markov-Kette.

Du Shen, Qi Luo, Denys Poshyvanyk und Mark Grechanik stellten 2015 einen Ansatz zur Entdeckung von Schwachstellen im Datenfluss von Webanwendungen vor [10]. Der Ansatz basiert auf der Annahme, dass unterschiedliche Kombinationen von Eingaben in die Anwendung (z.B. Client Requests) unterschiedliche Reaktionen des Systems auslösen und somit Schwachstellen erkannt werden können. Der dazu entwickelte Algorithmus (*GeneticAlgorithm-driven Profile*) erstellt auf Grundlage bestimmter Heuristiken Profile für Kombinationen von Eingaben in die Anwendung. Die Profile geben Informationen über die Performanz des Systems unter den gegebenen Eingaben. Durch die Auswertung der Profile lassen sich potentielle Schwachstellen des Systems erkennen. Der Algorithmus gehört zu den sog. *genetischen Algorithmen*. Ansätze, wie diese Form der Algorithmen für das Stresstesten von Webanwendungen genutzt werden können, gibt es auch schon seit dem Beginn des aktuellen Jahrtausends. So beschrieben zum Beispiel Lionel C. Briand, Yvan Labiche und Marwa Shousha im Jahr 2005, wie ein genetischer Algorithmus in einem Tool zum Testen von Echtzeit-Anwendungen angewandt werden kann [11]. Muhammad Zohaib Iqbal, Andrea Arcuri und Lionel Briand zeigten im Jahr 2012, auf welche Art und Weise genetische Algorithmen für das Testen von RTEs (*Real Time Embedded Systems*) verwendet werden können [12].

Ein anderer Ansatz im Bereich der automatisierten Lasttests ist das Analysieren von Logs, die während der Ausführung eines Stresstests generiert werden. Durch die automatisierte Analyse der Logs lassen sich Rückschlüsse auf das Verhalten eines Systems unter hoher Auslastung ziehen. Eine Umsetzung dieser Methodik wurde 2009 durch Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann und Parminder Flora vorgestellt [13]. Dabei wurde ein Algorithmus entwickelt, der die Log-Dateien von Stresstests mit der Performanz des Systems unter normalen Bedingungen vergleicht.

Die in diesem Abschnitt dargestellten Arbeiten konzentrieren sich darauf, automatisiert Testszenarien zu erstellen, mit denen Schwachstellen aufgedeckt oder

## *2 Vergleichbare Arbeiten*

Lasttests ausgeführt werden können. Der in dieser Arbeit vorgestellte Ansatz zielt nicht auf die Erzeugung solcher Testszenarien, sondern unterstützt die Erzeugung dieser Szenarien.

### 3.1 Begriffserklärungen

Im Folgenden werden die für diese Arbeit relevanten Begriffe „Emusphere“, „BFT-SMaRt“ und „Workload“ erörtert.

#### *Emusphere*

Emusphere ist eine an der Universität Passau entwickelte Plattform zum Testen von verteilten Systemen. Zwar gibt es viele weitere Testplattformen für verteilte Systeme, jedoch sind diese oft schwerfällig in der Benutzung oder haben zu wenige automatisierte Prozesse. Deswegen wurde bei der Entwicklung von Emusphere der Fokus insbesondere auf die Nutzerfreundlichkeit und ein hohes Automatisierungslevel gelegt. Weitere wichtige Eigenschaften von Emusphere sind die Wiederholbar- und Übertragbarkeit von Experimenten, sodass die erhaltenen Testergebnisse eine hohe Aussagekraft haben.

Um ein verteiltes System mit Emusphere zu testen, wird zuerst eine Experimentbeschreibung mit Hilfe der „Experiment Description Language“ (eine domänenspezifische Sprache) erstellt. In dieser sind alle relevanten Informationen zum Aufbau, dem Ablauf und den Rahmenbedingungen des Experiments enthalten. Diese Beschreibung wird im nächsten Schritt an den „Emusphere Executor“ weitergegeben. Dieser stellt die Benutzeroberfläche der Emusphere Plattform da und startet die in der Experimentbeschreibung definierten Tests. Dafür werden die Tests mit den von der „Host Environment“ zur Verfügung gestellten Ressourcen ausgeführt [14]. Abbildung 3.1 skizziert diesen Ablauf.

Die in dieser Arbeit vorgestellte Anwendung wird unabhängig von Emusphere entwickelt. Jedoch könnte sie in einem weiteren Schritt in Emusphere integriert werden.

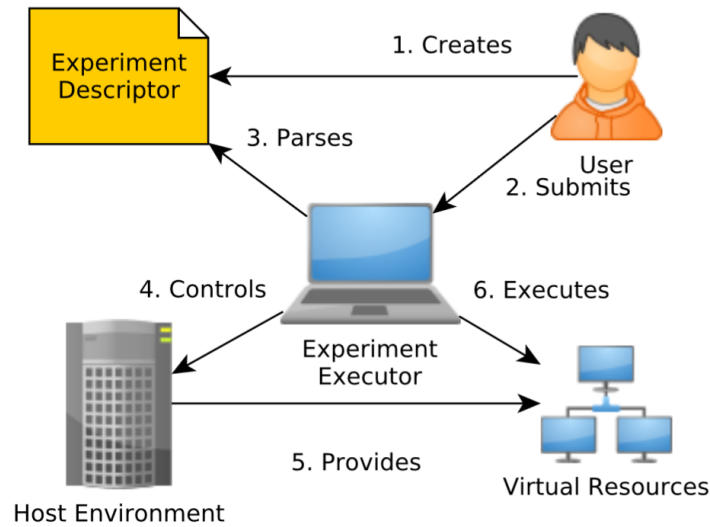


Abbildung 3.1: Ausführung eines Experiments mit Emusphere [14]

#### ***BFT-SMaRt***

BFT-SMaRt ist eine frei zugängliche Java Bibliothek, die mithilfe von replizierten Zustandsautomaten (in englisch: State machine replication (SMR)) eine höhere Ausfallsicherheit bzw. Fehlertoleranz für verteilte Systeme durch die Berücksichtigung der Eigenschaften des byzantinischen Fehlertoleranz Modells (Byzantine fault tolerance (BFT)) bietet.

SMR beschreibt eine Architektur für eine fehlertolerante Anwendung. Hierbei gibt es mehrere Zustandsmaschinen, die anfangs alle denselben Zustand haben, sowie einen Client. Eine Anfrage eines Clients wird an jede Zustandsmaschine geschickt. Alle korrekt funktionierenden Zustandsmaschinen produzieren das gleiche Ergebnis und gehen in denselben Zustand über [15]. BFT-SMaRt vereint SMR mit den Eigenschaften des byzantinischen Fehlertoleranz Modells. Unter der byzantinischen Fehlertoleranz versteht man die Eigenschaft eines verteilten Systems, eine bestimmte Anzahl an fehlerhaften Komponenten zu tolerieren. Leslie Lamport beschreibt diese Eigenschaft in seinem Artikel „The Byzantine generals problem“ [16] mit einer Analogie: Mehrere Generäle der byzantinischen Armee belagern eine Stadt von verschiedenen Seiten. Um sich auf einen gemeinsamen Angriffsplan zu einigen, müssen sie untereinander Nachrichten austauschen. Da es sein kann, dass unter ihnen Verräter sind, benötigen sie einen Algorithmus, der die folgenden zwei Eigenschaften besitzt. Zum einen muss dieser dafür sorgen, dass sich alle loyalen Generäle auf denselben Plan einigen, zum anderen, dass es den Verrätern nicht möglich ist, einen schlechten Angriffsplan zu etablieren. Auf verteilte Systeme übertragen bedeutet dies, dass sich alle beteiligten Komponenten auf eine gemeinsame Lösung einigen, und dass diese Lösung nicht von ein paar wenigen fehlerhaften Komponenten

### 3 Hintergrund

beeinträchtigt wird. Hierzu kommunizieren die Zustandsmaschinen untereinander und stimmen sich somit ab. Dies ermöglicht eine höhere Fehlertoleranz in verteilten Systemen, da einige wenige fehlerhafte Zustandsmaschinen keinen entscheidenden Einfluss auf die Zuverlässigkeit des Gesamtsystems haben.

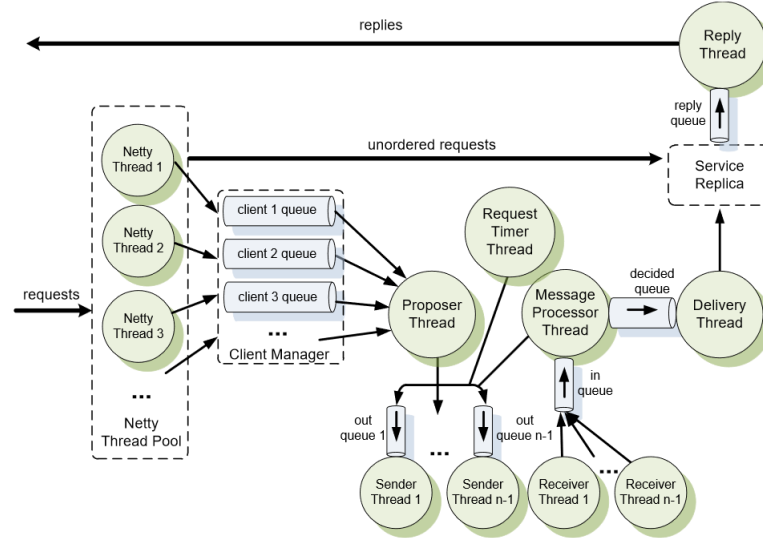


Abbildung 3.2: Kommunikationsprozess in BFT-SMaRt [15]

Abbildung 3.2 skizziert den Prozess vom Empfangen einer Clientanfrage bis zur Auslieferung der Antwort. Sobald eine Nachricht vom Client über den *Thread-Pool* empfangen wurde, wird geprüft, ob die Anfrage geordnet oder ungeordnet ist. Ungeordnete Anfragen müssen die Zustandsmaschinen nicht untereinander abstimmen, sodass diese Anfragen direkt an die *Service Replica* weitergegeben werden. Diese ist für die Generierung der Antwort zuständig. Geordnete Anfragen werden an den *Client Manager* weitergegeben, der die Anfrage verifiziert und einem Client zuordnet. Anschließend erreicht die Nachricht den *Proposer Thread*, der für die Abstimmung der Zustandsmaschinen und die Konsensfindung zuständig ist. Die Nachrichten, die die Zustandsmaschinen dabei austauschen, werden über die *Sender Threads* bzw. die *Receiver Threads* abgewickelt (basieren auf TCP Sockets). Sobald ein Konsens gefunden wurde, wird die Nachricht über den *Delivery Thread* an die *Service Replica* weitergegeben. Über den *Reply Thread* wird die Antwort an den Client gesendet.

In BFT-SMaRt werden Anfragen von Clients und Antworten an diese durch *TOMMessages* repräsentiert. Diese bestehen aus verschiedenen Parametern, wie z.B. der Client ID, der Session ID oder auch dem Nachrichtentyp. Dabei werden hauptsächlich geordnete und ungeordnete Anfragen unterschieden. Der Inhalt der Anfrage, also das Kommando des Clients an die Server, ist ein Parameter in

### 3 Hintergrund

Form eines Bytearrays. Die Zustandsmaschinen kommunizieren untereinander über *ConsensusMessages*. Diese werden verwendet, um sich auf eine gemeinsame Antwort zu einigen. Jene Nachrichten enthalten ebenfalls verschiedene Parameter, wie z.B. eine eindeutige ID oder den Nachrichtentyp.

Für diese Arbeit sind nur die *TOMMessages* relevant, denn diese Client Anfragen sollen von dem hier beschriebenen Workload Generator erzeugt werden können. Die Kommunikation zwischen den Zustandsmaschinen hingegen ist ein interner Prozess, auf den der Nutzer keinen direkten Zugriff hat. Somit ist eine Generierung dieser Nachrichten für eine Benutzersimulation im Kontext dieser Arbeit nicht von Bedeutung.

#### **Workload**

Der Begriff *Workload* lässt sich sinngemäß in den deutschen Begriff Arbeitslast übersetzen, welcher in der Informatik für unterschiedliche Prozesse verwendet wird. Im Kontext dieser Arbeit beschreibt der Begriff *Workload* den Datenverkehr, der bei der Ausführung von Netzwerkanwendungen von einem Client zu einem Server geschickt wird. Aufgrund der Vielzahl der heutzutage eingesetzten Netzwerkprotokolle (siehe Tabelle 3.1) lässt sich dieser Datenverkehr nicht vereinheitlichen, sondern ist je nach eingesetztem Protokoll unterschiedlich. Auch die Art und Weise, wie der Datenverkehr übertragen wird, variiert (z.B. verbindungsorientiert vs. verbindungslos). Weiter beschränkt sich der Begriff Workload in dieser Arbeit nicht auf einzelne Client Anfragen, sondern kann aus einer Vielzahl aneinandergereihter Anfragen von verschiedenen Clients an einen oder mehrere Server bestehen. Dabei können sich die Anfragen nicht nur in ihrem zugrundeliegenden Protokoll, sondern auch in dem Zeitpunkt ihrer Ausführung unterscheiden. Zusammenfassend steht der Begriff Workload für ein Bündel von verschiedenen Client Anfragen, die nach einem bestimmten Zeitplan an einen oder mehrere verschiedene Server geschickt werden.

Da es für das realistische Testen von Netzwerkanwendungen nicht ausreichen würde, einen einmal definierten Workload normal auszuführen, kann durch den in dieser Arbeit vorgestellte Workload Generator der Workload dynamisch zur Laufzeit verändert werden. Diese Veränderungen werden als *spezifizierter* Workload bezeichnet. Spezifizierter Workload ermöglicht es, das Nutzerverhalten von realen Anwendungsszenarien zu simulieren. Die verschiedenen Spezifikationen werden in Kapitel 4.3.3 (Abschnitt: Optionen) vorgestellt.

## 3.2 Datenübertragung im Internet

Das Internet, wie wir es heute kennen, bietet eine Vielzahl an Anwendungen. Dabei ist deren Nutzung schon lange nicht mehr der Wissenschaft oder dem Militär vorbehalten. Das Internet ist mittlerweile für fast jeden zugänglich. Um die große Masse an Nutzern, Anwendungen und Daten zu bewältigen, braucht es Regeln, die die Struktur des Internets vorgeben. Diese Regeln werden Protokolle genannt. Es



gibt zum jetzigen Zeitpunkt über 500 solcher Internetprotokolle, die oft als Internet-Protokollfamilie bezeichnet werden. Die Protokolle arbeiten auf unterschiedlichen Schichten und ergänzen sich zum Teil. Im Folgenden wird nun die Architektur der Internetkommunikation anhand der bekanntesten zwei Referenzmodelle (ISO/OSI Schichtenmodell und TCP/IP Referenzmodell) erklärt. Anschließend werden die Protokolle vorgestellt, auf deren Grundlage der in dieser Arbeit vorgestellte Workload Generator Datenverkehr erzeugt.

#### 3.2.1 Referenzmodelle

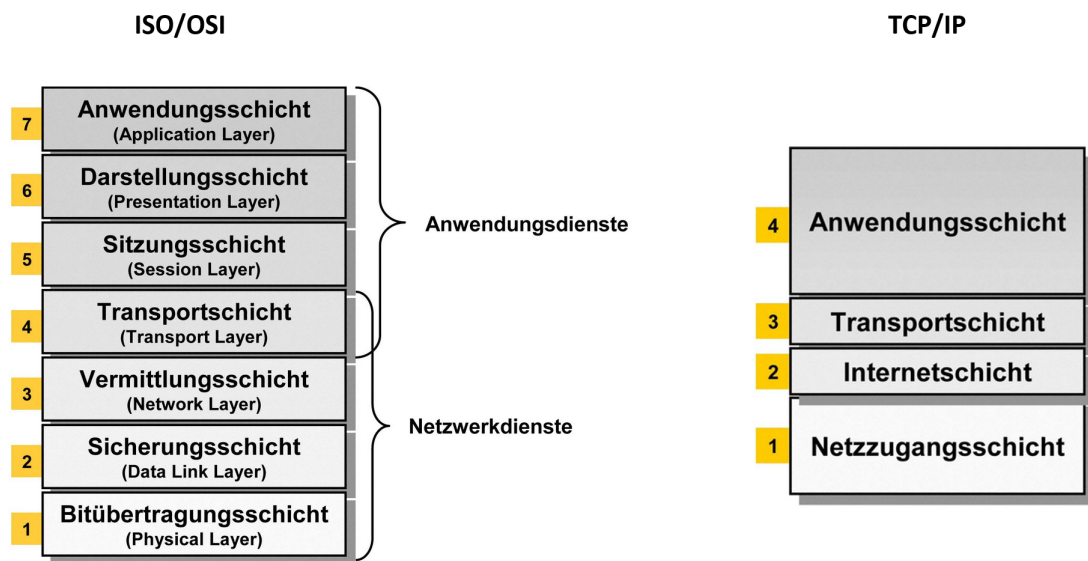


Abbildung 3.3: Referenzmodelle: Darstellung des ISO/OSI Schichtenmodells [17] (links) und des TCP/IP Schichtenmodells (rechts) [17]

#### ISO/OSI Schichtenmodell

Im Laufe der 1980er Jahre entwickelte die *International Organisation for Standardization* (ISO) das *Open System Interconnection* (OSI)-Referenzmodell. Ziel dieses Modells war es, eine einheitliche Grundlage für die Kommunikation von verschiedenen Rechnern zu schaffen. Damit sollte verhindert werden, dass es zur Bildung vieler homogener Teilnetze kommt, die miteinander nicht kompatibel sind und somit eine Kommunikation nicht möglich ist [18].

Das Modell ist in sieben Schichten unterteilt (siehe Abbildung 3.3), von denen jede eine bestimmte Aufgabe übernimmt. Beim Senden einer Nachricht mittels des ISO/OSI Schichtenmodells kommt es zur schrittweisen Kapselung. Angefangen bei der Obersten fügt jede Schicht dem Header der zu übermittelnden Nachricht nacheinander spezifische Informationen (welche Informationen das sind, wird in

### 3 Hintergrund

den nachfolgenden Erklärungen zu den einzelnen Schichten erläutert) zu. Auf der untersten Schicht wird die vollständig gekapselte Nachricht physikalisch an den empfangenden Rechner übertragen. Auf der Empfängerseite wird dieser Prozess nun rückwärts durchlaufen, man spricht von einer *Entkapselung der Nachricht*. Dabei werden die vom Sender hinzugefügten Kontrollinformationen von der jeweils gleichgestellten Schicht ausgewertet.

Die obersten Schichten (5-7) werden als *anwendungsorientiert*, die unteren (1-3) als *netzwerkorientiert* bezeichnet. Dabei werden die Dienste und Funktionen der unteren drei Protokolle meist mittels Hardware umgesetzt. Die Transportschicht (Schicht 4) nimmt hierbei eine Sonderrolle ein und stellt die Verbindung der beiden Gruppierungen da.

Das Modell besteht aus den folgenden sieben Schichten:

- **Bitübertragungsschicht**

Die Aufgabe der Bitübertragungsschicht ist der Auf- bzw. Abbau der physikalischen Verbindung. Auf dieser Ebene werden die Übertragungsparameter wie z.B. Übertragungsgeschwindigkeit, Übertragungsmedium oder die Bit-Codierung festgelegt. Anschließend wird die Nachricht (bestehend aus Nutzdaten und durch die oberen Schichten hinzugefügten Headerinformationen) bitweise übertragen. Beispiele für auf dieser Schicht angewendete Standards sind *V.24* oder *RS-232-C* (Schnittstellen für die Datenfernübertragung zur Übermittlung von Daten eines Computers auf ein Übertragungsmedium) [19].

- **Sicherungsschicht**

Die Sicherungsschicht kümmert sich um eine möglichst fehlerlose Übertragung und verleiht der zu übertragenden Nachricht eine Struktur. Dazu wird der Bitstrom der Nachricht in einzelne Blöcke (sog. *Frames*) aufgeteilt. Um der Empfängerseite das Erkennen von fehlerhaft übertragenen Blöcken zu ermöglichen, werden den Blöcken Prüfbits angehängt. Dieses Verfahren nennt man Prüfsummenverfahren. Die angehängten Prüfbits ermitteln sich aus dem Inhalt des zu übertragenden Frames. Auf der Empfängerseite bestimmt die Sicherungsschicht nun die Prüfbits des empfangenen Blocks und vergleicht diese mit den vom Sender angehängten Prüfbits. Stimmen diese nicht überein, wurde die Nachricht fehlerhaft übertragen. Ein Beispiel für ein solches Verfahren ist das *zyklische Redundanzverfahren* (*Cycling Redundancy Check* (CRC)).

Eine weitere Aufgabe der Sicherungsschicht ist die sog. *Media Access Control*. Hierbei geht es um den gleichzeitigen Zugriff mehrerer Rechner auf ein gemeinsames Übertragungsmedium. Um eine Datenkollision und infolge dessen einen Übertragungsfehler zu vermeiden, gibt es die Möglichkeiten des kontrollierten bzw. des konkurrierenden Zugriffs. Während beim kontrollierten Zugriff der Zugang zum Übertragungsmedium geregelt wird und somit keine Kollisionen auftreten, werden beim konkurrierenden Zugriff Lösungen für den Fall einer Kollision angeboten. Dafür kommen auf der Sicherungsschicht verschiedene Protokolle wie z.B. der *Token-Ring* (kontrollierter Zugriff) oder *ALOHA*

### 3 Hintergrund

(konkurrierender Zugriff) zum Einsatz. Diese Funktionalität wird durch Hardwarekomponenten wie *Bridges* oder *Switches* umgesetzt.

- **Vermittlungsschicht**

Auf dieser Ebene wird eine Adressstruktur für die Übertragung der Nachricht bereitgestellt. Hierbei geht es um die Wegfindung der einzelnen Pakete durch die Vielzahl an zwischengeschalteten Teilnetzen, die durchquert werden müssen. Aufgrund der Heterogenität der Teilnetze wird für die Übertragung ein Hardware-unabhängiges Format benötigt. Dieses wird durch das auf dieser Schicht eingesetzte IP Protokoll zur Verfügung gestellt. Dieses sieht vor, dass die Nachricht in einzelne Pakete (sog. *IP Datagramme*) aufgeteilt werden. Die IP-Datagramme bestehen aus einem Header, der die IP-Adresse des Zielrechners enthält, und den Nutzdaten der Anwendung. Jedem Rechner wird im World Wide Web eine 32 (IPv4) bzw. 128 Bit (IPv6) lange, eindeutige IP-Adresse zugewiesen, durch die der Rechner eindeutig identifiziert und angesteuert werden kann. Eine solche IP-Adresse besteht aus einer Netzadresse (Identifizierungsnummer des Teilnetzes) und einer Hostadresse (Nummer, die den Rechner innerhalb seines Teilnetzes identifiziert).

Die Umsetzung des IP-Protokolls findet in den Routern statt (Hardware Komponenten, die als Knotenpunkte zwischen Netzwerken dienen). Durch die Headerinformationen der Vermittlungsschicht ist es möglich, dass die Nachrichtenpakete nun von Router zu Router bis zum Zielrechner übertragen werden. Dieses Verfahren nennt man *Routing*. Hierbei werden die Pakete an den für den Rechner des Senders zuständigen Router weitergeleitet. Dieser ermittelt anhand der Zieladresse des Pakets den nächsten *Hop*, also den nächsten Router auf dem Weg zum Zielrechner. Dies wiederholt sich nun so lange, bis das Paket den Router des Zielrechners erreicht und an diesen zugestellt wird. Zur Ermittlung des nächsten Hops vergleichen Router die Zieladresse des Pakets mit ihren Routingtabellen. In diesen Tabellen sind für jede mögliche Zieladresse Einträge hinterlegt, zu welchem Router das Paket als nächstes weitergeleitet werden soll.

- **Transportschicht**

Die Transportschicht stellt das Bindeglied zwischen den in den vorherigen Abschnitten vorgestellten hardwarenahen Schichten und den anwendungsnahen Schichten dar. Dafür wird die zu übermittelnde Nachricht in einzelne Pakete aufgeteilt und an die Vermittlungsschicht weitergeleitet.

Eine weitere Funktion der Transportschicht ist der Aufbau einer Ende-zu-Ende Verbindung zwischen Sender und Empfänger. Transportschichtprotokolle haben dabei folgende Aufgaben:

- Sicherstellung der richtigen Übertragungsreihenfolge
- Vermeidung von Duplikaten
- Vermeidung von Paketverlust, Datenüberlauf und Netzüberlastung

### 3 Hintergrund

Die bekanntesten der auf dieser Ebene eingesetzten Protokolle sind das *Transmission Control Protocol* (TCP) und das *User Datagram Protocol* (UDP). TCP stellt einen verbindungsorientierten, zuverlässigen Dienst bereit, UDP dagegen einen verbindungslosen, unzuverlässigen Dienst.

Verbindungsorientiert bedeutet, dass vor dem Datentransport zwischen den Rechnern eine Verbindung aufgebaut werden muss, welche nach Beendigung des Datenaustausches wieder abgebaut wird. Der Aufbau einer solchen TCP Verbindung erfolgt durch einen *3-Way-Handshake*, der Verbindungsabbau durch einen *4-Way-Handshake*. Verbindungslose Dienste transportieren die Daten in einzelnen Paketen, ohne davor eine Verbindung aufgebaut zu haben. Dabei kann es vorkommen, dass Pakete in der falschen Reihenfolge übertragen werden. Um solche Fehler sowie Duplikate zu vermeiden, wird *Sequencing* eingesetzt. Das bedeutet, dass Datenpakete beim Senden mit einer fortlaufenden Nummer beschriftet werden, sodass das Herstellen der richtigen Reihenfolge sowie das Erkennen von Duplikaten auf der Empfängerseite möglich ist. Zuverlässige Dienste stellen sicher, dass die Daten vollständig übertragen werden. Dies geschieht, indem der Empfänger dem Sender den Erhalt einer Nachricht mittels eines *Acknowledgement-Paket* bestätigt. Somit bemerkt der Sender den Verlust eines Paketes und kann dieses nochmals schicken. Unzuverlässige Dienste verzichten auf diese Bestätigung, um zusätzlichen Übertragungsaufwand zu verhindern. Der Verlust einzelner Datenpakete und somit das nicht vollständige Übertragen einer Nachricht werden dabei in Kauf genommen.

Somit unterscheiden sich TCP und UDP grundlegend in ihren Anwendungsbereichen. TCP eignet sich für Anwendungen, denen eine fehler- und verlustfreie Übertragung wichtig ist, UDP dagegen für Anwendungen, denen eine hohe Übertragungsgeschwindigkeit wichtiger ist (z.B. *Voice over IP* oder Streaming von Videos) als eine vollständige und korrekte Übertragung.

- **Sitzungsschicht**

Die Sitzungsschicht definiert den Aufbau einer Übertragungssitzung (öffnen, schließen und verwalten einer Sitzung) [20]. Neben der Synchronisation der an der Sitzung beteiligten Komponenten werden auf dieser Schicht auch sitzungsspezifische Informationen (z.B. Passwörter) ausgetauscht und die Dialogkontrolle etabliert. In der Praxis wird die Sitzungsschicht selten durch die Implementierung eines entsprechenden Protokolls unterstützt. Beispiele für Protokolle dieser Schicht sind das PAP (*Password Authentication Protocol*) oder das SIP (*Session Initiation Protocol*).

- **Darstellungsschicht**

Die Aufgabe der Darstellungsschicht ist die korrekte Syntax und Darstellung der zu übertragenden Daten. Dies ist notwendig, da verschiedene Rechnerarten Ganzzahlen und Zeichen intern unterschiedlich aufbereiten. Ein Beispiel für eine rechnerunabhängige Beschreibungssprache von Datenstrukturen ist ASN.1 (*Abstract Syntax Notation One*). Dieser Standard beschreibt Datentypen, ohne

### 3 Hintergrund

auf eine rechnerinterne Zeichenkodierung (z.B. UTF-8) einzugehen, und ermöglicht somit den Datenaustausch zwischen zwei unterschiedlichen Rechnertypen. Bei der Übertragung von verschlüsselten Nachrichten werden auf dieser Ebene die Kryptographieverfahren angewandt.

- **Anwendungsschicht**

Die Anwendungsschicht stellt die Schnittstelle zwischen dem Netzwerk und der Anwendung dar. Die auf dieser Schicht arbeitenden Anwendungsprotokolle regeln die Nutzung des Netzwerkes durch eine Anwendung. Hierbei ist zu beachten, dass nicht das Programm selbst diese Schicht darstellt, sondern das jeweilige Protokoll, welches das Programm nutzt.

Tabelle 3.1 gibt eine Übersicht über einige bekannte Protokolle der Anwendungsschicht.

Anwendungsbereich	Protokolle	Transportschicht-protokoll
Elektronische Mail	SMTP (Simple Mail Transfer Protocol) POP3 (Post Office Protocol v3) IMAP (Internet Message Access Protocol)	TCP
Streaming	RTP (Real Time Transport Protocol) RTSP (Real Time Streaming Protocol)	UDP TCP/UDP
World Wide Web	HTTP (HyperText Transfer Protocol)	TCP
Dateiaustausch	FTP (File Transfer Protocol)	TCP
IP Telefonie	SIP (Session Initiation Protocol)	meistens UDP
Fernzugriff	TelNet (Teletype Network)	TCP

Tabelle 3.1: Übersicht über bekannte Protokolle der Anwendungsschicht (aufgeteilt nach ihren Anwendungsbereichen mit Angabe des zugrundeliegenden Protokolls der Transportschicht).

#### TCP/IP Schichtenmodell

Das TCP/IP Schichtenmodell (siehe Abbildung 3.3) ist zeitlich vor dem ISO/OSI Modell entstanden und diente als Grundlage für die OSI-Standardisierung [21]. Im Vergleich zu dem oben vorgestellten ISO/OSI Modell besteht das TCP/IP Modell nur aus vier Schichten. Die oberen drei Schichten des ISO/OSI Modells (Anwendungsschicht, Darstellungsschicht und Sitzungsschicht) werden zu einer Schicht (Anwendungsschicht) zusammengefasst, die unteren beiden (Sicherheitsschicht und Bitübertragungsschicht) zur Netzwerkzugangsschicht. Die Internetschicht ist der Vermittlungsschicht gleichzusetzen [22].

#### 3.2.2 Internetprotokolle

Im diesem Abschnitt werden die drei Protokolle vorgestellt, auf deren Grundlage die in Kapitel 4 vorgestellte Anwendung Workload erzeugt.

##### HTTP Protokoll

Das *HTTP* Protokoll (Hypertext Transfer Protocol) ist eines der am weitesten verbreiteten Protokolle der Anwendungsschicht. Es wird hauptsächlich dazu verwendet, Hypertext Dokumente (Websites) zwischen zwei Rechnern zu übertragen.

Entwickelt wurde es von dem britischen Informatiker Tim Berner-Lee und seinem Team am CERN (europäische Kernforschungseinrichtung in der Schweiz). Im Jahr 1991 wurde die erste Version des Protokolls (HTTP 0.9) zusammen mit der Seitenbeschreibungssprache *HTML* (Hypertext Markup Language) sowie einem ersten rudimentären Webbrowser veröffentlicht [1]. Somit bildet das HTTP Protokoll den Grundstein des World Wide Web und wird heute unter der Aufsicht des *W3C* (World Wide Web Consortium) weiterentwickelt. 1996 folgte durch die Veröffentlichung des RFC 1945 die Version HTTP/1.0, welche das Protokoll neben der bisher einzigen Methode *GET* um die Methoden *POST* und *HEAD* erweiterte [23]. Um die Performanz des Protokolls zu optimieren, wurde bei der im Jahr 1999 veröffentlichten Version HTTP/1.1 darauf verzichtet, für jede Anfrage eine neue TCP Verbindung aufzubauen. Von nun an wurden bestehende Verbindung, falls nötig, aufrechterhalten und konnten somit für mehrere Anfragen genutzt werden. Durch das damit einhergehende Wegfallen des ständigen Auf- bzw Abbaus von TCP Verbindungen (TCP-Handshake, TCP-Tear-down) verbesserte sich die Performanz des Protokolls erheblich [24]. Auch wurde das Protokoll um weitere Methoden wie beispielsweise *DELETE* und *PUT* erweitert. Der aktuelle Standard ist die Version HTTP/2.0, welche im Jahr 2015 eingeführt wurde, und die Geschwindigkeit bei der Übertragung von inzwischen immer umfangreicheren Websites weiter verbessert hat. HTTP Anfragen werden vom Client zum Server mittels einer TCP Verbindung übertragen. Der Aufbau einer solchen Anfrage hängt von der zugrundeliegenden Methode ab. Folgende Methoden stellt das Protokoll zur Verfügung:

- **GET**: Anfordern einer Ressource von einem Webserver durch Angabe der *URI* (Uniform Resource Identifier)
- **POST**: Senden von Daten vom Client an den Server. Die Daten werden als Teil der Nachricht verschickt und können vom Server weiterverarbeitet werden, beispielsweise um neue Ressourcen anzulegen oder Bestehende zu bearbeiten
- **PUT**: Hochladen einer Ressource unter Angabe der Ziel URI
- **DELETE**: Löschen der angegebenen Methode vom Server
- **PATCH**: Modifizieren einer Ressource auf einem Server. Ersetzt im Gegensatz zu PUT nicht die vollständige Ressource, sondern verändert sie nur

### 3 Hintergrund

- **HEAD**: Anfordern des Headers einer GET Anfrage. Erzeugt dieselbe Antwort wie eine GET Anfrage, jedoch besteht diese nur aus den Headerinformationen und nicht aus dem Nachrichteninhalt
- **TRACE**: Liefert die Anfrage in der Form zurück, wie der Server sie empfangen hat und wird zum Kontrollieren des Nachrichtenweges benutzt
- **OPTIONS**: Anfordern von allen zur Verfügung stehenden Methoden des Servers
- **CONNECT**: Öffnet einen direkten Tunnel zur Ressource

Codes	Bedeutung
1xx	Informationen
2xx	Erfolg
3xx	Umleitung
4xx	Ungültige Client Anfrage
5xx	Server Fehler

Tabelle 3.2: Übersicht über die Gruppen von Statuscodes einer HTTP Antwort

Die Antworten auf eine HTTP Anfrage bestehen aus einem Header und dem Dateninhalt. Der Header beinhaltet einen Statuscode sowie weitere Metadaten, wie beispielsweise Länge, Sprache oder Zeitpunkt der Antwort. Der dreistellige Statuscode gibt Auskunft über den Erfolg der Anfrage. Tabelle 3.2 gibt eine Übersicht über die fünf Gruppen von Statuscodes. Innerhalb dieser Gruppen gibt es nochmal eine genauere Unterteilung der Statuscodes, sodass detailliertere Informationen durch den Statuscode übermittelt werden können [25]. So steht zum Beispiel der Code 200 für den standardmäßigen Erfolgsfall, 404 dagegen signalisiert, dass die angeforderte Ressource nicht gefunden werden konnte.

Der Nachrichteninhalt der Antwort besteht aus den zu übermittelnden Daten. Dies sind Ressourcen wie beispielsweise HTML Dokumente, CSS-Stylesheets, Bilder oder auch Skripte (z.B. JavaScript).

#### FTP Protokoll

Das *FTP* Protokoll (File Transfer Protocol) lässt sich wie das HTTP Protokoll der Anwendungsschicht zuordnen und wird für den Austausch von Dateien zwischen einem Client und einem Server genutzt. Erstmals spezifiziert wurde das Protokoll im Jahr 1971 durch Abhay Bhushan [26]. Die Grundlage für den heute gängigen FTP Standard wurde 1985 von Jon Postel und Joyce Reynolds spezifiziert [27].

Wie auch das HTTP basiert das FTP Protokoll auf TCP Verbindungen, jedoch können diese nicht mehrmals genutzt werden. Für jede Anfrage wird eine neue TCP Verbindung benötigt. Der standardmäßige Port von Servern für FTP Anfragen ist

### 3 Hintergrund

Port 21. Dateien können dabei in zwei Richtungen transferiert werden, vom Client zum Server (Upload) oder vom Server zum Client (Download). Dabei unterscheidet man zwischen normalen und anonymen FTP Requests. Bei normalen FTP Requests werden zur Authentifizierung Benutzername und Passwort mitgeschickt. Nur nach einer erfolgreichen Authentifizierung erfolgt der Dateiaustausch. Bei anonymen Anfragen ist keine Authentifizierung nötig.

#### **BFT-SMaRt Protokoll**

Anders als das FTP und das HTTP Protokoll, ist BFT-SMaRt nur auf verteilte Systeme anwendbar. Wie in Kapitel 3.1 detailliert beschrieben, sorgt es dafür, dass sich mehrere Server (Replikas) auf eine gemeinsame Antwort an den Client einigen und das Gesamtsystem somit robuster gegenüber Manipulation wird. BFT-SMaRt wurde von Alysson Bessani, João Sousa (beide von der Universität Lissabon) und Eduardo Alchieri (Universität Brasilia) entwickelt, und in einem 2013 erschienen Artikel vorgestellt [15]. BFT-SMaRt ist im eigentlichen Sinne kein Protokoll, sondern eine Java-Bibliothek, die eine Implementierung eines BFT-SMR Protokolls beinhaltet.



## Eigener Versuch

In diesem Kapitel wird der im Rahmen dieser Arbeit entwickelte Workload Generator vorgestellt.

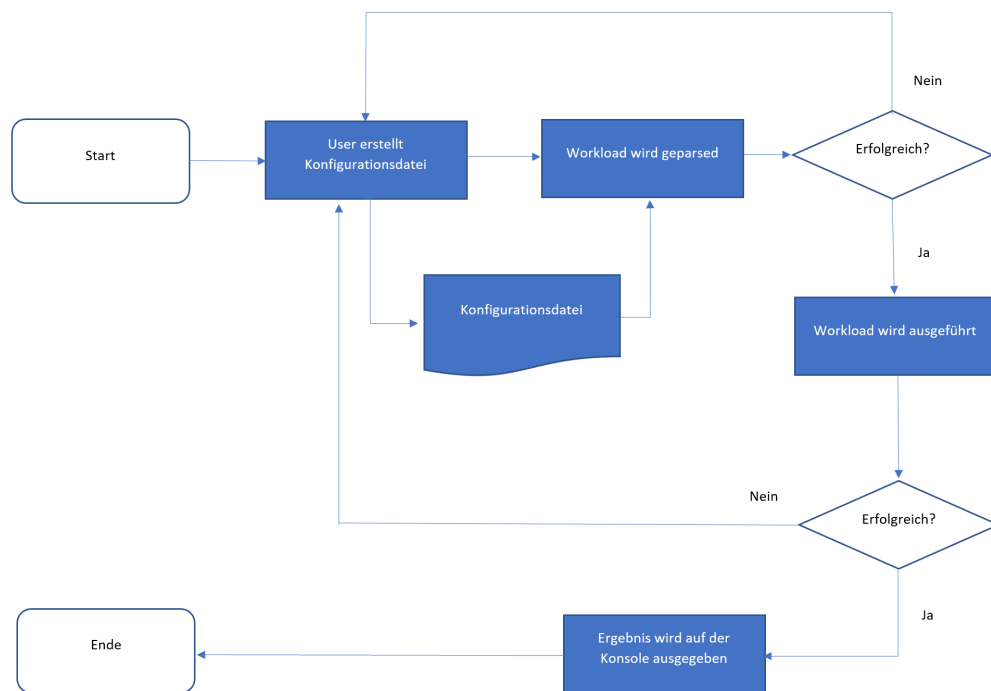


Abbildung 4.1: Ablauf der Anwendung

Abbildung 4.1 skizziert den Ablauf der Anwendung. Dieser besteht aus vier wesentlichen Schritten:

1. Erstellen einer Konfigurationsdatei

2. Parsen des Workloads
3. Ausführen des Workloads
4. Ausgabe der Ergebnisse

In diesem Kapitel werden zunächst die Anforderungen (4.1) und die Ausführung (4.2) der Anwendung beschrieben, bevor auf die einzelnen Schritte des Ablaufs eingegangen wird. Dafür wird zuerst erklärt, wie ein Nutzer eine Konfigurationsdatei erstellt (4.3). Anschließend wird der interne Aufbau und Ablauf der Anwendung vorgestellt (4.4). Zuletzt werden die für die Entwicklung verwendeten Technologien vorgestellt (4.5).

### 4.1 Anforderungen

Ähnlich wie die in Kapitel 2.1 vorgestellten Tools soll die Anwendung Workload für Protokolle der Anwendungsschicht (HTTP, FTP, BFT-SMaRt) erzeugen können. Der Fokus der Anwendung liegt auf der Erzeugung von Workload für BFT-SMaRt Anwendungen. Dafür hat der Nutzer die Möglichkeit, den Workload zu definieren, zu spezifizieren, auszuführen und eine Auswertung der Antworten zu erhalten. Die Applikation soll dabei folgende Anforderungen erfüllen:

- Unkomplizierte Konfiguration des Workloads durch JSON
- Ausführung des Workloads nach festgelegtem Zeitplan
- Spezifizierbarkeit des Workloads dynamisch zur Laufzeit
- Auswertung der Antworten des Workloads
- Erweiterbarkeit um weitere Protokolle

### 4.2 Ausführung der Anwendung

Die als .jar Datei vorliegende Anwendung wird über die Kommandozeile gestartet. Dazu muss folgender Befehl eingegeben werden:

```
java -jar [Name der .jar Datei] -f [Pfad der Konfigurationsdatei]
```

Um eine detailliertere Auswertung der Antworten zu erhalten, kann der Nutzer optional noch den Parameter -v angeben. Die Konfigurationsdatei ist ein JSON (*JavaScript Object Notation*) Dokument. Mittels dieses Dokuments kann der Nutzer den Workload definieren.

Durch das Starten der Anwendung werden nicht die definierten Ziele gestartet. Das heißt, die adressierten Server müssen bereits vor dem Start der Anwendung aktiv sein, da sonst keine Verbindung zu den Servern hergestellt werden kann und die Ausführung des Workloads fehlschlägt.

### 4.3 Konfigurationsdatei

Die Konfigurationsdatei dient dem Nutzer zur Definition des Workloads. Dafür wird das JSON Datenformat genutzt. In der Konfigurationsdatei definiert der Nutzer in drei Abschnitten zuerst alle Ziele (einzelne oder mehrere Server), dann die Anfragen (=Requests) und als letztes den Zeitplan (=Schedule), der bestimmt zu welchem Zeitpunkt welcher Request an welches Ziel gesendet werden soll. Jede Konfigurationsdatei hat somit das in Abbildung 4.2 dargestellte Grundgerüst.

```
{
  "targetGroups": [
  ],
  "requests": [
  ],
  "schedule": {
  }
}
```

Abbildung 4.2: Grundgerüst der Konfigurationsdatei

Im Folgenden werden die drei Komponenten genauer vorgestellt. Um den Aufbau der Konfigurationsdatei verständlich zu machen, wird dieser anhand eines Beispiels erklärt. Dieses Beispiel orientiert sich an der in der BFT-SMaRt Bibliothek mitgelieferten Beispielanwendung *MapDemo*. Diese liefert die Implementierung eines Servers, der eine *TreeMap* (*java.util*) zum Abspeichern von Key-Value-Paaren enthält.

In dem Beispiel wird nun eine Client Anfrage an vier BFT-SMaRt Replikas simuliert, die jeweils eine *TreeMap* besitzen. Mit dieser Anfrage soll ein Key-Value-Paar in jeder der *TreeMaps* gespeichert werden. Um eine leichtere Referenzierung auf dieses Beispiel zu ermöglichen, wird das Beispiel *Map Beispiel* genannt.

#### 4.3.1 TargetGroups

In diesem Bereich werden die möglichen Adressaten des Workloads beschrieben. Der Nutzer hat die Möglichkeit, eine oder mehrere *TargetGroups* zu definieren. Jede

```

"targetGroups": [
  {
    "id": "group1",
    "targets": [
      {
        "server": "localhost",
        "port": 11000
      },
      {
        "server": "127.0.0.1",
        "port": 11010
      },
      {
        "server": "localhost",
        "port": 11020
      },
      {
        "server": "localhost",
        "port": 11030
      }
    ]
  }
],

```

Abbildung 4.3: Aufbau der Komponente *targetGroups* für das *Map Beispiel*

Gruppe besteht aus einer eindeutigen *ID* und einer Liste an *Targets*. Diese Liste kann ein oder mehrere Target-Objekte enthalten. Für jedes Target-Objekt muss der *Servername* definiert werden, sowie optional ein *Port*. Abbildung 4.3 zeigt den Aufbau der Komponente für das *Map Beispiel*. In dem Beispiel ist eine TargetGroup definiert. Die Gruppe mit der ID *group1* besteht aus vier Target-Objekten, die alle den gleichen Server beschreiben, sich jedoch in den Ports unterscheiden. Wie in der Abbildung 4.3 zu sehen ist, kann der Server entweder durch seine Domain oder seine IP-Adresse spezifiziert werden. Um mehrere verschiedene TargetGroups zu definieren, werden diese nacheinander aufgelistet.

#### 4.3.2 Requests

Mit dieser Komponente konfiguriert der Nutzer die Anfragen. Wie auch bei den Adressaten hat der Nutzer die Möglichkeit, mehrere *Requests* zu definieren. Grundsätzlich ermöglicht es die Anwendung, Requests für drei Protokolle zu erstellen: HTTP, FTP und BFT-SMaRt. Neben Anfragen für diese drei Protokolle der Anwendungsschicht können auch einfache TCP und UDP Anfragen definiert werden.

Jedes Request-Objekt besteht aus generellen, sowie protokollspezifischen Parametern. Tabelle 4.1 listet die Parameter auf, die für jeden Request spezifiziert werden

#### 4 Eigener Versuch

müssen. In den folgenden Darstellungen werden Parameter, die optional anzugeben

Bezeichnung	Inhalt
ID	Name des Requests
numberOfClients*	Anzahl an Clients, die den Request ausführen
protocol	Zugrundeliegendes Protokoll

Tabelle 4.1: Protokollunabhängige Parameter eines Request

sind, mit einem \* gekennzeichnet. Parameter ohne dieser Angabe sind verpflichtend anzugeben.

Die ID ist ein frei wählbarer, aber eindeutiger Name, und muss als String angegeben werden. Mit *numberOfClients* kann festgelegt werden, wie viele Clients den Request parallel ausführen. Wird hierbei kein Wert angegeben, wird jeder Request standardmäßig von einem Client ausgeführt. Der Parameter *protocol* gibt das Protokoll des Requests an und legt somit fest, welche protokollspezifischen Parameter zusätzlich angegeben werden müssen. Folgende Werte kann der Parameter annehmen:

HTTP, FTP, TCP, UDP und BFTSMART

Abhängig davon müssen folgende protokollspezifische Parameter angegeben werden:

- HTTP

HTTP-Protokoll	
Bezeichnung	Inhalt
method	Zu verwendende HTTP Methode
resourcePath*	Pfad, unter der die angesteuerte Ressource hinterlegt ist
content*	Inhalt, der an den Webserver übermittelt wird

Tabelle 4.2: Parameter eines HTTP Request

- TCP & UDP

TCP-/UDP-Protokoll	
Bezeichnung	Inhalt
content	Inhalt des Request

Tabelle 4.3: Parameter eines TCP/UDP Request

- FTP

FTP-Protokoll	
Bezeichnung	Inhalt
method	Zu verwendende FTP Methode
localResource	Lokale Pfadangabe
remoteResource	Pfadangabe des Servers
username*	Nutzername für Login
password*	Passwort für Login

Tabelle 4.4: Parameter eines FTP Request

- BFTSMART

Eine Besonderheit stellt hierbei der *command* Parameter eines BFT-SMaRt

BFT-SMaRt		
Bezeichnung	Inhalt	
command	Bezeichnung	Inhalt
	type	Art des Byte Arrays
	content	Inhalt des Byte Arrays
type	Typ der Anfrage ( <i>ordered</i> oder <i>unordered</i> )	
clientType	Typ des Clients ( <i>synchron</i> oder <i>asynchron</i> )	

Tabelle 4.5: Parameter eines BFT-SMaRt Request

Request dar. Dieser ist im Vergleich zu den anderen kein standartmäßiges Key-Value-Paar, sondern ein eigenes Objekt mit eigenen Parametern.

BFT-SMaRt versendet Daten in Form von Byte Arrays (siehe Kapitel 4.4.2). Diese Arrays werden mit dem Command Objekt spezifiziert. Das Objekt enthält den *type* Parameter, mit dem der Typ des Byte Arrays spezifiziert wird (*DataOutputStream* oder *ObjectOutputStream*), und einen *content* Parameter.

Ein *DataOutputStream* wird verwendet, um Base64 kodierte Zeichenfolgen zu verschicken. Dementsprechend enthält der *content* Parameter die zu kodierende Zeichenfolge als String.

Ein *ObjectOutputStream* wird verwendet, um ein oder mehrere Objekte zu verschicken. Die Objekte können dabei primitive oder komplexe Datentypen sein. Der *content* Parameter wird verwendet, um die Objekte zu beschreiben. Jedes Objekt wird dabei mit einer ID und dem entsprechenden Datentyp beschrieben. Der *content* Parameter ist ein String, der eine (durch Kommas getrennte) Aneinanderreihung der Objektdefinitionen enthält. Soll beispiels-

#### 4 Eigener Versuch

weise der Integer 4123, der String „foo“ und der Boolean „false“ versendet werden, sieht der content Parameter folgendermaßen aus:

**„content“: „01:int,02:String,03:boolean“**

Die IDs (in dem Beispiel: 01, 02, 03) sind frei wählbar. Um nun die konkreten Werte der Daten festzulegen, werden die IDs als Parameter mit den entsprechenden Werten aufgelistet. Für das obere Beispiel sieht das wie folgt aus:

**„01“: 4123**

**„02“: „foo“**

**„03“: false**

Im genannten Beispiel werden nur Werte von primitiven Datentypen (Ausnahme: String) versendet. Sollen nun Objekte von externen Klassen oder Enums versendet werden, gibt man im content String *Object* bzw *EnumObject* als Datentyp an. Im Parameter zur Objektbeschreibung muss nun der Klassen- bzw. Enumname (Parameter: *classname*) und der Pfad, unter dem die Klasse bzw. das Enum gefunden werden kann (Parameter: *pathname*), angegeben werden. Für ein EnumObject muss noch der Typ (Parameter: *type*) spezifiziert werden, für ein Object der Konstruktor (Parameter: *constructor*). Dieser wird genau wie der content String aufgebaut (Auflistung der gewünschten Datentypen und Spezifikation der Daten als einzelne Parameter).

```
"requests": [
  {
    "id": "request1",
    "protocol": "BFTSMaRt",
    "type": "ordered",
    "command": {
      "type": "ObjectOutputStream",
      "content": "01:EnumObject,02:String,03:String",
      "03": "mapDemo",
      "02": "foo",
      "01": {
        "path": "src\\main\\resources\\BFT-SMaRt\\library\\bin",
        "classname": "bftsmart.demo.map.MapRequestType",
        "type": "put"
      }
    }
  }
],
```

Abbildung 4.4: Aufbau der Komponente *requests* für das *Map Beispiel*

Abbildung 4.4 zeigt den Aufbau der Komponente für das *Map Beispiel*. Von Seiten des Servers werden drei Parameter für das erfolgreiche Speichern eines Key-Value-Paares benötigt: ein EnumObject (Definiert den Typ der Operation, also ob ein Wert gespeichert oder ausgelesen werden soll) sowie zwei Strings (ein Key-Wert und ein Value-Wert). Dementsprechend definiert der content String diese drei Parameter, die anschließend mit Werten versehen werden. In dem Beispiel wird der Wert *mapDemo* unter dem Key *foo* gespeichert. Da der Wert gespeichert werden soll, wird der Enum-Typ *put* der Klasse *MapRequestType* gewählt, die unter dem angegebenen Pfad abgelegt ist.

### 4.3.3 Schedule

Mit dieser Komponente legt der Nutzer den Ablauf der Ausführung des Workload fest. Des Weiteren lässt sich dabei der Workload spezifizieren.

Um den zeitlichen Ablauf der Ausführung zu steuern, stehen dem Nutzer zwei verschiedene Komponenten zur Verfügung: die *Frames* und die *Events*. Ein Schedule-Objekt besteht aus einem oder mehreren Frames. Jedes Frame lässt sich in Events untergliedern. Die Frames werden nacheinander abgearbeitet. Besteht ein Schedule beispielsweise aus drei Frames, wird zuerst das als erstes aufgelistete Frame ausgeführt. Sobald alle Events dieses Frames ausgeführt und abgeschlossen sind, gilt das Frame als vollständig ausgeführt. Erst dann beginnt die Ausführung des zweiten Frames. Nach dessen Beendigung wird das dritte Frame ausgeführt. Sobald die Ausführung des letzten Frames beendet ist, ist der Schedule vollständig abgearbeitet und der Workload ausgeführt.

Jedes Frame besteht aus drei Parametern: einer ID, einer Liste von Events und den Optionen.

- Events

Die Event-Liste eines Frames besteht aus einem oder mehreren Event-Objekten. Ein Event beschreibt, welcher Request zu welchem Zeitpunkt an welches Ziel geschickt werden soll.

Jedes Event-Objekt hat vier verpflichtend anzugebende Parameter. Diese können Tabelle 4.6 entnommen werden. Sowohl der Request als auch die TargetGroup müssen in den oben genannten, gleichnamigen Komponenten definiert worden sein. Die Zeitangabe bestimmt, wie viele Millisekunden nach dem Start der Ausführung des Frames gewartet werden soll, bis das Event ausgeführt wird. Dadurch erhält der Nutzer die Möglichkeit, mehrere Events zeitgleich oder in einer von ihm definierten Reihenfolge ausführen zu lassen.



#### 4 Eigener Versuch

Bezeichnung	Inhalt
ID	Name des Events
request	Name des auszuführenden Request (ein Request mit diesem Namen muss in der Request-Komponente definiert sein)
targetGroup	Name der Liste mit den Zielen, an die der Request gesendet werden soll (ein Objekt mit diesem Namen muss in der TargetGroup-Komponente definiert sein)
time	Zeitpunkt der Ausführung des Events (in Millisekunden)

Tabelle 4.6: Parameter eines Event-Objekts

- Optionen

Das Options-Objekt ist optional anzugeben. Durch das Festlegen von bestimmten Optionen hat der Nutzer die Möglichkeit, den Workload zur Laufzeit zu verändern. Tabelle 4.7 gibt einen Überblick, über die zur Verfügung stehenden Optionen.

Option	Wirkung
transmissionType	Übertragungsmodus (parallel oder sequentiell)
iterations	Anzahl an Wiederholungen des Frames
requestsNumber	Änderung der Anzahl an Ausführungen pro Wiederholung
frequency	Änderungsrate der Ausführungszeit pro Wiederholung

Tabelle 4.7: Übersicht über die Optionen eines Frames

Dem Nutzer stehen zwei Übertragungsmodi zur Verfügung: parallel oder sequentiell. Bei der parallelen Übertragung werden die Events strikt nach Zeitplan ausgeführt. Kommt es dabei zu Überschneidungen bei der Ausführung, teilen sich die Events die zur Verfügung stehende Bandbreite. Bei der sequentiellen Übertragung nutzt nur ein Event die Übertragungsressourcen. Erst wenn das Event beendet ist, kann das Nächste gestartet werden. Dieses hat dann wiederum die zur Verfügung stehende Bandbreite exklusiv für sich. Bei der sequentiellen Übertragung kann es dazu kommen, dass der Übertragungskanal blockiert ist, und ein Event somit nicht zum definierten Zeitpunkt ausgeführt werden kann. Das Event wird dann zum nächstmöglichen Zeitpunkt ausgeführt. Wird kein Übertragungsmodus festgelegt, wird automatisch der parallele Übertragungsmodus gewählt.

Durch die Option *iterations* kann der Nutzer festlegen, wie oft die Ausführung eines Frames wiederholt werden soll. Dies wird benutzt, um den Workload zu spezifizieren. Möchte man beispielsweise eine sich fortlaufend

## 4 Eigener Versuch

erhöhende Anzahl an Requests an einen Server simulieren, kann man das Frame mehrmals nacheinander ausführen lassen. Durch die Optionen *requestsNumber* und *frequency* lässt sich die Anzahl der Events sowie die Frequenz ihrer Ausführung für jeden Durchgang unterschiedlich gestalten. Verschiedene Szenarien, wie beispielsweise das kontinuierliche Erhöhen der Anzahl an Anfragen mit immer kürzer werdenden Abständen zwischen den Anfragen, können somit erzeugt werden. Dies ermöglicht zum Beispiel das Ausführen eines Lasttests für einen Webserver, um herauszufinden, wie sich die Performanz des Servers (Verarbeitung der Anfragen und Senden von Antworten) bei steigender Auslastung verändert.

Soweit nicht anders angegeben, wird jedes Frame standardmäßig einmal ausgeführt. Wird das Frame wiederholt, greifen die Optionen *requestsNumber* und *frequency* erst ab dem zweiten Schritt. Das heißt, jedes Frame wird immer einmal ohne Spezifizierung ausgeführt, bevor dann die Optionen greifen.

Für die Erhöhung der Anzahl an Anfragen pro Durchgang gibt es verschiedene Modi. Diese werden durch die *requestsNumber* Option definiert, die aus dem Parameter *growth* sowie dem optionalen Parameter *linearGrowthFactor* besteht. Der Parameter *growth* kann drei verschiedene Werte annehmen. Dies sind die zur Verfügung stehenden Wachstumsarten:

- Lineares Wachstum (Parameterwert: *linear*):  
Dieser Modus erhöht die Anzahl an Requests pro Wiederholung des Frames konstant um den durch den Parameter *linearGrowthFactor* festgelegten Wert
- Exponentielles Wachstum (Parameterwert: *exponentially*):  
Die Anzahl der Requests erhöht sich für jede Wiederholung des Frames exponentiell
- Fibonacci Wachstum (Parameterwert: *fibonacci*):  
Der Wachstum pro Durchgang orientiert sich an der Fibonacci-Folge. Das bedeutet die Anzahl der Requests in einem Durchgang ist die Summe aus der jeweiligen Anzahl der beiden vorangegangenen Durchgänge

Abbildung 4.5 zeigt die Entwicklung der Anzahl an Requests pro Durchgang. Das in der Abbildung dargestellte Beispiel basiert auf der Annahme, dass für den Iterations Parameter der Wert 10 und für den linearen Wachstumsfaktor der Wert 9 eingestellt, sowie nur ein Event definiert wurde, das aus nur einem Client besteht. Die Liste der Ziele enthält in diesem Beispiel nur ein Ziel.

Mit der Option *frequency* kann der Ausführzeitpunkt der Events verändert werden. Dafür müssen die Parameter *mode* und *factor* angegeben werden. Die Frequenz der Requests lässt sich erhöhen (mode: increase) oder verringern (mode: decrease). Je nach gewähltem Modus wird der Ausführzeitpunkt eines jeden Events pro Durchgang mit dem angegebenen Faktor multipli-

## 4 Eigener Versuch

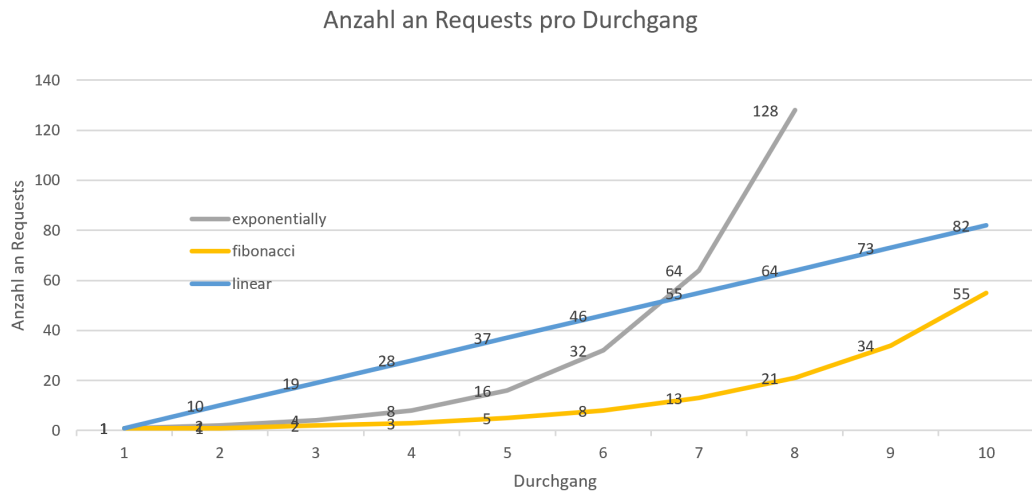


Abbildung 4.5: Überblick über die drei Wachstumsmodi (Der lineare Wachstumsfaktor hat in diesem Beispiel den Wert 9)

ziert (Verringern der Frequenz) bzw. dividiert (Erhöhen der Frequenz). Ist beispielsweise für ein Frame ein Event mit einem Ausführzeitpunkt von vier Sekunden nach Start der Frameausführung definiert, sowie eine Erhöhung der Frequenz um den Faktor zwei bei drei Iterationen veranschlagt, wird das Event beim ersten Durchgang normal nach vier Sekunden ausgeführt. Beim zweiten Durchgang wird das Event nach zwei, beim dritten Durchgang nach einer Sekunde ausgeführt.

Für das *Map Beispiel* wird nur ein Request benötigt. Das heißt, es muss nur ein Frame mit einem Event definiert werden. Da die Standard-Optionen eines Frames jedes Event nur einmal zum definierten Zeitpunkt ausführen, müssen für das aktuelle Beispiel keine Optionen angegeben werden, die Standard-Optionen reichen aus. Abbildung 4.6 zeigt den Aufbau der schedule Komponente für das Beispiel. Da die Standard-Optionen für das Beispiel ausreichen, müssen sie nicht extra angegeben werden. Zur besseren Veranschaulichung sind die Optionen in der Abbildung dennoch aufgelistet.

### 4.4 Architektur

Nachdem eine Konfigurationsdatei erstellt und dem Programm beim Start übergeben wurde, beginnt nun das eigentliche Generieren des Workloads.

Gesteuert wird die Anwendung durch die Klasse *App*. Diese wird bei der Ausführung des Programms gestartet und liest im ersten Schritt die Parameter der Konsoleneingabe aus (siehe Kapitel 4.2). Falls diese korrekt sind und die Pfadangabe

```

"schedule": {
  "frames": [
    {
      "id": "frame1",
      "events": [
        {
          "id": "event1",
          "request": "request1",
          "target": "group1",
          "time": 1000
        }
      ],
      "options": {
        "transmissionType": "parallel",
        "iterations": 1,
        "requestsNumber": {
          "growth": "linear",
          "linearGrowthFactor": 1
        },
        "frequency": {
          "mode": "increase",
          "factor": 1
        }
      }
    }
  ]
}

```

Abbildung 4.6: Aufbau der Komponente *schedule* für das *Map Beispiel*. Die gezeigten Optionen sind die Standard-Optionen eines Frames.

der Konfigurationsdatei existiert, wird mit der Ausführung fortgefahren. Falls nicht, wird ein Hilfstext ausgegeben und die Anwendung beendet.

Die weiteren Schritte nach dem erfolgreichen Starten der Anwendung sind das Parsen des Workloads, die Ausführung dieses Workloads und die Auswertung der Ergebnisse, welche in den nächsten Abschnitten näher erläutert werden.

Die Klassen der Anwendung sind in vier Pakete aufgeteilt. Während die Pakete *requests* und *responses* die Klassen für die (je nach Protokoll unterschiedlichen) Anfragen bzw. Antworten enthalten, sind in den Paketen *parser* und *executor* die für das Parsen bzw. Ausführen des Workloads benötigten Klassen integriert.

#### 4.4.1 Parsen des Workloads

Beim Parsen des Workloads wird die Konfigurationsdatei ausgelesen und aus den Werten eine Objekt Struktur erstellt, die im nächsten Schritt ausgeführt werden kann.

#### 4 Eigener Versuch

Um den Workload zu parsen, wird die Methode *parseWorkload* der Klasse *WorkloadParser* aufgerufen. Diese erhält als Argument den Pfad der auszulesenden Konfigurationsdatei und gibt ein Objekt der Klasse *Workload* zurück. Das Auslesen der Werte aus der Konfigurationsdatei erfolgt mithilfe der Google-Library *JSON.simple*. Diese ermöglicht das Umwandeln von Key-Value-Paaren in String-Objekte und primitive Datentypen (long, boolean). Somit wird aus den gegebenen Konfigurationsparametern eine Objekt Struktur erzeugt. Ist das Parsen erfolgreich, liegt der Workload in Form einer solchen Struktur vor. Diese ist in Abbildung 4.7

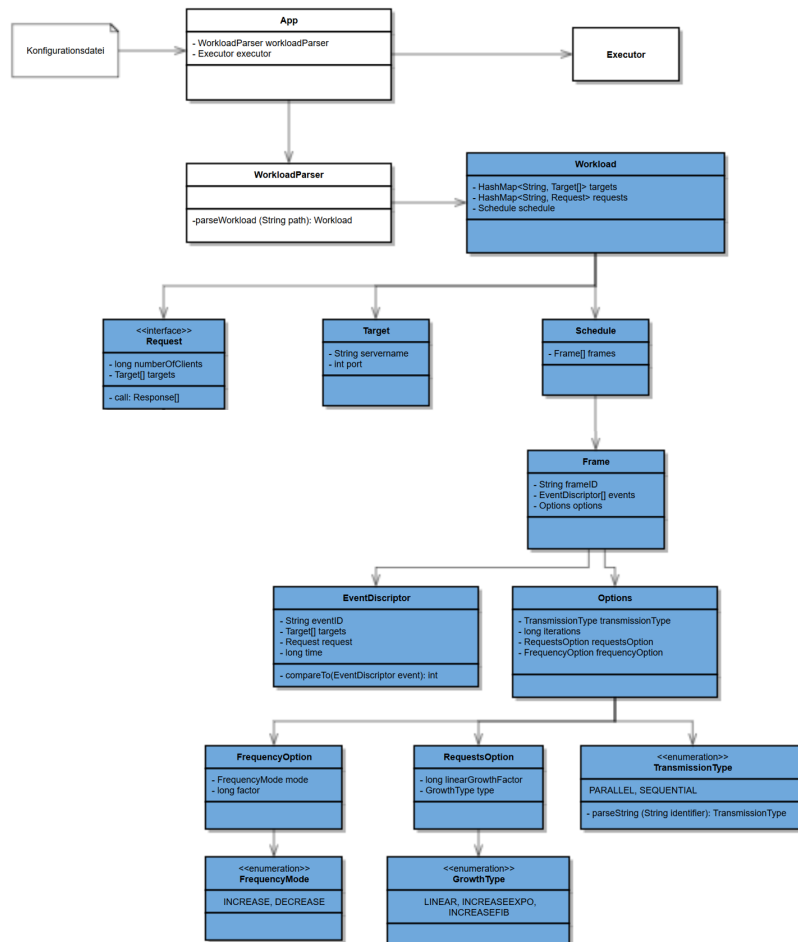


Abbildung 4.7: Klassenstruktur eines Workloads (Die Klassen, die zur Struktur des Workloads gehören, sind blau eingefärbt)

zu sehen.

Aus Gründen der Übersichtlichkeit wird in der Abbildung auf das Aufzählen jedes Attributes oder Methode einer Klasse verzichtet. Das Diagramm ist somit unvollständig und enthält nur die für das Verständnis der Struktur relevanten Attribute und Methoden. Auch stellt das Diagramm nicht alle Klassen der Anwendung dar.

Die für das Ausführen des Workloads und für die Darstellung der Ergebnisse verantwortlichen Klassen werden in dem Diagramm nicht dargestellt.

Fehlt in der Konfigurationsdatei ein Wert, der verpflichtend anzugeben ist, oder hat ein Parameter einen ungültigen Wert, schlägt das Parsen fehl. In diesem Fall wird eine *WorkloadParserException* geworfen, die den Grund für das fehlerhafte Parsen enthält.

### 4.4.2 Ausführen des Workloads

Nach dem Parsen folgt die Ausführung des Workloads. Der Ablauf der Ausführung ist in Abbildung 4.8 zu sehen. Das Diagramm enthält nur die für das Verständnis des Ablaufs wichtigen Methodenaufrufe.

Um die Ausführung zu starten, ruft die Klasse *App* die Methode *executeWorkload* der Klasse *Executor* auf. Diese bekommt als Parameter das im vorherigen Schritt erzeugte Workload Objekt übergeben. Die Rückgabe entspricht einem Objekt der Klasse *WorkloadResult*. Dieses Objekt enthält die Ergebnisse der Ausführung des Workloads.

Zu Beginn der Ausführung wird der Workload in seine verschiedenen Frames aufgespalten, die dann nacheinander abgearbeitet werden. Dies geschieht durch die Methode *executeFrame*. Die Methode führt die Events des Frames nun gemäß dem definierten Zeitplan aus. Ist ein Event an der Reihe, wird die Methode *executeEvent* aufgerufen, die als Parameter ein *EventDescriptor* Objekt übergeben bekommt. Dieses ist die Beschreibung des Events und enthält den Request und die Ziele. Die *executeEvent* Methode führt den Request nun aus.

Die Applikation bietet verschiedene Requests an. Diese unterscheiden sich in ihrem zugrunde liegenden Protokoll und benötigen somit unterschiedliche Parameter für ihre Ausführung. Um die in Kapitel 4.1 gestellte Anforderung der Erweiterbarkeit um weitere Protokolle zu erfüllen, gibt es eine abstrakte Klasse *Request*. Diese stellt die Oberklasse für alle Request-Klassen der verschiedenen Protokolle dar. Somit kann jeder Request unabhängig vom Protokoll ausgeführt werden, da der Executor lediglich die Methode *call* der Oberklasse Request aufruft. Bei der Erweiterung der Anwendung um ein weiteres Protokoll muss somit keine Änderung an der *Executor* Klasse vorgenommen werden.

Der Executor führt jeden Request in einem eigenen Thread aus. Die Verwaltung dieses ThreadPools übernimmt das *ExecutorService* Objekt. Um ein Event auszuführen, übergibt die *executeEvent* Methode den Request an den ExecutorService. Dieser ruft (sollte ein freier Thread verfügbar sein) die *call* Methode des Requests auf und gibt die Antworten, die bei der Ausführung des Requests erzeugt werden, als *Future* Objekt zurück. Diese Objekte bieten die Möglichkeit, die weitere Ausführung eines Programms solange zu blockieren, bis der von ihnen gehaltene Thread abgeschlossen ist. Diese Eigenschaft nutzt der Executor, um die sequentielle

#### 4 Eigener Versuch

Ausführung der Frames zu gewährleisten. Mit der Ausführung des nächsten Frames wird nur begonnen, wenn alle Future Objekte die Ausführung freigeben, also alle Events abgeschlossen sind.

Da ein Request in mehreren Events verwendet werden kann, kann es vorkommen,

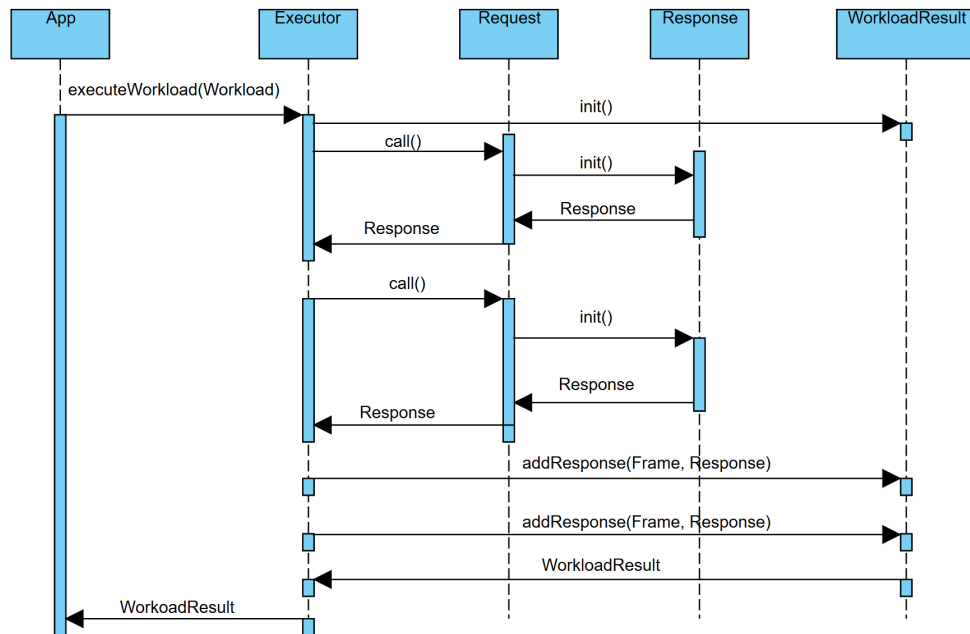


Abbildung 4.8: Beispielhafter Ablauf der Ausführung eines Workloads mit zwei Requests

dass ein Request im Laufe der Ausführung an mehrere unterschiedliche Ziele gesendet wird. Aus diesem Grund wird unmittelbar vor der Ausführung des Requests noch die Methode *setTargets* der *Request* Klasse aufgerufen, um das Ziel (kann ein einzelnes Ziel oder eine Gruppe von Zielen (=TargetGroup) sein) des Requests für das aktuelle Event zu setzen.

Wie bereits erwähnt, benötigen die Protokolle unterschiedliche Parameter. Deswegen gibt es für jedes Protokoll eine eigene Request Klasse. Jede dieser Klassen erbt von der abstrakten Oberklasse *Request* und muss somit eine eigene *call* Methode implementieren. Im Folgenden werden die Ausführungen der unterschiedlichen Protokolle erklärt:

- HTTP

Ein HTTP Request wird mithilfe der „Apache HttpComponents Client“-API ausgeführt. Diese stellt das *CloseableHttpClient* Objekt zur Verfügung,

#### 4 Eigener Versuch

mit dem die Requests ausgeführt werden können. Je nach HTTP Methode werden dafür die entsprechenden Objekte erzeugt (z.B. für die GET Methode ein *HttpGet* Objekt). Diese werden dann mittels der *execute* Methode durch den Client ausgeführt. Pro Client können mehrere Requests gleichzeitig ausgeführt werden. Die Anzahl wird für diese Applikation auf 100 gleichzeitige Verbindungen beschränkt. Wird die *execute* Methode aufgerufen, wird eine der 100 Verbindungen beansprucht (falls vorhanden). Nachdem die Ausführung beendet ist, wird die Verbindung wieder freigegeben. Sollten alle 100 Verbindungen belegt sein, wartet der Client mit der Ausführung solange, bis eine Verbindung frei wird. Unabhängig von der verwendeten HTTP Methode wird ein *HttpResponse* Objekt zurückgeliefert, das die Ergebnisse der Ausführung (z.B. Statuscode) enthält.

- FTP

Für FTP Requests wird die „Apache Commons FTPClient“-API verwendet. Diese stellt mit der *FTPClient* Klasse die Clients für einen FTP Request zur Verfügung. Ähnlich wie der HTTP Client kann ein FTP Client für mehrere Requests genutzt werden. Allerdings kann pro Client nur eine Verbindung aufgebaut werden. Dafür verbindet sich der Client zu dem gegebenen Server und loggt sich gegebenenfalls ein (falls Nutzernamen und Passwort definiert wurden). Mittels eines *FileOutputStream* (FTP GET Methode) bzw. eines *FileInputStream* (FTP PUT Methode) werden die Dateien übertragen. Sobald dieser Vorgang beendet ist, loggt sich der Client aus und trennt die Verbindung. Anschließend steht er für den nächsten Request zur Verfügung. Steht ein Client gerade nicht zur Verfügung (also befindet er sich aktuell in der Ausführung eines Requests), wartet der nächste Request solange, bis der Client wieder zur Verfügung steht.

- UDP

UDP Requests verwenden für die Ausführung *DatagramSocket* Objekte. Diese verschicken mittels der *send* Methode *DatagramPacket* Objekte. Diese Objekte enthalten sowohl die zu übertragenden Daten in Form eines Byte Arrays, als auch ein *InetAddress* Objekt und eine Portnummer. Die Antworten werden ebenfalls in Form eines *DatagramPacket* empfangen. Dieses enthält die erhaltenen Daten (auch in Form eines Byte Arrays).

- TCP

TCP Requests werden mithilfe von *Sockets* ausgeführt. Ein Socket baut durch Angabe eines *InetAddress* Objekts und einer Portnummer eine TCP-Verbindung zu einem Server auf. Über diese Verbindung können nun die Daten verschickt werden. Nach Beendigung der Übertragung wird das Socket Objekt



geschlossen. Dadurch wird die Verbindung zum Server abgebaut. Anders als bei HTTP und FTP Requests wird pro Request ein neuer Client, also ein neues Socket Objekt, benötigt.

- BFT-SMaRt

Für die Ausführung von BFT-SMaRt Requests wird die *BFT-SMaRt* Bibliothek verwendet. Diese stellt eine Schnittstelle für Client-Anwendungen (*ServiceProxy*) und eine für die Replikas (*ServiceReplica*) zur Verfügung. Auf Client Seite gibt es zusätzlich noch eine *AsynchServiceProxy*, die für das Versenden von asynchronen Requests verwendet wird.

Um Requests von einem Client zu den Replikas zu senden, bietet die *ServiceProxy* Klasse die Methoden *invokeUnordered*, *invokeOrdered* und *invokeUnorderedHashed*. Bei geordneten Requests versendet der Client die Anfragen in einer geordneten Reihenfolge, bei ungeordneten ist das nicht der Fall. Jede dieser Methoden erwartet als Parameter ein Byte Array, welches den Inhalt der Anfrage erhält. Zusammen mit dem Typ des Requests (geordnet oder ungeordnet) wird der Inhalt in eine *TOMMessage* umgewandelt. Wie in Kapitel 3.1 erklärt, werden diese Objekte zum Nachrichtenaustausch zwischen Client und Replikas verwendet.

Um eine Client Anfrage zu erzeugen, wird also nur der Typ und der Inhalt des Requests benötigt. Der Typ wird in der Konfigurationsdatei direkt spezifiziert, das Array über den *command* Parameter (siehe Kapitel 4.3.2).

Als Antwort auf die Ausführung eines Requests empfängt die *ServiceProxy* die Antwort in Form eines Byte Arrays.

Ist die Ausführung aller Events beendet, liegen die erhaltenen Antworten als Future Objekte vor. Nun beginnt die Auswertung dieser Antworten.

### 4.4.3 Ergebnisauswertung

Um die Ergebnisse auszuwerten, werden zuerst die in Future Objekten gespeicherten Antworten in Response Objekte extrahiert. Dies geschieht durch die *parseResponses* Methode des Executors, die für jedes Future Objekt wartet, bis die Ausführung des Requests beendet ist und ein Response Objekt vorliegt. Die Methode gibt alle erhaltenen Antworten in Form eines Response Arrays zurück.

Für jeden ausgeführten Request wird ein Response Objekt erzeugt. Diese sind, je nach Protokoll des Request, unterschiedlich. Um die einfache Erweiterbarkeit der Anwendung zu gewährleisten, gibt es auch für die Antworten der unterschiedlichen Protokolle eine abstrakte Klasse, die den Aufbau einer Antwort vorgibt. Die Klasse *Response* schreibt vor, dass in jedem Response Objekt die Anfangszeit und die Endzeit der Ausführung des Requests gespeichert sein muss. Weiter wird in jeder Antwort vermerkt, ob der Request erfolgreich ausgeführt wurde oder nicht.

#### 4 Eigener Versuch

Da die Anwendung die Möglichkeit bietet, eine detaillierte Auswertung pro gesendetem Request zu erhalten, muss jede Response Klasse die Methode *print* implementieren. Diese schreibt die für den entsprechenden Request relevanten Informationen in die Ausgabe. Je nach Protokoll unterscheiden sich die relevanten Informationen, sodass für jedes Protokoll, das von der Anwendung unterstützt wird, eine entsprechende Response Klasse vorliegen muss. Diese erweitert die abstrakte Klasse Response und gibt vor, wie die Detailauswertung für das entsprechende Protokoll aussieht.

Wie in Abbildung 4.8 zu sehen, werden die einzelnen Response Objekte in ein Objekt der Klasse *WorkloadResult* gespeichert. In diesem Objekt werden alle bei der Ausführung des Workloads erzeugten Antworten (nach Frames sortiert) gespeichert. Des Weiteren werden in diesem Objekt der Startzeitpunkt sowie der Endzeitpunkt eines jeden Frames gespeichert. Sobald die Antworten und Zeiten von allen Frames gespeichert sind, ist die Ausführung des Workloads beendet. Der Executor gibt das *WorkloadResult* Objekt an den Controller der Anwendung (die Klasse App) zurück. Diese veranlasst als letzten Schritt die Ausgabe der Ergebnisse auf der Konsole. Dazu wird die Methode *printResults* des *WorkloadResult* Objekts aufgerufen. Je nach dem, ob beim Start der Anwendung der Parameter -v für die Detailauswertung angegeben wurde, gibt die Methode nur die Gesamtstatistik der einzelnen Frames aus, oder die Antworten für jeden einzelnen Request. Somit wird für jedes Frame Folgendes ausgegeben:

- Ein Überblick über die Konfiguration des Frames (Anzahl an definierten Events sowie eine Auflistung der gewählten Optionen)
- Eine Gesamtstatistik des Frames:
  - Dauer der Ausführung des Frames in Sekunden
  - Anzahl der insgesamt ausgeführten Requests
  - Anzahl der insgesamt erhaltenen Antworten (prozentualer Anteil der erfolgreichen Requests)
  - Durchschnittliche *Round Trip Time* (RTT) der Requests in Millisekunden
- Eine detaillierte Auswertung pro ausgeführtem Request (falls beim Starten der Anwendung angegeben)

Die RTT bezeichnet die Zeit, die zwischen dem Versenden des Requests und dem Erhalten der Response vergangen ist. Diese wird mithilfe der Start- und Endzeit eines jeden Requests berechnet.

Abbildung 4.9 zeigt die Ergebnisausgabe auf der Konsole anhand eines einfachen Beispiels. In diesem Beispiel wurden zwei Frames ausgeführt, die nur aus HTTP Requests bestehen. Im ersten Frame wurden HTTP Get Requests an die Startseite der Homepage der Universität Passau geschickt, im Zweiten an die

```
-----Result for frame: UniPassauWebsiteTest -----  
  
--- Input ---  
Defined events: 1  
Defined options:  
  Transmission: PARALLEL  
  Iterations: 3  
  Request number: Growth: INCREASEFIB  
  Frequency: Mode: INCREASE , Factor: 2  
  
--- Results ---  
Total execution time for frame (in seconds): 2.151  
Total amount of executed requests: 20  
Responses received: 20 (100.0%)  
Average RTT (in milliseconds): 76.0  
  
-----Result for frame: LehrstuhlWebsiteTest -----  
  
--- Input ---  
Defined events: 1  
Defined options:  
  Transmission: PARALLEL  
  Iterations: 5  
  Request number: Growth: LINEAR , Growth factor: 2  
  
--- Results ---  
Total execution time for frame (in seconds): 5.716  
Total amount of executed requests: 105  
Responses received: 105 (100.0%)  
Average RTT (in milliseconds): 104.24761904761905
```

Abbildung 4.9: Beispielhafte Konsolenausgabe für das erfolgreiche Ausführen eines Workloads

Startseite des Lehrstuhls für Sicherheit in Informationssystemen. Um die Ausgabe übersichtlich zu halten, wurde auf die Detailauswertung der einzelnen Requests verzichtet. Die dargestellten Werte für die RTTs sind nicht belastbar, da sie unter anderem von der Bandbreite des Client Systems abhängen. Somit variieren diese Werte abhängig vom ausführenden Rechner. Die in der Abbildung dargestellten Werte sollen lediglich eine Übersicht über die Ergebnisausgabe des Workload Generators geben und ermöglichen keine Rückschlüsse über die Belastbarkeit der Server.

Die Ausgabe der Ergebnisse ist der letzte Schritt der Anwendung. Sobald die Auswertung ausgegeben wurde, beendet sich das Programm. Um einen Workload zu wiederholen oder um einen Neuen auszuführen, muss die Anwendung mit der entsprechenden Konfigurationsdatei als Parameter neu gestartet werden.

## 4.5 Verwendete Technologien

Die Anwendung wurde auf einem Windows 10 Betriebssystem mit Java 8 entwickelt und benutzt als Build-Management System Maven in der Version 3.6. Folgende

#### 4 Eigener Versuch

Bibliotheken wurden mittels Maven in das Projekt integriert:

- Apache Log4j (Stellt einen Logger bereit, der das Debuggen der Anwendung erleichtert)
- JSON.simple (Unterstützt das Parsen eines JSON Dokuments in eine Java Objektstruktur)
- Apache Commons CLI (Hilft bei der Übersetzung der Konsoleneingaben)
- Apache Commons Net (Stellt ein FTP Client Objekt zur Verfügung, mit dessen Hilfe FTP Requests versendet werden können)
- Apache HttpComponents (Stellt ein HTTP Client Objekt zur Verfügung, welches zum Versenden von HTTP Requests verwendet werden kann)
- BFT Smart/library (Stellt ein ServiceProxy Objekt zur Verfügung, mit dem BFT-SMaRt Requests verschickt werden können)

Als Entwicklungsumgebung wurde Eclipse gewählt. Für die Erstellung der JSON Konfigurationsfiles wurde der Editor Brackets verwendet. Um eine modulare Entwicklung zu unterstützen, wurde das Versionsverwaltungstool GitLab verwendet. Dadurch konnten einzelne Features während dem Entwicklungsprozess in separaten Branches entwickelt und später zur Hauptversion hinzugefügt werden.

In diesem Kapitel wird geprüft, ob der Workload Generator die Anforderungen erfüllt. Dafür werden zuerst Fehlerfälle und der Umgang des Programmes mit diesen geprüft. Anschließend wird getestet, ob der Workload auch so ausgeführt wird, wie es in der Konfigurationsdatei definiert ist. Zuletzt wird erörtert, ob die Anforderung der einfachen Erweiterbarkeit des Programmes um weitere Protokolle erfüllt ist.

Für die Validierung der Anwendung werden in den einzelnen Abschnitten Nutzerszenarien beschrieben. Diese Szenarien sollen realistische Anwendungsfälle des Workload Generators simulieren. Für jedes Szenario wird anschließend geprüft, ob der Workload Generator sich korrekt verhält.

Jedes Nutzerszenario hat folgenden Aufbau:

- Ziel (Beschreibung, welcher Bereich mit dem Nutzerszenario getestet werden soll)
- Ablauf (Beschreibung des Nutzerszenarios)
- Ausgabe (Ausgabe des Workload Generators oder des Servers)
- Beurteilung (Einordnung des Ergebnisses)

## 5.1 Umgang mit Fehlern in der Konfiguration

In diesem Abschnitt wird der Umgang des Programmes mit fehlerhaften Konfigurationen validiert. Dazu werden, wie oben beschrieben, realistische Nutzerszenarien simuliert.

- **Syntaktische Fehler in der Konfigurationsdatei**

### **Ziel**

Das Verhalten des Workload Generators bei syntaktischen Fehlern in der

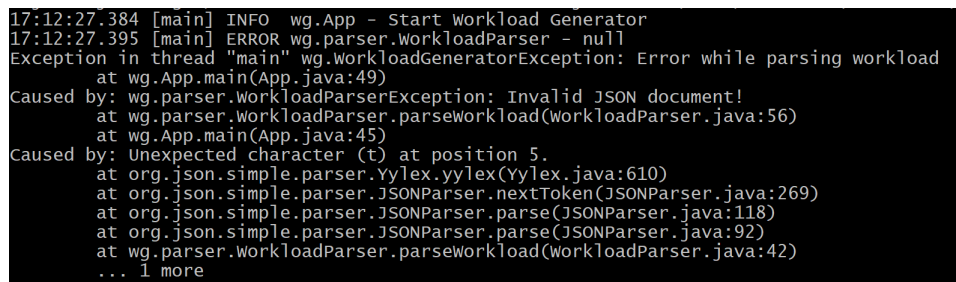
Konfigurationsdatei soll geprüft werden. Die Anwendung soll nicht abstürzen, sondern dem Nutzer eine aussagekräftige Fehlermeldung geben und kontrolliert beendet werden.

### Beschreibung

Ein Nutzer möchte einen BFT-SMaRt Request an vier Server senden. Beim Konfigurieren des Workloads unterläuft ihm ein Syntaxfehler, er vergisst, *targetGroups* in Anführungszeichen zu setzen. Weiter tippt er zwischen die Key-Value-Paare *server* und *port* des zweiten Servers neben dem benötigten Komma noch einen Punkt. Das Dokument besitzt nun keine gültige JSON Syntax mehr. Der Nutzer startet im Anschluss das Programm und übergibt diesem die fehlerhafte Konfigurationsdatei.

### Ausgabe

Abbildung 5.1 zeigt die Ausgabe der Anwendung. Die Anwendung erkennt die fehlerhafte Konfigurationsdatei und gibt eine Fehlermeldung an den Nutzer zurück. Diese beschreibt die Art des Fehlers („Invalid JSON document!“) und wo dieser gemacht wurde („Unexpected character (t) at position 5“). Anschließend wird die Anwendung ordnungsgemäß beendet.



```
17:12:27.384 [main] INFO    wg.App - Start Workload Generator
17:12:27.395 [main] ERROR  wg.parser.WorkloadParser - null
Exception in thread "main" wg.WorkloadGeneratorException: Error while parsing workload
    at wg.App.main(App.java:49)
    at wg.App.main(App.java:49)
Caused by: wg.parser.WorkloadParserException: Invalid JSON document!
    at wg.parser.WorkloadParser.parseWorkload(WorkloadParser.java:56)
    at wg.App.main(App.java:45)
Caused by: Unexpected character (t) at position 5.
    at org.json.simple.parser.Yylex.yylex(Yylex.java:610)
    at org.json.simple.parser.JSONParser.nextToken(JSONParser.java:269)
    at org.json.simple.parser.JSONParser.parse(JSONParser.java:118)
    at org.json.simple.parser.JSONParser.parse(JSONParser.java:92)
    at wg.parser.WorkloadParser.parseWorkload(WorkloadParser.java:42)
    ... 1 more
```

Abbildung 5.1: Ausgabe der Anwendung bei Syntaxfehlern in der Konfigurationsdatei

### Beurteilung

Zwar weist die Fehlermeldung nur auf einen der beiden Syntaxfehler hin, jedoch würde der zweite nach einer Korrektur des ersten Fehlers beim erneuten Ausführen der Anwendung angegeben werden. Die Fehlermeldung beschreibt die Art des Problems und an welcher Stelle dieses vorliegt. Somit erkennt die Anwendung syntaktische Fehler in der Konfigurationsdatei, gibt eine aussagekräftige Fehlermeldung zurück und stürzt nicht ab.

### • Semantische Fehler in der Konfigurationsdatei

#### Ziel

Das Verhalten des Workload Generators bei semantischen Fehlern in der

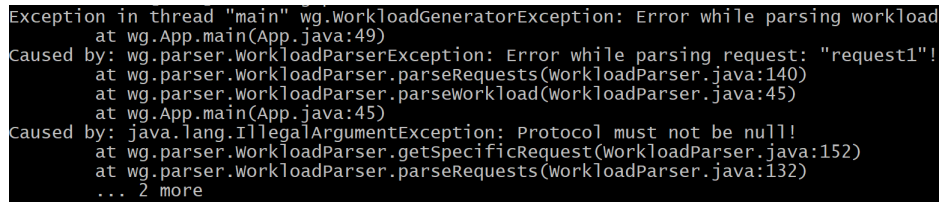
Konfigurationsdatei soll geprüft werden. Das Programm soll nicht abstürzen, sondern dem Nutzer eine aussagekräftige Fehlermeldung geben und kontrolliert beendet werden.

### Beschreibung

Ein Nutzer möchte einen BFT-SMaRt Request an vier Server senden. Dabei lässt er bei der Konfiguration des Requests den Parameter *protocol* weg, da er davon ausgeht, dass das Programm das gemeinte Protokoll automatisch erkennt. Anschließend startet er das Programm und übergibt diesem die fehlerhafte Konfigurationsdatei.

### Ausgabe

Abbildung 5.2 zeigt die Ausgabe der Anwendung. Der Fehler wird erkannt, und die Anwendung gibt eine Fehlermeldung aus. Diese enthält sowohl den fehlerhaft definierten Request („Error while parsing request: „request1!“), als auch die genaue Ursache für den Fehler („Protocol must not be null!“). Die Anwendung wird nach der Fehlerausgabe ordnungsgemäß beendet.



```
Exception in thread "main" wg.WorkloadGeneratorException: Error while parsing workload
    at wg.App.main(App.java:49)
Caused by: wg.parser.WorkloadParserException: Error while parsing request: "request1"!
    at wg.parser.WorkloadParser.parseRequests(WorkloadParser.java:140)
    at wg.parser.WorkloadParser.parseWorkload(WorkloadParser.java:45)
    at wg.App.main(App.java:45)
Caused by: java.lang.IllegalArgumentException: Protocol must not be null!
    at wg.parser.WorkloadParser.getSpecificRequest(WorkloadParser.java:152)
    at wg.parser.WorkloadParser.parseRequests(WorkloadParser.java:132)
    ... 2 more
```

Abbildung 5.2: Ausgabe der Anwendung bei einem Semantikfehler in der Konfigurationsdatei

### Beurteilung

Die Anwendung erkennt fehlende Parameter in der Konfigurationsdatei. Die zurückgegebenen Fehlermeldungen erhalten konkrete Informationen, welcher Fehler vorliegt. Somit ist die Anwendung in der Lage, semantische Fehler zu erkennen und dem Nutzer eine aussagekräftige Fehlermeldung zurückzugeben. Auch wird die ordnungsgemäße Beendigung der Anwendung nicht durch einen Semantikfehler gestört.

#### • Ungültige Konfigurationsparameter

##### Ziel

Das Verhalten der Anwendung bei fehlerhaften Konfigurationsparametern soll geprüft werden. Fehlerhafte Parameter sorgen dafür, dass die Konfigurationsdatei sowohl syntaktisch als auch semantisch korrekt ist, jedoch eine ordnungsgemäße Ausführung des Programmes nicht möglich ist (z.B. Angabe einer ungültigen URL).

### Beschreibung

Ein Nutzer möchte einen HTTP Get Request an die Startseite der Universität Passau ausführen. Dabei gibt er die URL des Servers in der Konfigurationsdatei falsch an (www.un-passauf.de statt www.uni-passau.de). Der Nutzer startet die Anwendung und übergibt die Konfigurationsdatei, die abgesehen von der ungültigen URL syntaktisch und semantisch fehlerfrei ist.

### Ausgabe

Abbildung 5.3 zeigt die Ausgabe der Anwendung. Die Anwendung scheitert bei der Ausführung des Requests und gibt eine Fehlermeldung an den Nutzer zurück. Diese enthält die Art des Fehlers („Error while executing HTTP Request“), sowie die genaue Ursache („UnknownHostException: www.uni-passauf.de“). Die Anwendung wird im Anschluss kontrolliert beendet.

```
Exception in thread "main" wg.WorkloadGeneratorException: Error while executing workload!
    at wg.App.main(App.java:52)
Caused by: java.util.concurrent.ExecutionException: wg.executor.WorkloadExecutionException: Error while executing HTTP request!
    at java.util.concurrent.FutureTask.report(Unknown Source)
    at java.util.concurrent.FutureTask.get(Unknown Source)
    at wg.executor.Executor.parseResponses(Executor.java:218)
    at wg.executor.Executor.executeFrame(Executor.java:128)
    at wg.executor.Executor.executeWorkload(Executor.java:60)
    at wg.App.main(App.java:46)
Caused by: wg.executor.WorkloadExecutionException: Error while executing HTTP request!
    at wg.requests.HttpRequest.executeSingleRequest(HttpRequest.java:137)
    at wg.requests.HttpRequest.call(HttpRequest.java:83)
    at wg.requests.Request.call(Request.java:1)
    at java.util.concurrent.FutureTask.run(Unknown Source)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
    at java.lang.Thread.run(Unknown Source)
Caused by: java.net.UnknownHostException: www.uni-passauf.de
    at java.net.InetAddressImpl.lookupAllHostAddr(Native Method)
    at java.net.InetAddress$2.lookupAllHostAddr(Unknown Source)
    at java.net.InetAddress.getAddressesFromNameService(Unknown Source)
    at java.net.InetAddress.getAllByName0(Unknown Source)
    at java.net.InetAddress.getAllByName(Unknown Source)
    at java.net.InetAddressImpl.resolve(SystemDefaultDnsResolver.java:45)
    at org.apache.http.impl.conn.SystemDefaultDnsResolver.resolve(SystemDefaultDnsResolver.java:45)
    at org.apache.http.impl.conn.DefaultHttpClientConnectionOperator.connect(DefaultHttpClientConnectionOperator.java:112)
    at org.apache.http.impl.conn.PoolingHttpClientConnectionManager.connect(PoolingHttpClientConnectionManager.java:374)
    at org.apache.http.impl.execchain.MainClientExec.establishRoute(MainClientExec.java:393)
    at org.apache.http.impl.execchain.MainClientExec.execute(MainClientExec.java:236)
    at org.apache.http.impl.execchain.ProtocolExec.execute(ProtocolExec.java:185)
    at org.apache.http.impl.execchain.RetryExec.execute(RetryExec.java:89)
    at org.apache.http.impl.execchain.RedirectExec.execute(RedirectExec.java:110)
    at org.apache.http.impl.client.InternalHttpClient.doExecute(InternalHttpClient.java:185)
    at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:83)
    at org.apache.http.impl.client.CloseableHttpClient.execute(CloseableHttpClient.java:108)
    at wg.requests.HttpRequest.executeSingleRequest(HttpRequest.java:118)
    ... 6 more
```

Abbildung 5.3: Ausgabe der Anwendung bei einem ungültigen Konfigurationsparameter

### Beurteilung

Anders als bei einem Syntax- oder Semantikfehler wird der Fehler erst bei der Ausführung des Requests bemerkt, und nicht schon während dem Parsen. Dennoch stürzt die Anwendung beim Auftreten des Fehlers nicht unkontrolliert ab, sondern gibt eine Fehlermeldung aus und beendet sich im Anschluss. Die Fehlermeldung ist aussagekräftig genug, um den Fehler zu verstehen und zu beheben.

Die in diesem Abschnitt abgebildeten Nutzerszenarien beinhalten nicht alle mögli-



chen Fehlerfälle. Es gibt eine Vielzahl an Varianten von Syntax- oder Semantikfehlern, die in der Konfigurationsdatei enthalten sein können. Die Resultate der hier untersuchten Nutzerszenarien zeigen aber, dass sowohl Syntax- als auch Semantikfehler das Programm nicht unkontrolliert fehlschlagen lassen und eine sinnvolle Fehlerrückmeldung an den Nutzer stattfindet. Eine vollständige Immunität der Anwendung gegen Fehler in der Konfigurationsdatei kann nicht gewährleistet werden.

### 5.2 Korrektheit der Ausführung

In diesem Abschnitt wird validiert, ob der Workload so ausgeführt wird, wie es in der Konfigurationsdatei vom Nutzer festgelegt wurde. Dafür wird zuerst geprüft, ob die Requests sowohl in ihrer Anzahl als auch in ihrem Inhalt richtig an die Server ausgeliefert werden. Anschließend wird verifiziert, ob der definierte Zeitplan eingehalten und die Events zum richtigen Zeitpunkt ausgeführt werden. Wie auch im vorherigen Abschnitt erfolgt die Prüfung mittels Simulationen von Nutzerszenarien.

- **Korrektheit der ausgelieferten Requests**

#### **Ziel**

Es soll geprüft werden, ob die Requests korrekt an die Server übertragen werden. Korrekt bedeutet dabei, dass der Inhalt der Konfiguration entspricht und die Anzahl der gesendeten Requests mit den festgelegten Werten übereinstimmt.

#### **Beschreibung**

Ein Nutzer möchte einen BFT-SMaRt Request an vier Server schicken, mit dem er den String Wert „Hello World“ in die vom Server geführte Map abspeichern kann. Dafür spezifiziert er in der Konfigurationsdatei das BFT-SMaRt Command als *ObjectOutputStream*. Auf diesen schreibt er den Enum-Typ *PUT* (signalisiert den Servern, dass ein Wert in die Map gespeichert werden soll), den Key, unter dem sein Wert gespeichert werden soll: „example“, und den Wert selbst: „Hello World“. Um zu prüfen, ob der Wert korrekt abgespeichert wurde, definiert der Nutzer einen zweiten Request. Dieser beinhaltet den Enum-Typ *GET* sowie den Key, unter dem der Wert abgespeichert sein soll: „example“.

Die Server sind eine modifizierte Version der in der BFT-SMaRt Bibliothek enthaltenen Demo *MapServer*. Die Modifikation beinhaltet eine einfache Konsolenausgabe der vom Server empfangenen Werte. Diese Ausgabe erfolgt vor der eigentlichen Ausführung. Somit wird verhindert, dass die Werte in irgendeiner Weise verändert werden.

#### **Ausgabe**

Abbildung 5.4 zeigt die Ausgabe eines Servers (die Ausgabe aller vier angesteuerten Server ist gleich) und die Ergebnisausgabe der Anwendung.

Aus Gründen der Übersicht wurden die vom Server erzeugten Log-Einträge zwischen den beiden Ausgaben nicht mit abgebildet. Die Ausgabe der Server deckt sich mit den in der Konfigurationsdatei festgelegten Werten. Es wurden zweimal Werte empfangen: zuerst die des PUT Requests, anschließend die des GET Requests. Die Ergebnisausgabe der Anwendung zeigt die Auswertung der zwei Requests. Die in der Abbildung 4.4 dargestellten *Execution times* sind vom System, auf dem die Anwendung läuft, abhängig, und somit nicht aussagekräftig. Für den ersten Request wurde eine leere Antwort empfangen, die Antwort des zweiten Requests enthält den String „Hello World“.

<pre>Received Values: -Object 1: PUT -Object 2: example -Object 3: Hello World  Received Values: -Object 1: GET -Object 2: example</pre>	<pre>--- Results per request --- Request: 1   Number of targets: 4   Execution time (in milliseconds): 730   Reply length: 0   Reply: Request: 2   Number of targets: 4   Execution time (in milliseconds): 439   Reply length: 18   Reply: t         Hello World</pre>
------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Abbildung 5.4: Serverseitige Ausgabe der empfangenen Objekte (links) und Ergebnisausgabe der Anwendung (rechts)

### Beurteilung

Die von den Servern empfangenen Werte stimmen mit den der Konfigurationsdatei überein. Somit wurden die Werte korrekt von der Anwendung an die Server übertragen. Bei der Auswertung der Ergebnisse fällt auf, dass für den ersten Request eine leere Antwort angezeigt wird (*Reply length: 0*). Dies liegt daran, dass die Implementierung der BFT-SMaRt Server für einen PUT Request nur dann keine leere Antwort zurückgibt, wenn für den entsprechenden Key bereits ein Eintrag existiert. Wird ein neuer Eintrag für den Key angelegt, wird ein leeres Byte Array zurückgeschickt. Da der in diesem Szenario gesendete PUT Request der erste Request an die Server war, lag kein Eintrag für den Key „example“ vor. Somit lässt sich die leere Antwort auf die Implementierung der Server zurückführen und die Ausgabe der Antwort ist korrekt. Für den zweiten Request wird der Wert „Hello World“ ausgegeben. Dies zeigt, dass der GET Request sowie die daraus resultierende Antwort mit dem Wert, korrekt versendet bzw. empfangen und dargestellt wurden.

- **Einhaltung des Zeitplans**

### Ziel

Es soll geprüft werden, ob die Anwendung die Requests zu den vom Nutzer festgelegten Zeitpunkten ausführt, und dabei die definierten Optionen einhält.

### Beschreibung

Ein Nutzer möchte einen einfachen Lasttest für die Website der Universität Passau durchführen. Dafür definiert er einen HTTP GET Request mit einem Client. Den Zeitpunkt der Ausführung des Events mit dem Namen *UniPassauWebsite* setzt er auf eine Sekunde nach Start des Frames. Um die Anzahl der Requests an die Website zu erhöhen, legt er fünf Durchgänge für das Frame fest. Für den Wachstumsmodus wählt er die Option *fibonacci*, für die Frequenz einen Anstieg mit dem Faktor zwei.

### Ausgabe

Abbildung 5.5 zeigt die Ausgabe der Anwendung. Um die Korrektheit der zeitlichen Ausführung zu prüfen, wurde das Log-Level auf *debug* gesetzt. Dies sorgt dafür, dass Debug Informationen ausgegeben werden. Diese Debug Informationen halten fest, welches Event in welcher Iterationsstufe zu welchem Zeitpunkt (in Millisekunden) nach dem Start des Frames ausgeführt wurde. Es ist zu sehen, dass insgesamt fünf Durchgänge mit insgesamt 12 Requests für das Frame ausgeführt wurden.

```
19:22:13.255 [main] INFO wg.App - Start workload Generator
19:22:14.772 [main] DEBUG wg.executor.Executor - Iteration: 1, Event: UniPassauWebsite, executed at: 1002
19:22:15.286 [main] DEBUG wg.executor.Executor - Iteration: 2, Event: UniPassauWebsite, executed at: 514
19:22:15.536 [main] DEBUG wg.executor.Executor - Iteration: 3, Event: UniPassauWebsite, executed at: 250
19:22:15.536 [main] DEBUG wg.executor.Executor - Iteration: 3, Event: UniPassauWebsite, executed at: 250
19:22:15.661 [main] DEBUG wg.executor.Executor - Iteration: 4, Event: UniPassauWebsite, executed at: 125
19:22:15.661 [main] DEBUG wg.executor.Executor - Iteration: 4, Event: UniPassauWebsite, executed at: 125
19:22:15.661 [main] DEBUG wg.executor.Executor - Iteration: 4, Event: UniPassauWebsite, executed at: 125
19:22:15.724 [main] DEBUG wg.executor.Executor - Iteration: 5, Event: UniPassauWebsite, executed at: 63
19:22:15.724 [main] DEBUG wg.executor.Executor - Iteration: 5, Event: UniPassauWebsite, executed at: 63
19:22:15.724 [main] DEBUG wg.executor.Executor - Iteration: 5, Event: UniPassauWebsite, executed at: 63
19:22:15.724 [main] DEBUG wg.executor.Executor - Iteration: 5, Event: UniPassauWebsite, executed at: 63
19:22:15.724 [main] DEBUG wg.executor.Executor - Iteration: 5, Event: UniPassauWebsite, executed at: 63
```

Abbildung 5.5: Ausgabe der Anwendung bei eingeschaltetem Debug Modus zur Überprüfung der Einhaltung des Zeitplans

### Beurteilung

Die Anwendung hat wie vom Nutzer konfiguriert fünf Wiederholungen des Frames ausgeführt.

Für den Wachstum nach Fibonacci entspricht die Anzahl an ausgeführten Requests der Summe der beiden vorherigen Durchgänge. Somit sollte für die fünf Durchgänge folgende Anzahl an Requests ausgeführt werden:

1. Iteration: 1
2. Iteration: 1
3. Iteration: 2 (1+1)
4. Iteration: 3 (1+2)
5. Iteration: 5 (2+3)

Die Anzahl der ausgeführten Requests stimmt mit diesen Zahlen überein. Somit wurde sowohl die Anzahl an Wiederholungen, als auch der gewählte Wachstumsmodus korrekt ausgeführt.

Für die Frequenz wurde ein Anstieg mit dem Faktor 2 festgelegt. Somit sollten die Events für die fünf Durchgänge zu folgenden Zeitpunkten ausgeführt werden (die Zahl entspricht der Anzahl an Millisekunden seit dem Start des aktuellen Durchgang des Frames):

1. Iteration: 1000
2. Iteration: 500
3. Iteration: 250
4. Iteration: 125
5. Iteration: 62,5

Die von der Anwendung enthaltenen Werte haben teils eine geringe Abweichung von diesen Zahlen. Das ist damit zu erklären, dass die Anwendung nur alle zehn Millisekunden prüft, ob ein Event ausgeführt werden muss. Somit sind Abweichungen von zehn Millisekunden vertretbar. Da alle Werte innerhalb dieser Abweichung liegen, wurden die Requests nach dem vom Nutzer festgelegten Zeitplan korrekt ausgeführt.

### 5.3 Erweiterbarkeit

Eine der in Kapitel 4.1 genannten Anforderungen war die Erweiterbarkeit der Anwendung. Dabei geht es neben der Erweiterung der Anwendung um neue Protokolle auch um die Modifikation von Bestehenden.

Ein Beispiel dafür ist die Erweiterung der BFT-SMaRt Requests um Microbenchmarks. Bei Microbenchmarks handelt es sich um Performanz Messungen der Server. Wie dies im Zusammenhang mit BFT-SMaRt aussehen kann, zeigt die in der Bibliothek enthaltene Demo Anwendung. Diese implementiert eine BFT-SMaRt Anwendung, die nach einer bestimmten Anzahl an ausgeführten Operationen bestimmte Leistungsparameter, wie z.B. die Latenz und den Durchsatz, auf Seiten des Servers ausgeben lässt. Um eine solche Performanz Messung mit einer automatisierten Request-Erzeugung zu verknüpfen, könnte der Workload Generator um einen *BFT-SMaRt Microbenchmark* Request erweitert werden. Dafür müssten folgende Schritte getätigt werden:

## 5 Validierung

1. Hinzufügen einer BFT-SMaRt Microbenchmark Request Klasse:  
Diese Klasse muss die abstrakte Klasse *Request* erweitern und somit eine *call* Methode bereitstellen, die die Ausführung des Requests regelt. Parameter des Requests müssen als Attribute definiert und im Konstruktor aus dem übergebenen JSON Objekt ausgelesen werden.
2. Eintragen des Protokolls in das Enum *ProtocolType*
3. Hinzufügen einer BFT-SMaRt Microbenchmark Response Klasse:  
Die Klasse erweitert die abstrakte Klasse *Response* und implementiert deren *print* Methode. Mittels dieser kann definiert werden, wie die Ausgabe des Requests für die Detailauswertung aussieht .

Die drei aufgezählten Schritte gelten nicht nur für das Microbenchmark Beispiel, sondern sind für die Erweiterung um ein beliebiges Protokoll gültig. Die Kernfunktionalität der Anwendung ist durch die beiden abstrakten Klassen *Request* und *Response* unabhängig von den Spezifikationen der Protokolle und ermöglicht somit das Erweitern der Anwendung um neue Protokolle in nur drei Schritten.

In dieser Arbeit wurde eine Anwendung zum automatisierten Erzeugen von spezifizierbaren Client Anfragen für das Testen von Netzwerkanwendungen vorgestellt. Die Anwendung ist eine Antwort auf das Problem der Request Erzeugung bei der Ausführung von Lasttests. Eine manuelle Erzeugung von Requests wäre sehr zeintensiv und würde den Anforderungen von Lasttests nicht gerecht werden. Somit wird eine Anwendung benötigt, die Client Anfragen entsprechend einer vorgegebenen Konfiguration erstellt und ausliefert. Bei der Analyse wissenschaftlicher Ansätze und kommerzieller Anwendung für das Erzeugen von Client Anfragen zum Testen von Netzwerkanwendungen, zeigt sich, dass es viele Ansätze und Anwendungen gibt, die sich mit dem Problem befassen. Allerdings gibt es noch keine Anwendung, die ein automatisiertes Erzeugen von BFT-SMaRt Requests ermöglicht.

Der in dieser Arbeit vorgestellte Workload Generator ermöglicht es einem Nutzer, Client Anfragen basierend auf den Protokollen: HTTP, FTP, TCP, UDP und für BFT-SMaRt Anwendungen zu erzeugen. Dafür erstellt der Nutzer eine Konfigurationsdatei, in der die Ziele der Requests genannt, die Requests definiert und spezifiziert werden, und der Zeitplan der Ausführung bestimmt wird. Die Konfigurationsdatei wird im JSON Format erstellt und anschließend zur Ausführung an die Anwendung übergeben. Die Anwendung prüft die Korrektheit und Vollständigkeit der Konfigurationsdatei und gibt im Fehlerfall eine Meldung an den Nutzer zurück. Verläuft die Prüfung positiv, wird der in der Konfigurationsdatei beschriebene Workload ausgeführt. Die Ausführung des Workloads ist unabhängig von protokollspezifischen Informationen. Somit kann die Anwendung bei Bedarf unkompliziert um weitere Protokolle erweitert werden. Bei der Ausführung werden die definierten Requests zu den vom Nutzer festgelegten Zeitpunkten an die definierten Server geschickt. Dabei werden die vom Nutzer bestimmten Optionen für die Spezifizierung berücksichtigt. Dies ermöglicht, dass sich die Anzahl und Frequenz der Ausführung von Requests zur Laufzeit verändert. Dies ermöglicht das Simulieren bestimmter Anwendungsszenarien, wie z.B. das für einen Lasttest notwendige, schrittweise Erhöhen der Anzahl an Requests. Nach Beendigung der Ausführung des Workloads erhält der Nutzer eine

Auswertung der Ergebnisse in Form einer Konsolenausgabe.

Die Validierung der Anwendung zeigt, dass das Programm in der Lage ist, syntaktische und semantische Fehler in der Konfiguration des Workloads zu erkennen und eine aussagekräftige Fehlermeldung auszugeben. Dies erhöht die Benutzerfreundlichkeit der Anwendung. Auch Fehler bei der Ausführung des Workloads (z.B. keine Erreichbarkeit der Server) werden von der Anwendung in eine sinnvolle Fehlermeldung umgesetzt. Weiter hat die Validierung gezeigt, dass die Requests auch so an die Server ausgeliefert werden, wie dies in der Konfigurationsdatei festgehalten wurde. Abschließend hat die Validierung anhand eines einfachen Beispiels gezeigt, dass die Anwendung unkompliziert um weitere Protokolle erweiterbar ist.

Setzt sich der in der Einleitung beschriebene Trend des stetig wachsenden Internets fort, wird das Themenfeld der Testautomatisierung weiter an Bedeutung gewinnen. Die sich ändernden Rahmenbedingungen des Internets, die auf soziale (z.B. mehr Menschen haben Zugang zum Internet), ökonomische (Digitalisierung in vielen Wirtschaftszweigen wie zum Beispiel der Landwirtschaft) und technische Gründe (z.B. technische Fortschritte im Bereich der künstlichen Intelligenz) zurückzuführen sind, werden dafür sorgen, dass neue Kommunikationsprotokolle für Netzwerkanwendungen benötigt werden. Ein Beispiel dafür ist das von Google entwickelte QUIC (*Quick UDP Internet Connection*) Protokoll, welches den Zugriff auf Websites beschleunigen soll. Die aus solchen Protokollen entstehenden Anwendungen müssen ebenfalls sorgfältig getestet werden, um ihre Zuverlässigkeit im Normalbetrieb zu gewährleisten. Dafür werden in der Zukunft neue Programme und Ansätze entwickelt werden müssen, die die Automatisierung von Tests für solche Anwendungen unterstützen.



## Anhang

Eine DVD ist dieser Arbeit angehängt, die sowohl den Quellcode des vorgestellten Workload Generators, als auch die Arbeit in digitaler Form enthält.  
Die Ordnerstruktur der DVD sieht wie folgt aus:

**Code** Das Wurzelverzeichnis des Maven Projekts

**Code/src** Das Wurzelverzeichnis des Quellcodes

**WorkloadGenerator** Die ausführbare Java-Datei des Programmes

**Bachelorarbeit** Die digitale Version dieser Arbeit

**Bachelorarbeit-Latex** Das Latex Verzeichnis der Bachelorarbeit



## Abbildungsverzeichnis

3.1	Ausführung eines Experiments mit Emusphere [14]	9
3.2	Kommunikationsprozess in BFT-SMaRt [15]	10
3.3	Referenzmodelle: Darstellung des ISO/OSI Schichtenmodells [17] (links) und des TCP/IP Schichtenmodells (rechts) [17]	12
4.1	Ablauf der Anwendung	20
4.2	Grundgerüst der Konfigurationsdatei	22
4.3	Aufbau der Komponente <i>targetGroups</i> für das <i>Map Beispiel</i>	23
4.4	Aufbau der Komponente <i>requests</i> für das <i>Map Beispiel</i>	26
4.5	Überblick über die drei Wachstumsmodi (Der lineare Wachstumsfaktor hat in diesem Beispiel den Wert 9)	30
4.6	Aufbau der Komponente <i>schedule</i> für das <i>Map Beispiel</i> . Die gezeigten Optionen sind die Standard-Optionen eines Frames.	31
4.7	Klassenstruktur eines Workloads (Die Klassen, die zur Struktur des Workloads gehören, sind blau eingefärbt)	32
4.8	Beispielhafter Ablauf der Ausführung eines Workloads mit zwei Requests	34
4.9	Beispielhafte Konsolenausgabe für das erfolgreiche Ausführen eines Workloads	38
5.1	Ausgabe der Anwendung bei Syntaxfehlern in der Konfigurationsdatei	41
5.2	Ausgabe der Anwendung bei einem Semantikfehler in der Konfigurationsdatei	42
5.3	Ausgabe der Anwendung bei einem ungültigen Konfigurationsparameter	43
5.4	Serverseitige Ausgabe der empfangenen Objekte (links) und Ergebnisausgabe der Anwendung (rechts)	45
5.5	Ausgabe der Anwendung bei eingeschaltetem Debug Modus zur Überprüfung der Einhaltung des Zeitplans	46

## Tabellenverzeichnis

3.1	Übersicht über bekannte Protokolle der Anwendungsschicht (aufgeteilt nach ihren Anwendungsbereichen mit Angabe des zugrundeliegenden Protokolls der Transportschicht). . . . .	16
3.2	Übersicht über die Gruppen von Statuscodes einer HTTP Antwort . . . . .	18
4.1	Protokollunabhängige Parameter eines Request . . . . .	24
4.2	Parameter eines HTTP Request . . . . .	24
4.3	Parameter eines TCP/UDP Request . . . . .	24
4.4	Parameter eines FTP Request . . . . .	25
4.5	Parameter eines BFT-SMaRt Request . . . . .	25
4.6	Parameter eines Event-Objekts . . . . .	28
4.7	Übersicht über die Optionen eines Frames . . . . .	28

## Literaturverzeichnis

- [1] LANG, Susanne: Eine kurze Geschichte des Internets. In: *Zeitschrift für kritische Sozialwissenschaft* (2017), S. 15
- [2] MOSBERGER, David ; JIN, Tai: httpperf—a tool for measuring web server performance. In: *ACM SIGMETRICS Performance Evaluation Review* 26 (1998), Nr. 3, S. 31–37
- [3] KANT, Krishna ; WON, Youjip: Performance impact of uncached file accesses in specweb99. In: *Workload Characterization for Computer System Design*. Springer, 2000, S. 87–104
- [4] TRENT, Gene ; SAKE, Mark: *WebSTONE: The first generation in HTTP server benchmarking*. 1995
- [5] SHARMA, Monika ; IYER, Vaishnavi S. ; SUBRAMANIAN, Sugandhi ; SHETTY, Abhinandhan: *Comparison of Load Testing Tools*. 2016
- [6] GLOBE NEWSWIRE: *WebLOAD Adds Real User Experience to Performance Testing*. <https://www.wallstreet-online.de/nachricht/8030091-webload-adds-real-user-experience-to-performance-testing>. Version: Okt 2015. – Aufgerufen am: 14.11.2018
- [7] GAROUSHI, Vahid ; BRIAND, Lionel C. ; LABICHE, Yvan: Traffic-aware stress testing of distributed systems based on UML models. In: *Proceedings of the 28th international conference on Software engineering* ACM, 2006, S. 391–400
- [8] ZHANG, Jian ; CHEUNG, Shing C.: Automated test case generation for the stress testing of multimedia systems. In: *Software: Practice and Experience* 32 (2002), Nr. 15, S. 1411–1435

- [9] AVRITZER, Alberto ; WEYUKER, ER: The automatic generation of load test suites and the assessment of the resulting software. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 9, S. 705–716
- [10] SHEN, Du ; LUO, Qi ; POSHYVANYK, Denys ; GRECHANIK, Mark: Automating performance bottleneck detection using search-based application profiling. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis* ACM, 2015, S. 270–281
- [11] BRIAND, Lionel C. ; LABICHE, Yvan ; SHOUSHA, Marwa: Stress testing real-time systems with genetic algorithms. In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation* ACM, 2005, S. 1021–1028
- [12] IQBAL, Muhammad Z. ; ARCURI, Andrea ; BRIAND, Lionel: Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis* ACM, 2012, S. 199–209
- [13] JIANG, Zhen M. ; HASSAN, Ahmed E. ; HAMANN, Gilbert ; FLORA, Parminder: Automated performance analysis of load tests. In: *2009 IEEE International Conference on Software Maintenance* IEEE, 2009, S. 125–134
- [14] KÖSTLER, Johannes ; SEIDEMANN, Jan ; REISER, Hans P.: Emusphere: Evaluating Planetary-Scale Distributed Systems in Automated Emulation Environments. In: *Reliable Distributed Systems Workshops (SRDSW), 2016 IEEE 35th Symposium on* IEEE, 2016, S. 49–54
- [15] SOUSA, João ; ALCHIERI, Eduardo ; BESSANI, Alysson: State machine replication for the masses with BFT-SMaRt / University of Lisbon. 2013. – Forschungsbericht
- [16] LAMPORT, Leslie ; SHOSTAK, Robert ; PEASE, Marshall: The Byzantine generals problem. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4 (1982), Nr. 3, S. 382–401
- [17] *Kapitel 1.* In: KHAN, Rukhsar: *Netzwerktechnik, Band 1.* AIRNET Technologie- und Bildungszentrum GmbH, 2009, S. 31
- [18] FACCHI, Christian: *Methodik zur formalen Spezifikation des ISO-OSI-Schichtenmodells*, Technische Universität München, Diss., 1995
- [19] PROEBSTER, Walter: *Rechnernetze: Technik, Protokolle, Systeme, Anwendungen.* Walter de Gruyter GmbH & Co KG, 2015
- [20] *Kapitel 12.* In: COMER, Douglas: *Computernetzwerke und Internets.* Prentice Hall, 1998, S. 209
- [21] HOLTkamp, Heiko: Einführung in TCP/IP / Universität Bielefeld. 1997. – Forschungsbericht

- [22] *Kapitel 1.* In: BADACH, Erwin Anatol und H. Anatol und Hoffmann: *Technik der IP-Netze*. Carl Hanser Verlag, 2001, S. 20
- [23] BERNERS-LEE, Tim ; FIELDING, Roy ; FRYSTYK, Henrik: Hypertext transfer protocol-HTTP/1.0. 1996. – Forschungsbericht
- [24] FIELDING, Roy ; GETTYS, Jim ; MOGUL, Jeff ; FRYSTYK, Henrik ; MASINTER, Larry ; LEACH, Paul ; BERNERS-LEE, Tim: RFC 2616: Hypertext transfer protocol-HTTP/1.1. 1999. – Forschungsbericht
- [25] *Kapitel 16.* In: TISCHER, Bruno Michael und J. Michael und Jennrich: *Internet intern: Technik & Programmierung*. Data Becker, 1997, S. 922
- [26] BHUSHAN, A: RFC-114: File Transfer Protocol. 1971. – Forschungsbericht
- [27] POSTEL, Jon ; REYNOLDS, Joyce: Rfc 959: File transfer protocol (ftp). 1985. – Forschungsbericht