Hochschule Wismar
University of Applied Sciences
Technology, Business and Design
Fakultät für Ingenieurwissenschaften, Bereich EuI

Masterthesis

Bestimmung der Skalierungsgrenzen von verteilten Applikationen mithilfe eines DES-Frameworks

von: Patrice Matz
geboren am 17.07.1997
in Neubrandenburg

Gutachter: Prof. Dr.-Ing. Matthias Kreuseler
Prof. Dr.-Ing. Sven Pawletta
betrieblicher Betreuer: B. Sc. Oliver Roggelin

Zusammenfassung

Verteilte Applikationen können aus hunderten unabhängig voneinander skalierenden Diensten bestehen. Die Zusammenhänge zwischen diesen Diensten sind oft unüberschaubar. Das ist problematisch, denn ein nicht oder langsam skalierender Dienst reicht aus, um die Nutzeranzahl der gesamten Applikation zu begrenzen. Es ist schwierig und mit hohen Kosten verbunden, begrenzende Komponenten zu identifizieren bevor sie einen negativen Einfluss auf die Nutzererfahrung haben. In dieser Thesis werden Erkenntnisse vorangegangener Arbeiten zu Reverse Engineering mittels Aspect Oriented Programming (AOP), Queueing Theory und Discrete Event Simulation (DES) kombiniert, um verteilte Applikation automatisch nachzubilden und zu simulieren. Hierzu wurden programmiersprachenunabhängige Konzepte und generische Implementierungen für einen Profiler und eine Simulation entwickelt. Während der Verifizierung zeigte sich, dass die Messung der Ressourcenauslastung nicht ausreichend zuverlässig ist, um das Verhalten der Applikation nachzuverfolgen. Dennoch konnte die Struktur von Funktionsaufrufen erfasst und Abhängigkeiten erkannt werden. Die Simulation wurde mit einer eigens entwickelten verteilten Beispielapplikation verifiziert. Die durchgeführten Tests legen nahe, dass die maximale Nutzeranzahl und die Wartezeit akkurat simuliert werden können.

Abstract

Distributed applications can consist of hundreds of independently scaling services. The interconnections between these services are often unwieldy. This is problematic since one service that scales too slowly or not at all is enough to limit the number of users for the entire application. Identifying limiting components before they have a negative impact on the user experience is both difficult and costly. In this thesis, insights from previous work on reverse engineering using AOP, queueing theory, and DES are combined to automatically replicate and simulate distributed application. For this purpose, programming language-independent concepts and generic implementations for a profiler and a simulation were developed. During the verification, it became apparent that the method of measuring resource utilization is not sufficiently reliable to track the behavior of the application. However, it was still possible to understand the structure of function calls and identify dependencies. The simulation was verified with a custom-developed distributed demo application. Tests suggest that the maximum number of users and the waiting time can be accurately simulated.

Inhaltsverzeichnis

1 Einleitung **6**
1.1 Motivation . . . 6
1.2 Kontext . . . 8
1.3 Zielstellung . . . 9
1.4 Struktur der Arbeit . . . 9

2 Grundlagen **10**
2.1 Simulationen . . . 10
2.1.1 Überblick . . . 11
2.1.2 Discrete Event Simulation . . . 12
2.2 Applikations Profiler . . . 13
2.2.1 Dynamische Profiler . . . 13
2.2.2 Statische Profiler . . . 14
2.3 Aspektorientierte Programmierung . . . 15
2.4 Queueing Theory . . . 16

3 Stand der Forschung **19**
3.1 MONARC . . . 19
3.2 Improving Dynamic Data Analysis with Aspect-Oriented Programming 21
3.3 Tracing mit AspectJ in verteilten Systemen . . . 23

4 Anforderungen **24**

5 Grobkonzept **25**
5.1 Profiler . . . 26
5.2 Simulation . . . 26
5.3 Programmiersprachen und Simulationsframeworks . . . 27

6 Feinkonzept **30**
6.1 Profiler . . . 31
6.2 Modell . . . 32
6.3 Begriffsdefinition . . . 34
6.3.1 Profiler . . . 34
6.3.2 Profil . . . 35
6.3.3 Szenario . . . 35
6.3.4 Interaktion . . . 35
6.3.5 Funktion . . . 35
6.3.6 Funktion-Server-Mapping . . . 36
6.3.7 Szenarioverteilung . . . 36

6.4 Design Entscheidungen . . . 36
6.4.1 Skalierung . . . 36
6.4.2 Messung der CPU Auslastung . . . 37

7 Profiler **39**
7.1 Datensammlung . . . 39
7.1.1 Java Aspect . . . 40
7.1.2 Systemdaten . . . 41
7.2 Datentransformation . . . 41
7.2.1 Call-Tree . . . 41
7.2.2 Validierung . . . 45
7.2.3 Messen des Nutzerverhaltens . . . 45
7.2.4 Erfassen von RPC . . . 45
7.2.5 Profilgeneration . . . 47

8 Simulation **50**
8.1 Modell . . . 50
8.2 Inputs . . . 51
8.2.1 Profil . . . 51
8.2.2 Mapping . . . 53
8.2.3 Service Definition . . . 53
8.2.4 Verteilungsgeneration . . . 54
8.3 Objekte . . . 54
8.3.1 Event . . . 54
8.3.2 Funktion . . . 55
8.3.3 Service . . . 56
8.3.4 Server . . . 56
8.4 Simulation-Engine . . . 57
8.5 Monitoring . . . 58
8.6 Visualisierung . . . 59
8.6.1 Visualisierungsziele . . . 59
8.6.2 Visualisierungstechniken . . . 60
8.6.3 Implementierung . . . 62
8.6.4 Integration der Simulation . . . 64

9 Verifizierung und Validierung **65**
9.1 Queueing Theory . . . 65
9.2 Validierung des Profilers . . . 67
9.3 Rekonstruktion einer Messung . . . 68
9.4 Konstruiertes Beispiel . . . 70
9.5 Erfüllung der Anforderungen . . . 73
9.6 Auswertung . . . 74

10 Fazit und Ausblick **76**

Literaturverzeichnis **77**

Abbildungsverzeichnis **82**

Code Listings **83**

A Anhang **84**
A.1 Experteninterview 84

Selbstständigkeitserklärung **90**

1 Einleitung

Microservices erfreuen sich steigender Akzeptanz und Verbreitung, einer Umfrage von F5, Inc.[1] nach verwendeten 60 % aller Befragten im Jahr 2020 Microservices, 2019 lag der Anteil noch bei 40%.[1] Microservices bieten eine Reihe von Vorteilen gegenüber anderen Softwarearchitekturen. Befragte in einer Umfrage von RedHat sahen diese unter anderem in verbesserter CI/CD, einfacherer Wartung, schnellerer „time to market“ und besserer Skalierung.[2]

Besonders die Skalierung ist abhängig von Services[2] im Umfeld einer Applikation[3]. Hat ein Service Abhängigkeiten zu einer Applikation, die nicht skaliert, ist die Skalierbarkeit des Services selbst eingeschränkt, ohne dass dies offensichtlich ist. Bei einer Neuentwicklung sollte dieser Fall bereits in der Planung vermieden werden. Bei historisch gewachsenen Anwendung kann es aber im Laufe der Entwicklung zu solchen Fällen kommen. Handelt es sich bei den begrenzenden Applikationen um Closed-Source Applikationen oder externen Services, kann es sehr schwer sein die maximale Performance der Applikation zu bestimmen, bevor sie im produktiven Betrieb erreicht wird.

Dieses Problem kann bei allen verteilten Applikationen auftreten, bei Microservice basierten Applikationen im Besonderen kommt es durch die Vielzahl von Services schnell dazu, dass die Abhängigkeiten einer Applikation unüberschaubar werden.

1.1 Motivation

Verteilte Applikationen im allgemeinen und Microservices im speziellen, können schwer nachvollziehbare Abhängigkeiten besitzen. Das ist ein Problem, wenn die maximale Nutzeranzahl eines Services bestimmt werden soll.

[1] F5, Inc. entwickelt NGINX, ein High Performance Load Balancer und Web Server

[2] In dieser Arbeit werden Web-Server und Web-Services als Services bezeichnet. Ein Service kann ein Teil eines größeren verteilten Services sein und aus weiteren Services bestehen.

[3] Um den untersuchten Service von Services im Umfeld zu unterscheiden, wird dieser als Applikation bezeichnet.

Lasttests können eine Antwort bieten. Durch Lasttests gegen alle Services und Abhängigkeiten einer Applikation können erhebliche Kosten und organisatorische Aufwände entstehen. Lasttests sollten nie gegen ein produktives System durchgeführt werden, daher muss ein weiteres Deployment auf äquivalenter Hardware allein für diese Tests bereitgestellt werden. Zusätzlich wären diese Tests sehr unflexibel, jede Konfigurationsänderung würde ein neues Deployment des Softwaresystems erfordern. Theoretische Optimierungen oder „Was wäre, wenn" - Fragen könnten mit Lasttests nicht beantwortet werden.

Ist nur der begrenzende Dienst Ziel der Lasttests ergibt sich ein neues Problem. Mit umfangreicher Kenntnis der Plattform und des Nutzerverhaltens könnte ein erfahrener Entwickler eventuell eine grobe Abschätzung treffen wie viele Anfragen ein Nutzer durchschnittlich in einem gewissen Zeitraum gegen einen dieser begrenzenden Dienste verursacht. Die Zahl der gleichzeitigen Nutzer die hieraus hervorgeht ist aber mit großer Unsicherheit und großem Aufwand verbunden und kann höchstens die Größenordnung der gleichzeitigen Nutzer vorgeben.

Die Einbindung externer Schnittstellen kann die Skalierung einer Applikation ebenfalls begrenzen. Im Gegensatz zu proprietären, aber selbst gehosteten Programmen, sind Lastests hier oft keine Option. Lasttests gegen Schnittstellen würde ein hohes Maß an Koordination mit den verantwortlichen Organisationen erfordern. So entstehen Aufwände und Kosten auf beiden Seiten, bei einer geringen Anzahl von Schnittstellen wäre dieser Weg unter Umständen gangbar, andernfalls ist es schwer den Aufwand zu rechtfertigen. Finanzielle, zeitliche und weitere Aufwände erschweren Untersuchungen der Skalierungsgrenzen eines Systems und können dazu führen, dass diese nicht durchgeführt werden.

Eine Simulation des Systems könnte diese Aufwände erheblich reduzieren und gleichzeitig eine Menge weiterer Möglichkeiten bieten.

1.2 Kontext

Diese Thesis entstand in Kooperation mit der DVZ Datenverarbeitungszentrum Mecklenburg-Vorpommern GmbH am Beispiel des MV-Serviceportals. Das MVSP implementiert das OZG[4] und ist eine Plattform für die BürgerInnen des Landes MV, welche aber auch von BürgerInnen anderer Bundesländer nutzbar ist. Die potenzielle Nutzermenge entspricht somit der Bevölkerung Deutschlands. Daher ist die Skalierungsgrenze des MVSP sehr relevant. Das Framework, welches im Rahmen dieser Arbeit entsteht, wird als Entscheidungsunterstützung bei der Migration des MVSP zu einer Microservice basierten Architektur entwickelt.

Das MVSP besteht aus 3 Komponenten und hat mindestens 9 weitere Abhängigkeiten zu weiteren Diensten. In der Zukunft könnte in extremen Fällen aber pro Antrag ein weiterer externer Dienst benötigt werden. Die drei Komponenten sind: das Servicekonto, das Formularmanagementsystem und die elektronsiche Antragstellung. Jede dieser Komponenten nutzt eine eigene Datenbank. Die elektronische Antragstellung ist der Kern des MVSP und bündelt einen Großteil der Funktionalität. Das Servicekonto ist der Identity Provider, es verifiziert Nutzer und speichert personenbezogene Daten. Das Formularmanagementsystem verwaltet HTML-Formulare und stellt diese der elektronischen Antragstellung zur Verfügung. In Quelle [4] wird genauer auf die Komponenten und Funktionsweise des MVSP eingegangen. Das die eletronische Antragstellung wurde in Java implementiert, daher wird die Implementierung der erarbeiteten Konzepte in Java fokusiert.

[4]Das Online Zugangsgesetz (OZG) wurde im Jahr 2017 verabschiedet. Es hat zum Ziel mehr als 500 Verwaltungsleistungen pro Bundesland bis zum Jahr 2022 Bürgern digital bereitzustellen.[3]

1.3 Zielstellung

Das Ziel dieser Arbeit besteht in der möglichst genauen Bestimmung der Skalierungsgrenzen von verteilten Applikationen. Erreicht wird dies durch die Simulation des Verhaltens dieser Applikationen unter Last.

Hierzu wird ein Discrete-Event-Simulation-Framework (DES) entwickelt, welches eine Vielzahl von Applikationen abbilden kann und somit die Untersuchung von bestehenden Applikationen in schwer konstruierbaren Szenarien ermöglicht. Das Framework wird am Beispiel des MV-Serviceportal validiert.

1.4 Struktur der Arbeit

Der Inhalt dieser Thesis beginnt mit einem Grundlagenkapitel. Wenn der Leser mit diesen Themengebieten bereits vertraut ist, kann dieses Kapiteln übersprungen werden. Hier wird ein kleiner Ausschnitt der genannten Themengebiete kurz erläutert, damit der Leser die folgenden komplexeren Konzepte einfacher verstehen kann.

Nach den Grundlagen folgt ein Kapitel zum Stand der Forschung. Hier wird auf den Inhalt und die Erkenntnisse einiger Arbeiten eingegangen, auf welchen diese Thesis aufbaut. Der Stand der Forschung hat auch Einfluss auf das Konzept.

In einem Experteninterview wurden Anforderungen für die Implementierung des Konzeptes gesammelt. Diese Anforderungen und der Stand der Forschung sind die Grundlage für das Konzept, welches für diese Thesis entwickelt wird.

Das Konzept wird in zwei Kapitel erläutert. Begonnen wird mit dem Grobkonzept, in welchem eine Auswahl der Technologien und Strategien für die Modellierung und Implementierung getroffen wird. Anschließend folgt das Feinkonzept. In diesem Kapitel wird das entwickelte Konzept und die allgemeine Herangehensweise erläutert. Dieses Kapitel ist besonders wichtig für das Verständnis der Folgenden.

In den beiden folgenden Kapiteln wird auf die Implementierung eingegangen. Im Kapitel Verifikation und Validierung wird das implementierte Konzept auf seine Anwendbarkeit und Korrektheit in einer Reihe von Tests geprüft.

Abschließend folgen das Fazit und Verbesserungsvorschläge, in diesem Kapitel wird auf die Anwendbarkeit des Konzeptes und der Implementierung eingegangen.

2 Grundlagen

In dieser Thesis werden wichtige Verfahren aus den Gebieten der Simulation und des Applikationsprofiling vereint. Dieses Kapitel soll dazu dienen ein Grundwissen über die relevanten Themenbereiche zu schaffen, um diese Thesis und den Stand der Forschung besser einordnen zu können. Es wird mit einer allgemeinen Einleitung zu Simulationen und Arten von Simulationen begonnen, eine besonders relevante Art von Simulation wird hierbei hervorgehoben, auf sie wird in einem weiteren Unterkapitel gesondert eingegangen. Anschließend folgt eine Einführung in das Applikationsprofiling, relevante Profilingkonzepte und zu beachtende Besonderheiten.

2.1 Simulationen

Computer Simulationen begannen nach dem Zweiten Weltkrieg als Werkzeuge der Meteorologie und Atomforschung. Seitdem haben sie Anwendung in weitere Disziplinen, wie der Astronomie, Biologie, Verhaltensforschung, Medizin und vielen weiteren gefunden.

Simulationen können drei Aufgaben erfüllen: sie können der Erklärung oder Demonstration dienen, der Vorhersage von Daten oder zum Verstehen von bereits gesammelten Daten. [5]

Ein Simulationsmodell ist eine Approximation eines Systems, das Verhalten dieser Approximation kann durch Algorithmen beschrieben werden. Diese Sammlung von Algorithmen ist das Modell. Wird dieses Modell auf einem Computer ausgeführt wird es Computer Simulation genannt. Die Ausführung eines Modelles umfasst das Setzen von Startparametern zum Zeitpunkt t und das Berechnen des Zustandes des Systems für den Zeitpunkt t+1. Der Zustand zum Zeitpunkt t+1 wird dann wiederum die Grundlage für die Berechnung des Zustandes zum Zeitpunkt t+2, dieser Vorgang wird bis zum Erreichen des Abbruchkriteriums wiederholt. Die Entwicklung des Zustandes des Systems wird oft auf eine Art visualisiert, welche Messungen des simulierten Systems entspricht. [5]

Computer Simulation ist ein sehr breites Feld, bedingt durch die vielen Disziplinen in der sie eingesetzt werden kann. Aufgrund dessen haben sich verschiedene Subtypten der Simulation herausgebildet, auf diese wird im folgenden Kapitel eingegangen.

2.1.1 Überblick

Es gibt verschiedene Ausprägungen von Simulationen, üblicher Weise haben sie eine Kombination der folgenden Eigenschaften. Eine Simulation kann stochastisch oder deterministisch sein, statisch oder dynamisch, kontinuierlich oder diskret, lokal oder verteilt. Durch die Kombination dieser Attribute lassen sich 16 Simulationsarten herausbilden.

Stochastisch Stochastische Simulationen nutzen Zufallszahlengeneratoren, um zufällige Ereignisse oder Abweichungen zu generieren.

Deterministisch Deterministische Simulationen können beliebig häufig wiederholt werden und erzeugen immer wieder dasselbe Ergebnis.

Statisch Statische Simulation besitzen keinen Mechanismus, um auf neue Input-Parameter zu reagieren.

Dynamisch Dynamische Simulationen modellieren das Verhalten eines Systems in Abhängigkeit von seiner Umgebung, durch veränderte Eingangsdaten.

Kontinuierlich Kontinuierliche Simulationen nutzen kontinuierliche Funktionen, um das Verhalten eines Systems zu jedem Zeitpunkt bestimmen zu können.

Diskret Diskrete Simulationen nutzen diskrete Funktionen zur Bestimmung des Zustandes eines Systeme und können den Zustand eines Systems nur zu diskreten Zeitpunkten bestimmen. Eine Sonderform hierbei ist die Diskrete Event Simulation.

Lokal Lokale Simulationen werden auf nur einem Computer ausgeführt.

Verteilt Bei besonders umfangreichen oder detaillierten Simulationen kann die Laufzeit bei einer lokalen Ausführung schnell zu lange dauern. In diesem Fall könnte die Entwicklung einer verteilten Simulation sinnvoll sein, welche die Ressourcen mehrere Server gleichzeitig nutzen kann. Ein prominentes Beispiel hierfür ist „folding at home” eine verteilte Proteinfaltungssimulation, die Rechner, die an der Ausführung dieser Simulation beteiligt sind, werden von Privatpersonen bereitgestellt. Dieses Netzwerk ist das erste ExaFLOP Computer System. [6]

Die oben beschriebenen Simulationsarten sind eine Möglichkeit Simulationen nach Charakteristika ihrer Ausführung zu klassifizieren. Eine weitere Möglichkeit ist die Klassifizierung nach dem Modellierungsansatz. Hierbei kann mindestens zwischen Mathematischen Modellen und Agentenbasierten Modellen zu unterschieden werden.

Simulationen basierend auf mathematischen Modellen oder formelbasierte Simulationen bilden üblicherweise physikalische Prozesse ab. Diese Art von Simulation kann für Wettervorhersagen, Stresssimulation in Werkstücken oder auch Flüssigkeitssimulationen genutzt werden.

Agentenbasierte Simulationen werden oft in der Epidemiologie, Ökologie oder auch KI eingesetzt. Anders als bei den formelbasierten Simulationen gibt es hier keine global gültige Formel, welche den Zustand aller Komponenten vorhersagt, stattdessen berechnet jede Einheit ihren Zustand selbst, abhängig von ihrer Umgebung.[5]

2.1.2 Discrete Event Simulation

In New York erfolgte im Jahr 1968 die erste Konferenz bei der Discrete Event Simulation die Hauptrolle einnahm. In den folgenden Jahrzehnten bestanden die größten Probleme in der begrenzten Rechenkraft der verfügbaren Computer, aber auch die großen Schwierigkeiten während der Entwicklung und Anwendung dieser Simulationen.[7] Auch wenn DES bereits ein halbes Jahrhundert alt ist, kann sie bei vielen Problemen angewandt werden.[8] Bei DES handelt es sich um eine Weiterentwicklung der Fixed Timestep Simulation und einen Vorgänger der Process Oriented Discrete Event Simulation (PODES).

Bei der Fixed Timestep Simulation wird der Zustand des Systems in gleichmäßigen Zeitabständen simuliert. Wobei das System vom Zustand eines Zeitpunktes zum Zustand eines anderen Zeitpunktes mithilfe einer Transformationsfunktion übergeht. Dieser Ansatz ist leicht verständlich und leicht implementierbar, besitzt aber auch zwei Schwächen. Wenn es Zeitspannen in dem simulierten Zeitraum gibt, in denen es zu keinen Änderungen kommt werden dennoch Berechnungen durchgeführt. Zusätzlich kann der feste minimale „Timestep“ zu abweichenden Ergebnissen führen. Ist eine höhere Präzision gefordert muss die Simulation mit einem kleineren Timestep erfolgen, was zu einem höheren Rechenaufwand und längerer Laufzeit führt. DES vermeidet diese Probleme, indem die Simulationszeit nicht in festen Abständen fortschreitet, sondern nur mit neuen Ereignissen (Events). Die Grundidee besteht darin, dass eine Änderung im Zustand des Systems nur dann erfolgen kann, wenn

sie durch ein Event ausgelöst wird. Dieses Event kann simulationsintern sein oder als Input von außerhalb des Systems kommen. Somit werden keine Berechnungen durchgeführt, wenn diese nicht zu Zustandsänderungen führen.

PODES nutzt den technischen Fortschritt der letzten Jahrzehnte und verwendet OS-Prozesse. Jedes aktive Objekt innerhalb der Simulation besitzt einen eigenen Prozess. Die Kommunikation und Synchronisation kann mit Interrupts erfolgen. Die Effizienz dieses Ansatzes ähnelt der der DES, allerdings eignet sicher dieser Ansatz besonders für Simulationen mit vielen Events, da die Prozesse für eine höhere Auslastung von Mehrkern-CPUs sorgen.

2.2 Applikations Profiler

Profiling ist eine Art von dynamischer Programmanalyse, welche z. B. die Messung der Auslastung bestimmter Ressourcen, das Zählen von Funktionsaufrufen, die Messung von Funktionslaufzeiten oder auch Laufzeitkomplexitätsanalyse eingesetzt werden kann. Profiler dienen dazu einen Einblick in die Funktionsweise und Ausführung eines Programmes zu gewinnen. Abhängig von der Art der Datenerhebung kann der Einfluss auf die Performance variieren. [9] [10]

2.2.1 Dynamische Profiler

Es kann zwischen mindestens fünf Arten von Profilern unterschieden werden: Event-Basierte Profiler, Statistical / Sampling-Basierte Profiler, Instrumentation, Interpreter Instrumentation und Hypervisor. Im Folgenden werden die drei erst genannten Arten näher erläutert, da diese am häufigsten eingesetzt werden. Interpreter Instrumentation und Hypervisor Profiling sind meist deutlich aufwendiger in der Anwendung.

Sampling-Basierte Profiler messen relevante Ressourcenauslastungen und Statistiken zu Zeitpunkten nach statistischen Zufallsverteilungen, in vielen Fällen der Gleichverteilung. Diese Art von Profiler kann unter dem Safe Point Bias leiden. Sampling-Basierte Profiler sollten Samples in zufälligen Abständen sammeln, wobei allerdings einige dieser Profiler ihre Samples zu Safe Points sammeln. Ein Safe Point ist ein Zeitpunkt, zu welchem die Ausführung eines Programmes pausiert wird, um Daten zu erheben. Nur zu diesen Zeitpunkten kann der gesamte Zustand eines Programmes konsistent erfasst werden. Das

kann zu erheblichen Diskrepanzen zwischen den Ergebnissen mehrerer Profiler Ergebnissen führen. [11]

Event-Basierte Profiler reagieren auf bestimmte, definierbare Ereignisse und messen nur zum Zeitpunkt dieser Ereignisse. Mit diesem Ansatz ist es möglich inputspezifisches Verhalten und Input spezifische Performance zu messen. Traditionelle Profiling-Techniken betrachten das Gesamtsystem und versuchen Ressourcenauslastungen bestimmten Zeilen im Quellcode zuzuordnen. Moderne Systeme, im besonderen Webapplikationen, müssen sehr reaktiv sein. Traditionelle Profiler würden aber unter Umständen relevante Funktionen als irrelevant einstufen, da sie relativ ressourcensparend sind, verglichen mit z. B. Initialisierungsfunktionen. Allerdings haben eben diese nur einen geringen Einfluss auf die Nutzererfahrung, verglichen mit einer Funktion zum Beantworten einer Nutzeranfrage. In solchen Fällen braucht es eine Art von Profiler, bei welchem relevante Ereignisse spezifizierbar sind. Somit kann die Ressourcenauslastung detaillierter und Event-Basiert betrachtet werden. [12]

Instrumentation ist ein Sammelbegriff für Code, welcher Teil eines Programmes ist und für die Beobachtung dessen Verhaltens eingesetzt wird. Aspektorientierte Programmierung kann hierzu gezählt werden. Instrumentation kann zu mehreren Zeitpunkten erfolgen: bei der Erzeugung einer ausführbaren Datei, während der Ausführung oder mit dem Start eines Programmes. Hierauf wird in Kapitel 7.1.1 genauer eingegangen. Instrumentation wird üblicherweise für ein spezifisches Programm entwickelt. Die Entwicklung bedarf daher ein tiefes Verständnis der Struktur eines Programmes. Die erzeugten Ergebnisse sind meist sehr aussagekräftige, da die gemessenen Metriken vorher bewusst ausgewählt wurden.

2.2.2 Statische Profiler

Ebenfalls erwähnenswert sind statische Code-Analyse-Tools, diese können auch als statische Profiler bezeichnet werden. Sie untersuchen nicht die Binärdatei eines Programmes oder das Verhalten zur Laufzeit. Stattdessen analysieren statische Profiler den Quellcode. Ein Vorteil von statischen Profilern liegt darin, dass sie das gesamte Programm analysieren können, da sie Zugriff auf den gesamten Quellcode und somit jede Eventualität haben.

Dynamische Profiler untersuchen das Programm zur Laufzeit. Um dieselbe Code Abdeckung mit einem dynamischen Profiler zu erzielen, müsste jeder mögliche Input

untersucht werden. Dieser Fuzzing [13] Ansatz kann sehr ressourcenintensiv sein. Selbst in diesem Fall kann es unerreichbare Code Abschnitte geben, welche nur durch statisches Profiling zu identifizieren sind. Allerdings ist das Verhalten von Programmen in vielen Fällen dynamisch und von den verarbeiteten Daten abhängig. Dieses dynamische Verhalten kann von einem statischen Profiler nur im begrenzten Maße evaluiert werden. Besonders interpretierte Sprachen wie Python oder MatLab sind mit statischen Profilern schwer evaluierbar. [14]

2.3 Aspektorientierte Programmierung

Aspektorientierte Programmierung (AOP) beschreibt ein Programmierparadigma, welches darauf abzielt Modularität zu erhöhen, indem Querschnittsthemen in Aspekte ausgeteilt werden. AOP kann auch als Form von Instrumentation genutzt werden. Der Quellcode muss hierfür nicht manuell angepasst werden. Somit kann die Business Logic vom Logging getrennt werden. Ein Beispiel für die Anwendung eines Aspektes wäre das Loggen der Laufzeit von Funktionen, oder das Zählen der Error Meldungen während der Bearbeitung einer Funktion. [15][16]

AOP ist eine eigene Programmierdiziplin, in diesem Kapitel wird nur auf Teilaspekte eingegangen, die für diese Thesis relevant sind. AOP wird in Kapitel 7.2.1 eingesetzt.

Als Aspect Weaving wird das Einflechten oder Einfügen von Aspekten in ein Programm bezeichnet. Im Folgenden werden vier Arten des Aspect Weaving erläutert: Source Code Weaving, Compile Time Weaving, Binary Weaving und Load Time Weaving.[17]

Source Code Weaving Diese Art des Weaving wird nicht von AspectJ[1] verwendet und ist heute von geringer Bedeutung. Diese Art des Weaving ähnelt Pre-Processor Anweisungen, was den Vorteil hat, dass keine Bibliotheken oder Compiler genutzt werden müssen, allerdings kann dieser Ansatz zu großen Binaries führen.

Compile Time Weaving Bei diesem Ansatz werden Aspekte zur Compile Zeit in das ausführbare Programm eingefügt. Hierzu sind ein spezieller Compiler und eine Runtime Bibliothek nötig. Der Overhead durch Compile Time Weaving ist sehr gering, auch die Größe der Binary ist nur wenig beeinflusst. Mit dieser Art des Weavings ist es nicht möglich bestimmte Aspekte nur unter bestimmten

[1] AspectJ ist ein AOP Framework für Java.

Bedingungen zu „Weaven“, da alle Aspekte zum Zeitpunkt des Compilen in die Binary eingefügt werden. Außerdem kann mit diesem Ansatz nur Quellcode angereichert werden, welcher auch vom Autor des Programmes geschrieben wurde, externe Bibliotheken und private Funktionen werden ignoriert.

Binary Weaving Dieser Ansatz besitzt die selben Vorteile wie Compile Time Weaving, da das Weaving aber nach dem Kompilieren stattfindet, können Funktionen von Dritten berücksichtigt werden. Es entsteht kein Overhead beim Starten der Applikation. Allerdings werden Aspekte immer eingewoben, ähnlich dem Compile Time Weaving.

Load Time Weaving Ein Weaving Agent wird mit dem Start der Applikation geladen und fügt Aspekte dynamisch und abhängig von der Umgebung der Applikation ein. Dieser Agent wird mit einer Konfigurationsdatei angepasst. Diese Art des Weaving erzeugt einen Overhead vergleichbar mit dem des Compile Time Weavings, da das Weaving in beiden Fällen nur ein Mal stattfindet. Mit diesem Ansatz ist es aber möglich zwischen mehreren Aspekten zu wählen, z. B. in Abhängigkeit vom Betriebssystem. Mit diesem Ansatz ist es möglich private Methoden und Methoden in Bibliotheken von Dritten zu berücksichtigen.

Proxy Based LTW Diese spezielle Form des LTW wird von Spring AOP[2] genutzt. AspectJ nutzt die 3 vorhergenannten Weaving Arten. Sie ist grundsätzlich eingeschränkter als LTW, wird aber nativ von Spring unterstützt, einem momentan beliebten Framework. Es können nur öffentliche Methoden berücksichtigt werden, statische und private Methodenaufrufe werden nicht beachtet. Dieser Ansatz ist nur der Vollständigkeit halber mitaufgelistet und besitzt für diese Arbeit keine weitere Relevanz.

2.4 Queueing Theory

Queueing Theory ist ein Teilbereich der Mathematik, welcher sich mit der Modellierung von Systemen mit Warteschlangen beschäftigt. Die erste Arbeit im Feld der Queueing Theory mit dem Titel „The Theory of Probabilities and Telephone Conversations” im Jahr 1909 veröffentlicht und von A. K. Erlang verfasst. Erlang arbeitete für eine Telekommunikationsfirma in Kopenhagen und beschäftigte sich dort mit dem Problem der Wartezeitminimierung, in einem Vermittlungszentrum. Während seiner Arbeit realisierte er, dass Wartezeitminimierung ein Problem von

[2]Aspring AOP ist eine Art der AOP, welche im Spring Framework eingesetzt wird.

größerer Bedeutung ist und begründete mit seiner Veröffentlichung das Feld der Queueing Theory. [18]

Ein einfaches Queueing System ist ein System, in das Objekte eingehen, welche von einer Reihe verarbeitender Stationen in einer bestimmten Zeit verarbeitet werden. Sind alle Stationen beschäftigt, werden sie in eine Warteschlange eingefügt. In welcher Reihenfolge die Objekte der Schlange entnommen werden, bestimmt die „discipline" - Disziplin. Diese Queue kann unter anderem als FIFO-, Priority- oder Zufalls-Queue implementiert werden. Diese Disziplinen können auch kombiniert werden, zusätzlich können Präemptive oder nicht Präemptive Verarbeitungsstrategien an der Station angewandt werden. Handelt es sich bei den Objekten im System, um z. B. Kunden, muss auch davon ausgegangen werden, dass Kunden nach einer gewissen Zeit das System verlassen können, was ebenfalls mittels Queueing Theory modelliert werden kann. [19]

Queueing Theory kann zur Validierung bestimmter Aspekte von Simulationen genutzt werden.[20]

Zur Beschreibung eines Queueing Systems mit einer Queue kann die Kendall-Lee Notation verwendet werden. Es gibt keine Notation für Queueing Systems mit mehreren Queues. [21] Diese Notation enthält die 6 charakterisierenden Eigenschaften eines Queueing Systems, getrennt mit „/". Beginnend mit der Ankunfts- und Service-Zeitverteilung folgen die Anzahl der Arbeiter, die Disziplin der Queue, die maximale Anzahl von Objekten im System und zuletzt die Anzahl von Objekten, welche jemals in das System kommen könnten. Als Beispiel soll die folgende Notation gelten: M/M/5/FCFS/20/inf entspricht einem System mit einer eingehenden Exponentialverteilung und einer exponentiellen Verarbeitungszeit, 5 Verarbeitenden Stellen, welche eine First-Come-First-Served Queue verwenden, diese Queue hat eine Kapazität von 20 Objekten, die Objekte werden aus einer unendlichen Menge von potenziellen Objekten ausgewählt. [18]

Einige relevante Metriken von Queueing Systemen sind die Folgenden [21]:

L Durchschnittliche Anzahl von Objekten im System

L$_q$ Durchschnittliche Anzahl von Objekten in der Warteschlange

W Durchschnittliche Zeit, welche ein Objekt im System verweilt

w$_q$ Durchschnittliche Zeit, welche ein Objekt in der Warteschlange verweilt

p Serverauslastung

Ein Queueing System kann auch aus miteinander verknüpften Queues bestehen. Es existieren zwei Arten von Queueing Networks: offene und geschlossene Systeme. Offene Systeme haben eine variable Anzahl von Objekten über einen Zeitraum, geschlossene Netzwerke haben keine Ein- oder Ausgänge von Objekten, somit ist die Anzahl von Objekten konstant. [21]

Queueing Theory kann zur Modellierung vieler Prozesse verwendet werden. Einfache Modelle können teils analytisch betrachtet werden, im besonderen Queueing Networks, sind besser mittels Simulation zu betrachten. Diese Modelle können als grobe Richtline zur Evaluierung der Performance eines reellen Systems genutzt werden. [21]

3 Stand der Forschung

Dieses Unterkapitel dient der Zusammenfassung relevanter Aspekte existierender Arbeiten. Im speziellen handelt es sich um das MONARC Projekt am CERN und Arbeiten zum Tracing mit AspectJ, welche großen Einfluss auf diese Arbeit hatten und häufig referenziert werden.

3.1 MONARC

Das Models of Networked Analysis at Regional Centres for LHC Experiments (MONARC) Projekt begann circa im Jahr 1998 am CERN. An dem Projekt waren über 50 Personen aus 8 Nationen beteiligt. Das Ziel des Projektes bestand darin die zu erwartende Performance von Hochleistungsrechenzentren und Verbunden von mehreren Rechenzentren zu simulieren. Die Laufzeit des Projektes wurde ursprünglich auf circa 200 Personen Monate geschätzt.[22]

MONARC war das erste und größte Projekt, das während der Recherche für diese Thesis gefunden werden konnte, welches DES für die Simulation verteilter Systeme einsetze. Seit dem Jahr 2000 gab es eine Reihe weiterer Projekte zu diesem Thema, wie z. B. [23], [24], [25], oder auch [26]. All diese Arbeiten beschäftigen sich mit der Simulation von verteilten Computersystemen, haben aber jeweils unterschiedliche Schwerpunkte und verwenden eine Form der DES. Auf MONARC wird als erstes und größtes Projekt besonders eingegangen.

Im Rahmen dieses Projektes wurde das MONARC Framework entwickelt, ein Closed Source, Java basiertes Process Oriented Simulation Framework. Der Fokus während der Entwicklung lag auf der Anwendbarkeit des Frameworks für Large Hadron Collider (LHC) Experimente. Mit diesem Framework ist es möglich die Ressourcenauslastung und Laufzeit von rechenintensiven Analyseprozessen akkurat vorherzusagen. Das Framework ermöglicht auch die Kostenvorhersage unter anderem für verschiedene Netzwerkarten (national, international, lokal) und weitere genutzte Ressourcen.

Das erklärte Ziel des Simulationsframeworks besteht darin eine realistische Simulation von verteilten Rechnernetzwerken für anpassbare Datenverarbeitungsprozesse bereitzustellen, um somit eine flexible und dynamische Umgebung zur Evaluierung verschiedener Architekturen zu bieten.

Das Framework besitzt eine Simulations-Engine, welche einem semaphorbasierten Scheduling-Mechanismus für „aktive Objekte“ bereitstellt. Das aktive Objekt ist eine Basisklasse, von der alle zeitabhängigen Komponenten abgeleitet sind. Es implementiert Funktionen für die Kommunikation mit weiteren Klassen, Unterbrechungen und Wiederaufnahme der des Prozesses. Aufgrund dessen ist es möglich Mechanismen wie Caching oder Swapping zu simulieren. Da das Framework Prozessbasiert ist und simulierte Programme sehr viele aktive Objekte besitzen können, wurde eine Komponente eingeführt, welche aktive Objekte recycelt, somit wird vermieden das Prozesslimit des Betriebssystems zu erreichen. [27]

Wenn mehrere Funktionen auf einer CPU ausgeführt werden, hat dies Auswirkungen auf die Laufzeit der Funktionen entsprechend Abbildung 3.1. Task1 hat einen Endpunkt TF1, zu T2 wird Task1 unterbrochen, da eine weitere Funktion auf der CPU begonnen wurde, wodurch ein neuer Zeitpunkt TF1 berechnet wird. Zum neuen Zeitpunkt TF1 wird Task2 unterbrochen, da dieser Funktion nun mehr Ressourcen zur Verfügung, stehen endet Task2 früher und TF2 kann vorgezogen werden. [28]

Für jeden neu erzeugten Job sucht der Scheduler nach Servern, welche diesen Job ausführen können. Findet der Scheduler mehrere Server, wird der mit der geringsten Auslastung gewählt. Sollte der RAM des Servers durch den neuen Job zu mehr als 100% ausgelastet werden, wird der Job dem nächsten Server zugewiesen. Sind alle Server ausgelastet, wird die Funktion zum frühesten Zeitpunkt zugewiesen, zu dem ein Server die nötigen Ressourcen besitzt.

Das entwickelte Simulationsframework wird als CPU und Code effizient eingeschätzt und ist in der Lage verteilte Datenverarbeitungsaufgaben gut abzubilden, selbst wenn diese sehr komplex sind. [28] Diese Einschätzung galt zum Zeitpunkt der Validierung. Eine spätere Veröffentlichung weist aber auch darauf hin, dass Weiterentwicklungen in der Speicher- und Netzwerktechnik weiterverfolgt werden müssen, um auch in Zukunft gute Ergebnisse zu produzieren. [29]

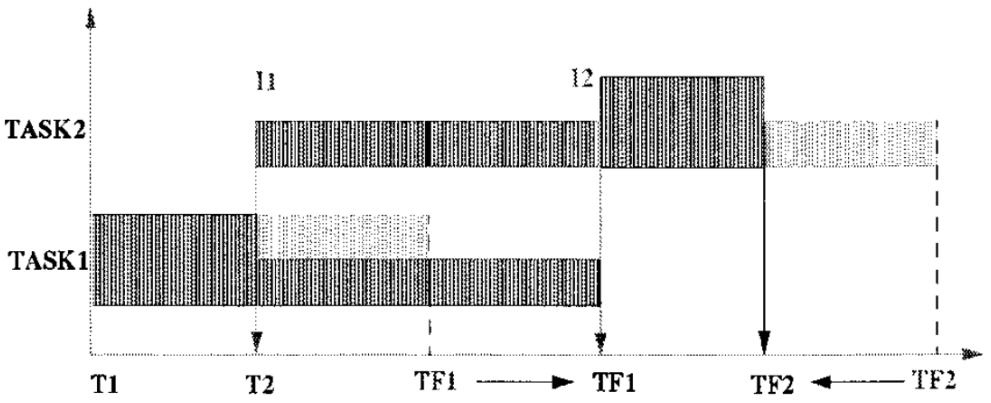

Bild 3.1: MONARC: Ressourcenallozierung [28]

3.2 Improving Dynamic Data Analysis with Aspect-Oriented Programming

Dieses Paper wurde von Thomas Gschwind und Johann Oberleitner in der Abteilung für Verteilte Systeme im Institut für Informationssysteme an der TU Wien im Jahr 2003 veröffentlicht. In diesem Paper wird ein Reverse Engineering Tool beschrieben, welches AspectJ nutzt, um Entwicklern einen tiefen Einblick in die Funktionsweise eines Programmes zum Reverse Engineering dessen zu bieten. [30] Diese Arbeit ist relevant, da das Rekonstruieren des Verhaltens einer Applikation in den Bereich des Reverse Engineering fällt.

Zu dem Zeitpunkt existierten zwar Applikationen zum vollständigen Reverse Engineering von Applikationen, diese hatten aber oft einen zu großen Overhead, für die bloße Ergänzung eines Feature, um praktikabel zu sein. Die Autoren identifizieren einige Probleme mit existierenden Ansätzen wie z. B. begrenzter Einblick in die Funktionsweise, da Funktionsparameter nicht beachtet werden oder auch dass ein Programm erneut kompiliert werden muss, um nötige Instrumentation einzufügen. Es wird die Verwendung von Aspekt-Orientierter Programmierung mit AspectJ als eine Lösung vorgeschlagen und in einem Reverse Engineering Tool Namens ARE implementiert.

Eine hervorhebenswerte Eigenschaft von AspectJ ist, dass es einen sehr tiefen Einblick, bis auf die unterste Funktionsebene, erlaubt. Mit AspectJ kann der Beginn und das Ende jeder Funktion erfasst werden. Aufgrund dieser Daten können nicht

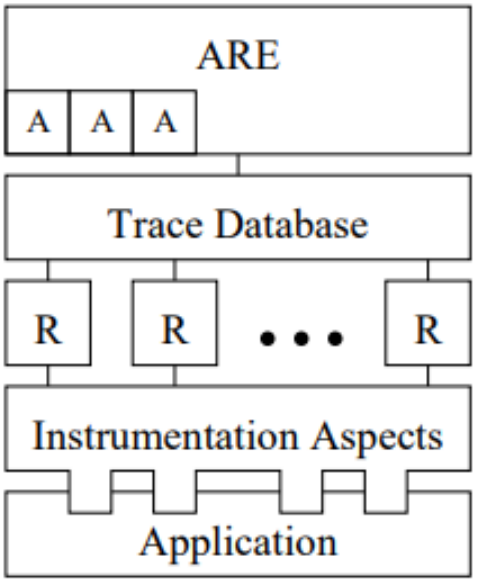

Bild 3.2: ARE Architektur [30]

nur Programm Traces[1], sondern auch Objektinstanzen durch die Ausführung des Programmes nachverfolgt werden.

Das Paper bietet eine kurze Einleitung zu AspectJ, der Funktionsweise im Allgemeinen und der Implementierung relevanter Aspekte. In Abbildung 3.2 ist die Architektur des erstellten Tools visualisiert. Die Instrumentation in Form des Aspektes wird automatisch von dem AspectJ Compiler in die Applikation eingefügt und misst während der Ausführung, die alle Traces, welche in die Trace Datenbank übernommen werden. Das Tool ARE filtert die Traces und stellt nur die relevanten Traces dar.

Mit dem Tool, welches für dieses Paper implementiert wurde, war es möglich eine BeanBuilder Applikation von Sun Microsystems nur aufgrund der mit dem Tool gesammelten Daten zu erweitern.

Die Autoren kommen zu dem Fazit, dass AspectJ und AOP geeignet sind, um das Verhalten einer Applikation zur Laufzeit zu beobachten und ausreichend relevante Erkenntnisse aus diesen zu ziehen.

[1]Ein Trace (kurz für Stack-Trace) besteht aus aktiven Stack-Frames. Stack-Frames gehören zu Funktionsaufrufen eines Programmes. Über einen Stack-Trace kann nachvollzogen werden auf welchem Weg eine Funktion aufgerufen wurde.[31]

3.3 Tracing mit AspectJ in verteilten Systemen

Dieses Paper von Lionel C., Briand Yvan Labiche und Johanne Leduc von der Carleton University in Kooperation mit Siemens Corporate Research Inc., baut auf der obenstehenden Arbeit auf und überträgt die dort gesammelten Erkenntnisse auf verteilte Systeme. [32]

In dem Paper von Thomas Gschwind und Johann Oberleitner wird nicht zwischen Threads unterschieden die Ergebnisse der Untersuchung beziehen sich nur auf rein sequenzielle und monolithische Programme.

Es werden vier Schwierigkeiten hervorgehoben: die Instrumentation des Programmes, die Speicherung in einem geeigneten Format und die Rekonstruktion von Traces aus diesen Logs und dem Zusammenführen dieser Traces zu einem annähernd kompletten Abbild der Applikation. Im Besonderen wird auf die ersten zwei Punkte eingegangen.

Die Instrumentation erfolgt pro Komponente des verteilten Systems die gesammelten Daten werden auf einem zentralen Logging-Server zusammengeführt. Threads können seit Java 1.5 über ihre Thread ID identifiziert werden, allerdings können diese IDs wiederverwendet werden, daher müssen zusätzlich ObjectID und Klassenname an den Server übermittelt werden. So liegen genügend Informationen vor, um Traces zu rekonstruieren. Da nur die Abfolge der Events relevant ist, werden Traces über einen Integer identifiziert, welche mit jedem Event inkrementiert wird. Wäre die zeitliche Reihenfolge relevant, müsste die Systemzeit zu dem Zeitpunkt ermittelt werden und stattdessen übertragen werden.

Die Kommunikation der Komponenten des verteilten Systems in diesem Paper erfolgte per Java Remote Method Invocation (RMI), daher wurde ein RMI Interceptor implementiert[2]. Erfolgt die Kommunikation der Komponenten auf andere Art müssen weitere Interceptoren implementiert werden.

Wo der Ansatz der vorangegangenen Arbeit flexibel und auf mehrere Programmiersprachen anwendbar war, ist der Ansatz dieser Arbeit nur auf verteilte Applikationen, welche vollständig in Java implementiert wurden anwendbar. Es zeigt aber grundsätzlich das Potential und die Möglichkeit auch verteilte Applikationen mit einem Aspektorientierten Ansatz zu analysieren.

[2]Ein Interceptor folgt dem Interceptor Pattern. Dieses Pattern beschreibt eine Komponente die Funktionen, IO- oder Netzwerkoperationen abfängt und weiterleitet, sich aber als 3te Partei in diese Operation einfügt.

4 Anforderungen

In diesem Kapitel wird auf die Anforderungen der Entwickler des DVZ eingegangen, für welche die Implementierung dieser Thesis von Bedeutung ist. Hierzu wurde ein Experteninterview mit einem Entwickler durchgeführt, welcher die Applikation von Anfang an mitentwickelte. Eine zu Gunsten der Leserlichkeit gekürzte Version kann im Anhang (A.1) gefunden werden.

Die folgenden Anforderungen wurden durch das Experteninterview gefunden.

Anforderungen an die Simulation:

- Ziel der Simulation ist die Bestimmung der maximalen Nutzeranzahl.
- Eine Abweichung von weniger als 30% ist ideal.
- Es sollen begrenzende Komponenten identifiziert werden.
- Die Simulation der bis zu 25.000 erwarteten täglichen Nutzer soll mindestens in Echtzeit möglich sein.

Anforderungen an die Visualisierung:

- Zu visualisierende Metriken sind die folgenden: Anzahl gleichzeitiger Nutzer, Auslastung der Server, Antwortzeit der Interaktionen.
- Möglichst wenige Plots mit Filteroperationen werden bevorzugt.

Anforderungen an die Benutzbarkeit:

- Die Vor- und Nachbereitungszeit der Messungen soll pro Durchlauf nicht mehr als 60 min. in Anspruch nehmen.

5 Grobkonzept

In diesem Kapitel werden Anforderungen mit dem Stand der Forschung kombiniert, um ein Grobkonzept zu entwickeln. Hierzu werden Strategien und Technologien evaluiert, unter der Berücksichtigung dieser Anforderungen. Wie aus den Grundlagen und dem Stand der Forschung hervorgeht, braucht es für die Simulation eines Systems drei Dinge: ein Verständnis der Mechanismen des Systems, eine Beobachtung der Struktur und der aktiven Komponenten und ein Simulationsmodell, welches die Mechanismen des Systems so abbildet, dass beobachtetes Verhalten nachgebildet werden kann.

Hierzu braucht es mindestens drei Komponenten:

- Profiler
- Simulation
- Visualisierungskomponente

Die gesammelten Daten müssen in einem Format gespeichert werden, welches von der Simulation genutzt werden kann. Somit ist auch ein validierbares Datenformat nötig.

Der Profiler sammelt Daten über die Funktionen der Applikation und speichert diese in einem Format, welches sowohl menschenlesbar als auch von der Simulation verwertbar ist. Die Simulation verwendet dieses Applikationsprofil und weiteren Nutzerinput (z. B. Anzahl der auszuführenden Funktionen) und erzeugt somit Daten, welche von der Visualisierungskomponente dargestellt werden, siehe Abbildung 5.1.

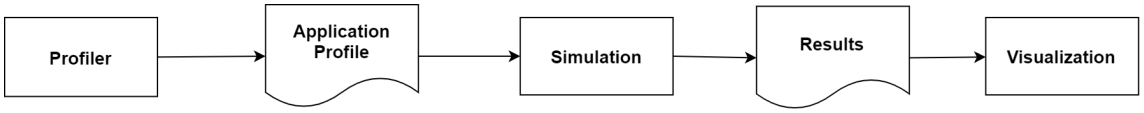

Bild 5.1: Flowchart Komponenten und Datenflüsse (vereinfacht)

5.1 Profiler

Der Profiler sammelt Daten über die Ausführung der Funktionen einer Applikation. Im Besonderen die Abfolge von Funktionen und die dadurch erzeugte Auslastung.

Es existieren eine Reihe von kommerziellen Profilern, sowohl statische als auch dynamische. Leider sind diese meist nicht quelloffen oder sehr spezialisiert, was den Export der Daten in einem verwertbaren Format und die Nutzung dieser erschwert. Zusätzlich ist ein sehr tiefer Einblick in die Funktionsweise und die funktionalen Zusammenhänge zur Laufzeit nötig. Es wird daher ein instrumentationsbasierter Ansatz für diese Thesis verfolgt.

Wie in den Arbeiten zum Tracing mit AspectJ bereits gezeigt wurde, ist ein instrumentationsbasierter Profilingansatz geeignet, um das Verhalten einer Applikation zu rekonstruieren. Wie in [30] bereits erwähnt, sind andere Reverse Engineering Tools oft übermäßig schwergewichtig und unhandlich. Aus diesen Gründen scheint es sinnvoll einen instrumentationsbasierten Ansatz mit AspectJ zu verfolgen und gesammelte Daten im Anschluss mit einem weiteren Tool in ein verwertbares Format zu konvertieren. Das Paper [30] könnte hierbei zum Vorbild genommen werden.

Das Profilingkonzept ist erweiterbar, um auch verteilte Applikationen zu „profilen“. In dieser Thesis wird aber nur das lokale Profiling einer monolithischen Applikation betrachtet.

5.2 Simulation

Das Framework, welches im Rahmen des MONARC Projektes entwickelt wurde, scheint eine der ersten genutzten Implementierungen der Prozessorientierten DES zu sein. In diesem und nachfolgenden Projekten wurden die Vorteile dieses Ansatzes, hinsichtlich der Laufzeitgeschwindigkeit und Flexibilität in der Modellierung komplexer Prozesse, aufgezeigt. Allerdings kann der Umgang mit prozessorientierten Simulationen umständlich und schwer verständlich sein, da die Laufzeitgeschwindigkeit nur von begrenzter Bedeutung ist, kann der simplere Ansatz der DES genutzt werden. DES haben sich bereits in der Vergangenheit als geeignet zur Simulation von Rechner-Netzwerken gezeigt.

5.3 Programmiersprachen und Simulationsframeworks

Für die praktische Implementierung der Konzepte dieser Thesis kommen eine Reihe von Programmiersprachen in Frage. C++ besitzt die mutmaßliche beste Performance, Java ist eine weitverbreitete Programmiersprache mit guter Threading Unterstützung, Python und JavaScript besitzen hohe Entwicklungsgeschwindigkeiten. Die Auswahl der Programmiersprache erfolgt nach den subjektiv bewerteten Kriterien Entwicklungsgeschwindigkeit, Lesbarkeit, Integrierbarkeit und in geringem Maße Performance. Unter Integrierbarkeit ist zu verstehen, inwieweit z. B. eine Visualisierungskomponente integriert werden kann.

In diesen Programmiersprachen wurden Simulations-Frameworks entwickelt, welche im Jahr 2009 in dem Paper "A Performance Comparison of Recent Network Simulators" [33] auf ihre Laufzeitgeschwindigkeit untersucht wurden. Die Untersuchung erfolgte anhand einer Netzwerksimulation. Es wurden die Frameworks ns-2 (C++), OMNet++ (C++), ns-3 (C++), SimPy (Python 2) und JiST/SWANS (Java) untersucht. Es ist anzumerken, dass Python 2 für die Untersuchung verwendet wurde, mittlerweile aber Python 3 der Standard ist, hier kann es zu Abweichungen kommen. Das Paper stellt fest, dass PySim eine 14 mal längere Laufzeit besitzt als JiST, das schnellste Framework. JiST, ns-3 und OMNeT++ werden als Favoriten hervorgehoben.

C++ hat den mutmaßlichen Vorteil der höheren Performance, verglichen mit Java und Python. Wie das eben genannte Paper zeigte, ist dies tatsächlich nicht immer der Fall. Java ist aufgrund des Garbage Collectors und der damit entfallenden manuellen Speicherverwaltung, sowohl leserlicher, also auch schneller in der Entwicklung. Somit scheint C++, in diesem Anwendungsfall, keinen Vorteil gegenüber Java zu besitzen.

Python besitzt aufgrund des einfachen Dependency Managements, vielen Bibliotheken und großen Entwicklerpools eine sehr hohe Entwicklungsgeschwindigkeit. Die Lesbarkeit ist aufgrund der einfachen Syntax und der Vielzahl von Bibliotheken teilweise vergleichbar mit Pseudocode. Dies ist von Vorteil, da somit in dieser Arbeit Listings leichter verständlich sind. Python kann das Dash Framework nutzen, welches eine sehr schnelle Erstellung von Visualisierungs-Dashboards ermöglicht und daher wünschenswert wäre. Da Dash in Python implementiert ist, ist dies problemlos möglich.

Wie im Kapitel Anforderungen bereits genannt, ist die Performance der Simulation nicht von besonderer Bedeutung. Erste Tests ergaben, dass mit einer Implementierung in Python ca. 10.000 Funktionen pro Sekunde simuliert werden können. Somit ist Python schnell genug, um den Anforderungen an die Performance zu genügen. Aufgrund dieser Randbedingung, der guten Lesbarkeit, guten Integrierbarkeit der Visualisierungskomponente und der hohen Entwicklungsgeschwindigkeit wird Python zur Implementierung genutzt. Wäre Performance von größerer Bedeutung, sollte Java bevorzugt werden.

Es gibt mehrere in Python implementierte Simulationsframeworks. Im Folgenden werden PySim, PyNSim und Mesa die Möglichkeit betrachtet ein neues Framework zu implementieren.

Mesa ist ein agentenbasiertes Framework mit dem Ziel eine Python, basierte Alternative zu NetLogo, Repast oder MASON zu sein. Auch wenn sich der vorliegende Sachverhalt agentenbasiert, modellieren lassen würde, wurde im vorangegangenen Kapitel bereits für einen Processing Network Ansatz entschieden, somit fällt Mesa aus der Betrachtung. [34]

PyNSim ist ein Framework zum Bauen von Ressourcen-Netzwerk Simulationen. PyNSim unterstützt unter anderem agentenbasierte Modelle, die Agenten kommunizieren in diesem Fall nicht direkt, sondern über das Netzwerk. Die Agenten können unterschiedliche Submodelle (Engines) implementieren. PyNSim ist ein leichtgewichtiges und vielseitiges Framework, dessen Einsatz möglich ist. [35]

SimPy ist ein prozessbasiertes DES Framework, es basiert auf Python-Generators und kann für die Simulation von asynchronen Netzwerken und Multi-Agent-Simulationen genutzt werden. Von den drei untersuchten Frameworks scheint es das vielseitigste und ausgereifteste zu sein, weshalb es auch in der oben genannten Studie untersucht wurde. Es wurde im Jahr 2002 veröffentlicht und war zu dem Zeitpunkt in Python 2 implementiert, seit SimPy3 basiert es auf Python 3. SimPy ist auch das einzige Framework, dessen Performance mit anderen populären Frameworks verglichen wurde. Diese hohe Performance basiert auf dem prozessorientierten Ansatz, bei welchem jedes aktive Objekt durch einen eigenen Thread repräsentiert wird. Aufgrund der Vielseitigkeit, langen Zeit am Markt und Performance wird SimPy gegenüber PyNSim vorgezogen und weiter untersucht. [36] [37]

Es besteht auch die Möglichkeit ein neues Framework zu implementieren. Wie bereits im Kapitel Grundlagen erläutert, sind prozessorientierte DES eine Weiterentwicklung der DES, welche komplexer im Umgang ist, da jedes aktive Objekt als ein

eigener Thread repräsentiert wird. Die Kommunikation mit diesen Objekten erfolgt durch Interrupts. Die Modellierung eines Systems und die Weiterentwicklung einer Simulation können hierdurch erschwert werden, allerdings sorgt dies auch für eine gute Auslastung der CPU, was die Performance erhöht.

Eine Schwierigkeit in der Modellierung und Weiterentwicklung mit einem PODES Framework wie PySim besteht in der M;odellierung von Services. In Kapitel 6 wird hierauf genauer eingegangen. Um das dynamische Skalieren eines Services abzubilden wird zwischen Servern und Services unterschieden, wobei jeder Service einen oder mehrere Server enthält. Diese Abstraktion erschwert die Modellierung, da aktive Objekte (Objekte mit eigenem Thread) nicht intuitiv identifiziert werden können. Zusätzlich beeinflusst die Auslastung eines Service die Verarbeitungsgeschwindigkeit der Funktion und die Funktion beeinflusst wiederum die Auslastung des Service.

Das DVZ hat ein Interesse an der Weiterentwicklung dieser Thesis. Bei einer DES entfällt die Schwierigkeit des Mappings von aktiven Objekten, somit ist die Weiterentwicklung einfacher. Dies geht allerding zu Kosten der Performance, wobei zumindest ein Teil dieser Performance durch gezieltes Multithreading zurückgewonnen werden kann. Das Monitoring der Simulation ist ebenfalls simpler.

Diese Thesis entsteht in Kooperation mit dem DVZ-MV, welche die entstehende Applikation weiterentwickeln und -nutzen möchte, zusätzlich muss diese Thesis in begrenzter Zeit fertiggestellt werden. Daher wird ein neues Simulationsframework in Python entwickelt und implementiert.

6 Feinkonzept

Das MVSP wird mit ansteigender Zahl der angebotenen Leistungen populärer. Es ist ein Zeitpunkt absehbar, zu welchem eine vertikale Skalierung der Server nicht mehr ausreicht, um die wachsende Anzahl von Nutzern zu bewältigen. Es müssen horizontale Skalierungsstrategien zur Bewältigung dieser Nutzermengen gefunden werden.

Eine Möglichkeit der horizontalen Skalierung besteht in der Replikation des bestehenden Deployment und der Ergänzung eines Load-Balancers. Dies wäre ressourcenintensiv, unflexibel und könnte zu Dateninkonsistenz führen, da jede Replik ihre eigene Datenbank und ihr eigenes Servicekonto (Kapitel 1.2) besitzt. Um Dateninkonsistenz zu vermeiden, müssten diese Komponenten zwischen Repliken geteilt werden, was wiederum zu einem festen Performance Limit führt. Eine Alternative könnte die Zerlegung der elektronischen Antragsstellungskomponente in Microservices sein. Somit könnte jeder Dienst unabhängig voneinander skaliert und deployt werden. Auch in diesem Fall gäbe es Komponenten, die zwar im DVZ gehostet, aber nicht entwickelt wurden und somit schwerer oder gar nicht horizontal skalierbar sind. Angenommen das MVSP wurde Cloud-Nativ in automatisch und dynamisch skalierende Microservices geteilt, dann gäbe es weiterhin Komponenten wie das Servicekonto, die nicht horizontal skalierbar sind und somit eine harte Grenze für die maximale Nutzeranzahl vorgeben.

Ein Ziel dieser Arbeit ist herauszufinden, wo diese Grenze liegt. Eine Möglichkeit diese Grenze abzuschätzen, wären Lasttests gegen ein Deployment, welches dem der Produktion entspricht. Aufgrund der potenziell hohen Kosten und des großen Aufwands ist dies ist nicht praktikabel.

Zusätzlich gibt es weitere Herausforderungen: nicht jeder Dienst wird von jedem Nutzer gleichmäßig beansprucht, es gibt zentrale Funktionen, die von einem Großteil der Nutzer genutzt werden und andere, die kaum genutzt werden. Es reicht nicht aus die Server Logs auszuwerten und Durchschnittswerte für die Anzahl der Anfragen pro Nutzer zu berechnen und diese auf die begrenzenden Dienste zu übertragen. Diese Werte wären nur für den „Durchschnittstag“ korrekt, somit sind sie an jedem

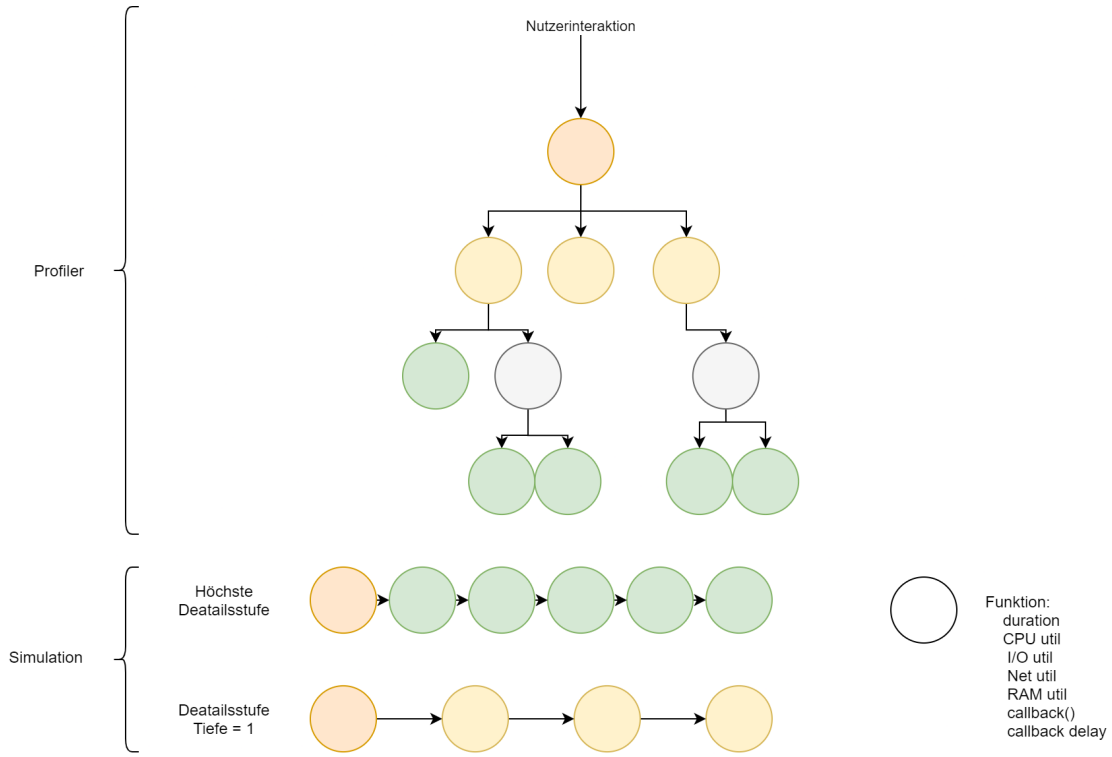

Bild 6.1: Profiling und Simulation des Call-Tree

anderen Tag falsch. Eine bessere Frage wäre die nach der maximalen Nutzeranzahl für ein bestimmtes Szenario (6.3.3) oder einen bestimmten Use-Case. Die gesuchte Antwort ist nicht eine Zahl, sondern eine Menge von Zahlen.

Eine Möglichkeit zu dieser Antwort zu kommen, ist eine Simulation. Diese Simulation approximiert das Verhalten der realen Applikation, mithilfe von vorher gesammelten Messdaten (6.3.2), mit welchen die Lasterverteilung im System simuliert werden kann.

6.1 Profiler

Um das Verhalten eines Systems zu simulieren, muss zuerst das System selbst modelliert werden. In diesem Fall soll das durch das Aufbauen eines Call-Tree aller aufgerufenen Funktionen mit ihren assoziierten Ressourcenauslastungen, mit einem Profiler geschehen.

Jeder Funktionsaufruf führt zu einem baumartigen Call-Tree, wobei die Nutzerinteraktion (6.3.4) die Wurzel darstellt, diese Interaktion kann der Aufruf einer bestimmten Route oder das Drücken eines Buttons sein. Jeder Node in diesem Baum

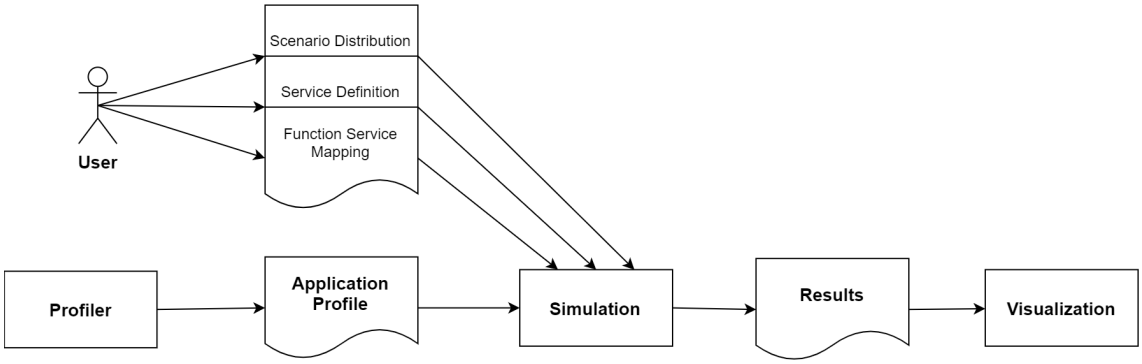

Bild 6.2: Komponenten und Datenflüsse

entspricht einer Funktion, wobei viele Funktionen nicht nur aus elementaren Operationen, sondern auch aus weiteren Funktionsaufrufen bestehen. Jede dieser Funktionen hat eine assoziierte Ressourcenauslastung (Laufzeit, CPU- Auslastung, RAM, I/O, Network).[9]

Dieser Baum kann in eine Liste von Funktionen konvertiert werden. Beim Umformen des Baumes in eine Liste kann die Traversierungstiefe variiert werden je tiefer der Baum traversiert wird, desto genauer ist die Simulation aber auch Ressourcenintensiver und zeitaufwendiger wird sie aber auch. Es ist anzumerken, dass es sich bei diesem Call-Tree nicht um einen Call-Stack handelt. Da im Call-Tree mehrere parallele Funktionsaufrufe enthalten sein können, spiegelt der Call-Tree die Struktur des Codes nur wider, wenn ein rein sequenzieller Funktionsaufruf betrachtet wird. Von einem DVZ-Entwickler implementierte parallele Funktionen werden nicht betrachtet. Hierzu wäre ein Vorgehen, wie es in Kapitel 3.3 beschrieben wird, nötig.

6.2 Modell

Das abstrahierte Simulationsmodell muss mindestens Server- und Funktionsobjekte (6.3.5) enthalten, wobei die Serverobjekte die Bearbeitung der Funktionsobjekte und auch die Kommunikation zwischen Servern abbilden.

Die gesammelten Call-Trees werden in ein Applikationsprofil konvertiert und stellen zusammen mit dem Funktion-Server-Mapping die Struktur des verteilten Systems da. Um das Verhalten der Struktur zu beobachten, muss Last erzeugt werden. Hierzu werden Listen von Zeitpunkten, entsprechend mathematischen Verteilungen erzeugt. Zu jedem dieser Zeitpunkte wird ein Szenario (6.3.7) erzeugt. So können sich überlagernde Verteilungen verschiedener Szenarien erzeugt werden. Es können

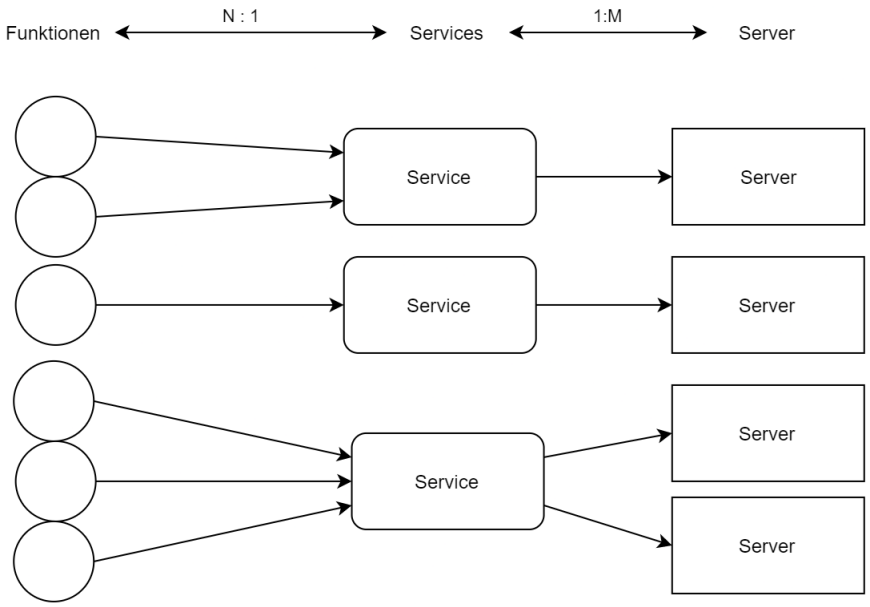

Bild 6.3: Kardinalitäten der Objekte

Szenarien entsprechend verschiedenen Verteilungen z. B. Normal- oder Student-t-Verteilung[38] erzeugt werden. Ebenso könnten historische Muster reproduziert werden. Das bisher beschriebene Verfahren ist als DES modellierbar. DES wurde bereits in der Vergangenheit erfolgreich verwendet, um verteilte Systeme zu simulieren.

Jede Funktion kann einen oder mehrere Callbacks haben, diese wiederum sind selbst Funktionsobjekte. Von welchem Dienst diese Funktionen ausgeführt werden, wo sie also Last erzeugen, ist abhängig vom Mapping (6.3.6). Funktionen können Services klassenbasiert zugeordnet werden oder über ihren vollständigen Namen. Jede Funktion gehört zu genau einem Service, wobei jeder Service N Funktionen besitzen kann. Dieser Service kann skaliert werden, sodass ein Service auf mehreren Servern ausgeführt wird, siehe 6.3.

Durch die Variierung des Mappings von Funktionen auf andere Dienste können unterschiedliche Dekompositionen eines Monolithen untersucht werden. Durch die (automatische) Skalierung von Services könnte das Skalierungsverhalten und die Elastizität untersucht werden.

Funktionen können als Interaktionen (6.3.4) zusammengefasst werden. Mehrere Interaktionen können aneinandergefügt werden, um ein Szenario zu bilden. Ein Szenario ist äquivalent zu einem Use-Case oder einer Use-Case-Variation. Ein Szenario wäre das Einloggen des Nutzers, Lesen des Posteingangs, Absenden eines Antrages und anschließendem Ausloggen. Dasselbe Szenario, wobei das Lesen des Posteingangs und das Absenden eines Antrages vertauscht sind, wäre ein neues Szenario,

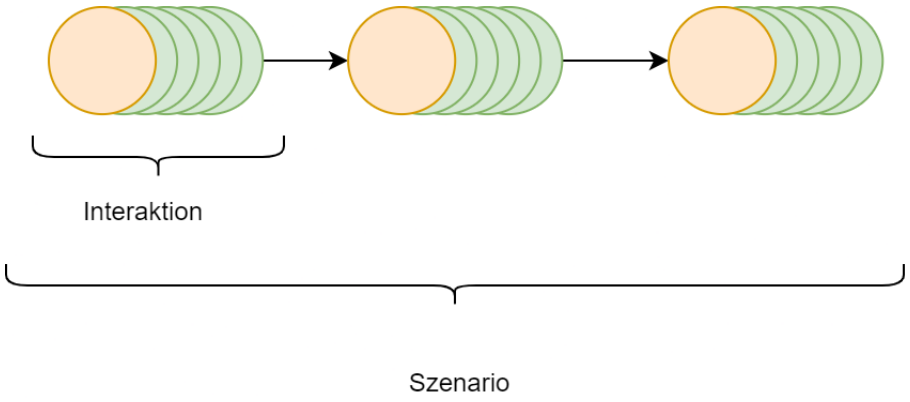

Bild 6.4: Komposition eines Szenarios

da sich die Reihenfolge der Interaktionen geändert hat.

Eine diskrete Verteilung von Szenarien dient als einer der Inputs für die Simulation. Im Idealfall entspricht die Kurve der fertiggestellten Interaktionen der Verteilungsfunktion der Input-Verteilung. Kommt es zur Überlastung eines Services ist dem nicht mehr so. Die hierbei entstehende Kurve der fertiggestellten Interaktionen kann über ein beliebiges t differenziert werden, das Maximum der hierdurch entstehenden Funktion entspricht der maximalen Nutzeranzahl über einen Zeitraum mit der Länge t. Somit kann die Anzahl gleichzeitiger Nutzer unter den definierten Umständen und in dem definierten Szenario bestimmt werden.[21]

6.3 Begriffsdefinition

In diesem Konzept wird eine Reihe von Begriffen verwendet, welche Beschreibungen in den folgenden Kapiteln erleichtern soll. Auf diese Konzepte wird während der Implementierung in den Kapiteln 7 und 8 genauer eingegangen.

6.3.1 Profiler

Der Profiler in diesem Kontext ist die Komponente, die ein Applikationsprofil, abgekürzt Profil, erzeugt. Der Profiler kann aus mehr als nur einer Komponente bestehen. Der Profiler vereint Messdaten der Ressourcenauslastung, Netzwerkverkehr und Start und Endzeitpunkte von aufgerufenen Funktionen einer Applikation. Diese Daten werden in einem Applikationsprofil in menschenlesbarer und von der Simulation nutzbaren Form gespeichert.

6.3.2 Profil

Das Applikationsprofil wird vom Profiler erzeugt und enthält Szenarios, Szenarios wiederum enthalten Interaktionen und Interaktionen enthalten Funktionen. Ein Applikationsprofil wird im JSON Format gespeichert und kann mit dem JSON-Schema in Listing A.2 validiert werden. Ein Applikationsprofil ist menschenlesbar. Profile können auch manuell erstellt werden.

6.3.3 Szenario

Ein Szenario entspricht einem Use-Case oder einer Use-Case Variation. Ein Szenario wäre z. B. das folgende: Ein Nutzer ruft das MVSP auf, der Nutzer trägt seine Login Daten ein und loggt sich ein, der Nutzer sieht seine Nachrichten ein, der Nutzer geht auf sein Applikationsprofil und loggt sich anschließend aus.

Die Einzelaktionen in dieser Aufzählung sind Interaktionen. Die Reihenfolge dieser Interaktionen ist relevant. Würde der Nutzer aus diesem Beispiel zuerst sein Applikationsprofil aufrufen und anschließend seine Nachrichten einsehen, wäre dies ein neues Szenario.

Somit können verschiedene Szenarien aus nur einer Messung generiert werden, indem die Reihenfolge der Interaktionen variiert wird. Allgemein formuliert ist ein Szenario also eine Abfolge von Interaktionen eines Nutzers.

6.3.4 Interaktion

Eine Interaktion ist eine elementare Operation, die der Nutzer gegen das Backend eines Services ausführen kann. Das Scrollen auf einer Seite gehört nicht hierzu, da es hier keine Kommunikation mit dem Backend gibt das Aufrufen einer Seite hingegen wäre eine Interaktion.

Interaktionen bestehen aus einer Abfolge von Funktionen, diese können sowohl sequenziell als auch parallel sein.

6.3.5 Funktion

Eine Funktion ist ein lasterzeugendes Objekt mit einer Laufzeit. Jede Funktion hat eine assoziierte Ressourcenauslastung für die CPU in %, RAM in MB, IO in Input-Output-Operationen und Netzwerkverbindungen in MB. Diese Last wird auf dem

ausführenden Server für die Laufzeit der Funktion erzeugt. Die Laufzeit einer Funktion entspricht einem Messwert, kann aber durch die Auslastung des simulierten Servers auch mehr Zeit in Anspruch nehmen. Ist ein Server überlastet, erhöht sich die Laufzeit, indem die Verzögerung einberechnet wird.

6.3.6 Funktion-Server-Mapping

Der Profiler „profilt" nur eine Applikation. Mit dem Funktion-Server-Mapping (kurz Mapping) können Funktionen abhängig von ihrem Klassennamen oder dem kompletten Funktionsnamen Services zugeordnet werden. Theoretische wäre es somit möglich die Dekomposition eines Softwaremonolithen in Microservices zu untersuchen.

6.3.7 Szenarioverteilung

Eine Szenarioverteilung entspricht einer mathematischen Verteilung über einen Zeitraum. Eine Verteilung wird mit einem Definitionsbereich erzeugt, dieser Definitionsbereich kann als Zeitraum betrachtet werden. Über diesen Zeitraum würden Zufallszahlen entsprechend der gewählten Verteilung erzeugt werden. Diese Zufallszahlen können als Startpunkte für Szenarien verwendet werden. Auf diese Art werden Szenarioverteilungen in der Simulation erzeugt.

6.4 Design Entscheidungen

Das oben beschriebene Konzept basiert in Teilen auf vorangegangenen Projekten, allerdings gibt es auch teils erhebliche Unterschiede zu diesen Projekten. Dieses Kapitel dient der Hervorhebungen dieser.

6.4.1 Skalierung

Das MONARC Framework ging von Serverfarmen fester Größe aus, auf welche die Last gleichmäßig verteilt werden konnte. Im Gegensatz dazu wird in dieser Thesis von einer Menge elastischer Microservices ausgegangen, welche den Großteil ihrer Aufgaben mit lokalen Daten bearbeiten können. Daher muss das Framework mit einer variablen Menge von Servern umgehen können. Allerdings ist der Netzwerkaspekt weniger relevant.

Da die untersuchte Applikation hauptsächlich durch die CPU begrenzt ist, liegt der Fokus auf der Simulation der CPU-Auslastung.

6.4.2 Messung der CPU Auslastung

MONARC nutzte die Anzahl der Zyklen, die ein Programm zur Ausführung brauchte, um die Laufzeit und Auslastung zu bestimmen. Dies war möglich, da es sich um Rechnerfarmen mit bekannten und identischen CPUs und binär kompilierte Programme handelte. Dieser Ansatz ist in dieser Arbeit nicht verfolgbar, aufgrund der folgenden Gründe. Die Performance eines Programmes kann als die Laufzeit definiert und wie folgt berechnet werden. [39]

$$CPUTime = \frac{IC * CPI}{F} \tag{6.1}$$

CPU-Time ist die benötigte Zeit auf der CPU ohne Wartezeiten, IC ist die Anzahl der Instruktionen und CPI die durchschnittliche Zahl der Instruktionen pro Zyklus und F ist die Frequenz, welche sich aus den Zyklen pro Sekunde berechnet.

Moderne CPUs haben variable Taktfrequenzen, abhängig von der anliegenden Last, der aktuellen Temperatur der CPU und dem maximalen zulässigen Energieverbrauch. Bei Programmiersprachen wie Java oder Python werden Just in Time Compiler verwendet. Die tatsächliche Anzahl von Instruktionen, die dieses Programm benötigt, ist somit von der spezifischen Laufzeitumgebung abhängig. Diese JIT-Compiler können Caching-Mechanismen implementieren, welche die gemessene Laufzeit einer Funktion weiter verfälschen.

Stattdessen wird die CPU-Auslastung in Prozent und die Laufzeit der Funktion verwendet. Die CPU-Auslastung berechnet sich aus dem Anteil der Zeit, welche mit einem Prozess verbracht wurde (CPUTime) und der tatsächlich vergangenen Zeit TT.

$$CPUUtil = \frac{CPUTime}{TT} \tag{6.2}$$

Die CPU-Auslastung eines Prozesses ist einfach auszulesen, aber auch nur begrenzt gültig. Das ausgelesene Ergebnis ist nur auf der aktuellen CPU, mit der aktuellen

Temperatur und mit der aktuellen Laufzeitumgebung gültig. Diese Unsicherheit wird in Kauf genommen.

Durch diese Trennung der Performance in Laufzeit und Auslastung ist es auch möglich die Simulation als Queueing Network zu betrachten, auch wenn die akkurate Messung oder Zuweisung der CPU-Auslastung einer bestimmten Funktion nicht möglich ist.

Es wird davon ausgegangen, dass 10 Nutzer, die gleichzeitig dieselben Schritte durchlaufen, auch genau 10-mal so viel Last verursachen, wie ein Nutzer es tun würde. Caching Mechanismen und Optimierungen werden nicht berücksichtigt, da sie sehr umständlich und zeitintensiv in der Implementierung wären.

7 Profiler

In diesem Kapitel werden die Datensammlung und die Transformation der gesammelten Daten in ein nutzbares Format erläutert.

7.1 Datensammlung

Es müssen drei Arten von Daten gesammelt werden: die Laufzeiten der Funktionen, die Ressourcenauslastung über den Zeitraum der Messung und ausgehende Netzwerkverbindungen. Diese Daten werden mit zwei unabhängigen Tools gesammelt, einem Aspekt und einem Python Tool, siehe Abbildung 7.1. Ein Äquivalent zum implementierten Java Aspekt kann in jeder Programmiersprache implementiert werden, die AOP unterstützt. Das Python Tool ist durch die verwendete Programmiersprache plattformunabhängig, die Implementierung könnte aber auch durch ein anderes Tool erfolgen, insofern z. B. eine höhere zeitliche Auflösung gefordert ist. Beim Zusammenführen dieser Daten werden die Zeitstempel als Schlüssel genutzt.

Durch diese Teilung der Komponenten in einen spezifischen Aspekt und ein generisches Tool können Profile für beliebige Programme mit relativ geringem Aufwand erzeugt werden, da maximal der Aspekt neu implementiert werden muss.

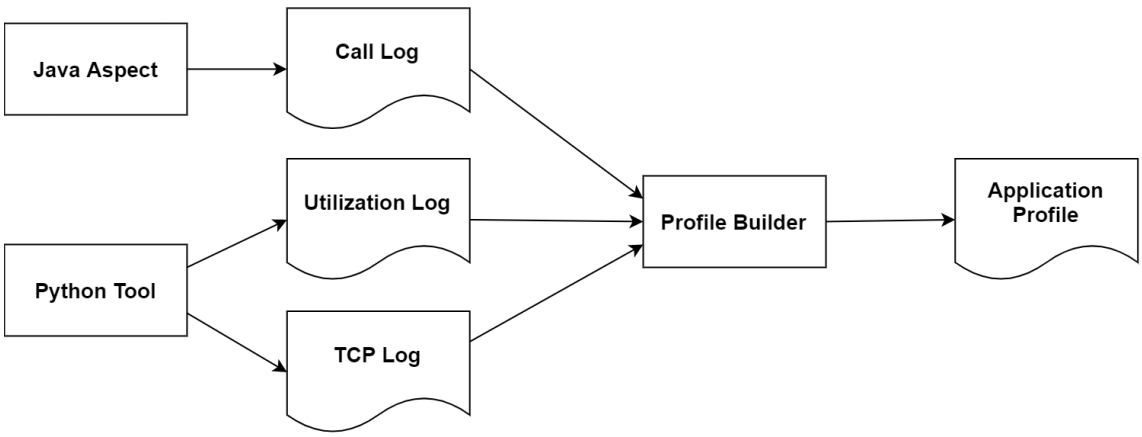

Bild 7.1: Relevante Daten und Komponenten für die Erstellung eines Applikationprofils

7.1.1 Java Aspect

Im Kapitel Stand der Forschung wurde bereits gezeigt, dass ein Tracing von Methoden in Java Applikationen mit AOP und AspectJ möglich ist. In diesem Kapitel wird auf die Implementierung für diese Thesis eingegangen.

Load Time Weaving wird bereits zu Monitoring Zwecken im MVSP verwendet und eine geringe Auswirkung auf die Laufzeitgeschwindigkeit, erlaubt aber einen tiefen Einblick in den Ablauf von Funktionen und wird daher weiterverwendet.

Der in Java implementierte Aspekte loggt den Start- und Endzeitpunkt jeder Funktion mit annähernder Nanosekundengenauigkeit und kann im Listing 7.1 gefunden werden. Die Verarbeitung dieser Daten wird im Kapitel Datentransformation erläutert.

```
@Aspect
public class Monitor {
    private FileWriter csvWriter = new FileWriter("..\\callLog.csv");

    long startTime;
    long startNano;

    public Monitor() throws IOException {
        this.startTime = System.currentTimeMillis() * 1000000;
        this.startNano = System.nanoTime();
    }

    @Around("execution(* *(..)) && !@annotation(de.dvzmv.x.aspects.NoLogging)")
    public Object log(ProceedingJoinPoint pjp) throws Throwable {

        long t = this.startTime + (System.nanoTime() - this.startNano);
        String name = pjp.getSignature().toShortString().split("\\(")[0];
        csvWriter.append("start " + name + " " + t + "\n");

        Object result = pjp.proceed();

        t = this.startTime + (System.nanoTime() - this.startNano);
        csvWriter.append("end " + name + " " + t + "\n");
        csvWriter.flush();

        return result;
    }
}
```

Code Listing 7.1: Call Interceptor Aspekt

7.1.2 Systemdaten

Zum Sammeln der Ressourcenauslastung und dem Loggen der ausgehenden Netzwerkverbindungen wurde ein Programm in Python implementiert, welches alle Prozesse einer Anwendung beobachtet. Hierzu wurde die Bibliothek `psutil` genutzt. Diese Bibliothek bietet ein einheitliches Interface für das Abrufen der Betriebssystemschnittstellen. In der Implementierung werden nur CPU- und RAM-Auslastung auf diese Art ausgelesen, IO- und Netzwerkauslastung sind für das MVSP nicht begrenzend und werden daher nicht vom Profiler betrachtet, obwohl die Simulation damit umgehen könnte. Die Autoren der Bibliothek empfehlen die Ressourcenauslastung nicht öfter als alle 100ms auszulesen, um zuverlässige Ergebnisse zu erhalten. Die Netzwerkverbindungen können in kürzeren Intervallen erfasst werden. Aus diesem Grund werden die Netzwerkverbindungen 10-mal so oft erfasst wie die Ressourcenauslastung. Sollte eine höhere zeitliche Auflösung gewünscht sein, kann ein anderes Tool implementiert werden. Der implementierte Profile Builder kann mit unterschiedlichen zeitlichen Auflösungen umgehen. Die Implementierung kann im digitalen Anhang unter „./Profiler/utilLogger" gefunden werden.

7.2 Datentransformation

In diesem Unterkapitel wird die Transformation der drei Logdateien in ein Applikationsprofil erläutert. Begonnen wird mit der Rekonstruktion des Call-Tree. Anschließend werden TCP-Verbindungen miteinbezogen und die Ressourcenauslastung den ausgeführten Funktionen anteilig zugeordnet. Abschließend wird aus dem Call-Tree ein Applikationsprofil generiert.

7.2.1 Call-Tree

Ein schemakonformes Applikationsprofil wird erzeugt, indem ein Call-Tree aufgebaut und anschließend zur gewünschten Tiefe oder zeitlichen Genauigkeit traversiert wird. Dieser Call-Tree wird aus dem Call-Log erzeugt. Er kann sich von einem Call-Tree, der auf Basis einer statischen Code-Analyse erstellt wurde, unterscheiden. Dies ist der Fall, wenn das analysierte Programm asynchrone Funktionen enthält. Nur wenn das Programm Single-Threaded ist, entspricht der rekonstruierte Call-Tree dem Quellcode. Dieses Verhalten ist akzeptabel, da es hierbei nur darum geht die Reihenfolge der Funktionen und der assoziierten Ressourcenauslastungen einzuhalten.

Es wird davon ausgegangen, dass ausgehend von einer Root-Funktion alle untergeordneten Child-Funktionen später beginnen und früher enden als die übergeordnete Funktion. Ist dies nicht der Fall, handelt es sich nicht um eine Child-Funktion, sondern eine parallel ausgeführte Funktion. Aufgrund dieser Annahme kann ein Call-Tree aus einem gegebenen Call-Log rekonstruiert werden. Hierzu ist es sinnvoll ein weiteres Objekt einzuführen: einen Binärbaum, in welchem Funktionen entsprechend ihrem Startzeitpunkt angeordnet sind, somit ist es möglich effizient die Parent-Funktion einer einzuordnenden Funktion zu finden. Der Zusammenhang dieser Objekte ist in Abbildung 7.2 dargestellt. Die Verwendung eines Binärbaumes ist nicht zwingend nötig, es ist aber effizienter, wenn der Call-Tree asynchrone Funktionen enthält.

Implementiert wurden diese Objekte als doppelt verkettete Listen, wobei der Call-Tree aus verknüpften Funktionsobjekten und der Binärbaum aus Node-Objekten besteht. Jedes Node-Objekt enthält eine Referenz auf ein Funktionsobjekt. Somit die es einfach und effizient möglich ein Child-Node einem Parent-Node zuzuordnen. Hierzu wird durch alle Zeilen der Logdatei iteriert. Ist das erste Wort „start”, wird der Endzeitpunkt der Funktion bestimmt und anhand dieser Zeitpunkte die Parent-Funktion im Binärbaum gefunden. Anschließend wird ein Funktionsobjekt erstellt, dem gefundenen Parent-Node untergeordnet und in den Binärbaum eingefügt. Das gefundene Ende der aktuellen Funktion wird aus den übrigbleibenden Zeilen gelöscht. Diese Schritte werden wiederholt, bis das Ende der Datei erreicht wurde.

Ein Teil der Implementierung kann in Listing 7.2 gefunden werden.

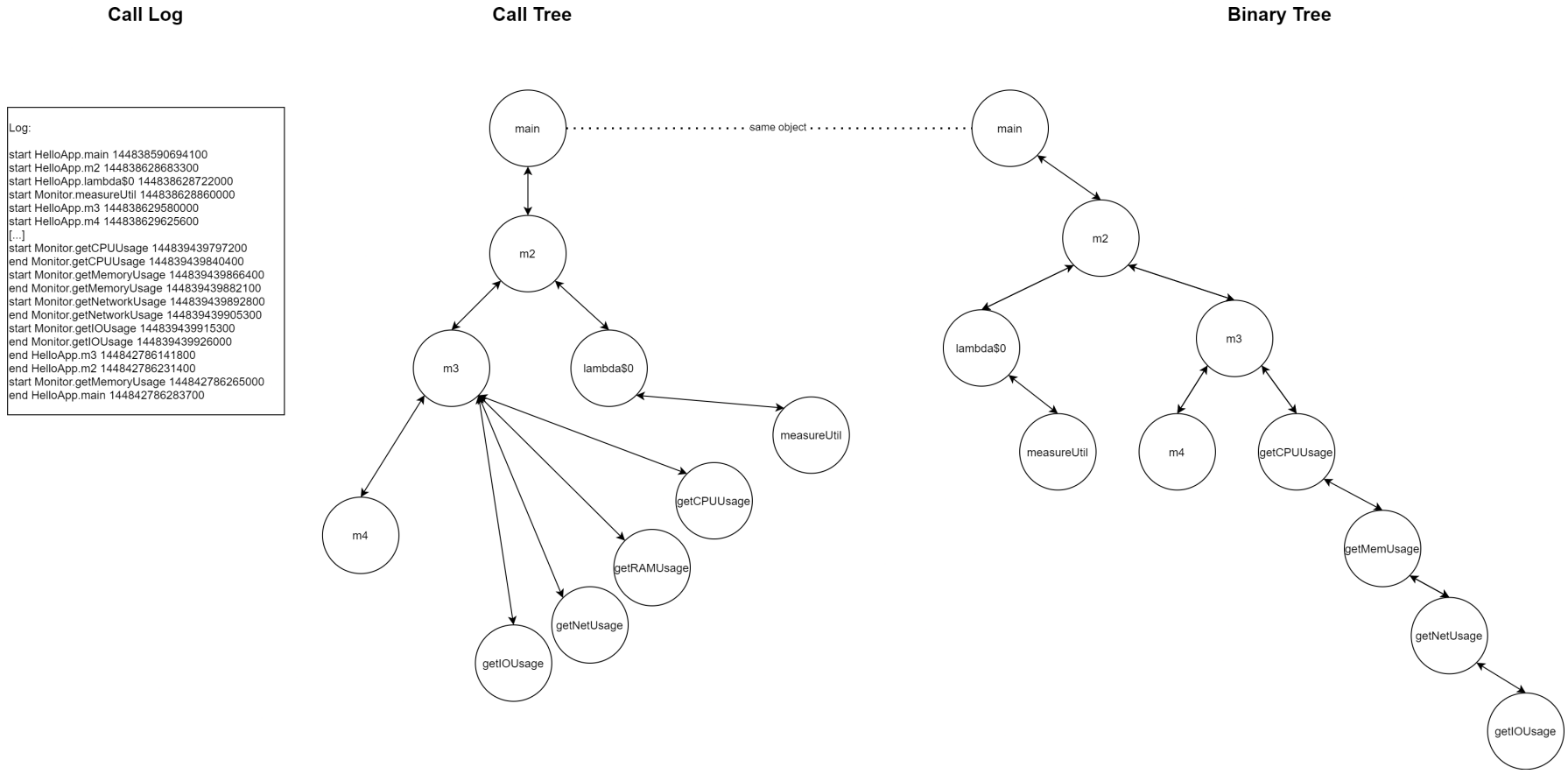

Bild 7.2: Rekonstruktion des Call-Trees

```
def convert(path):
    start = time.time()
    lines = getLines(path)
    maxTs = lines[-1][2]
    root = Function(None, "root", 0)
    bt = BinaryTree(root)
    lenLines = len(lines)
    i = indexOfNext(lines, "startRoot")

    while i < lenLines:
        line = lines[i]
        keyword = line[0]
        identifier = line[1]
        ts = line[2]

        if keyword == "end":
            i += 1
            continue

        if keyword == "endRoot":
            i += indexOfNext(lines[i+1:], "startRoot") + 1
            continue

        end = None
        if keyword == "startRoot":
            bt = BinaryTree(root)
            end = maxTs
        else:
            end = endOfFunction(identifier, lines[i:])

        parent = bt.getParent(ts, end)
        func = Function(parent, identifier, ts, end)
        parent.addChild(func)
        bt.addNode(func)

        endIndex = indexOf(func.id, lines, "end")
        if endIndex is not None:
            del lines[endIndex]

        i += 1
        lenLines = len(lines)

    return root
```

Code Listing 7.2: Funktion zum Aufbau des Call-Trees

7.2.2 Validierung

Der erzeugte Call-Tree kann durch den Vergleich mit dem vorliegenden Quellcode validiert werden. Um das zu vereinfachen, wurde eine Visualisierung in der Form einer interaktiven Web-Ansicht erstellt. In Abbildung 7.3 ist ein Screenshot dieser interaktiven Web-Ansicht abgebildet, diese stellt den gebauten Call-Tree für mehrere Aufrufe des MVSP dar. Der Call-Tree wurde vorher mit einer maximalen Tiefe von 10 und eine Mindestlänge der Funktion von 1ms gekürzt. Die Root-Funktion ist grün, asynchrone Funktionen rot und synchrone Funktionen blau markiert. Mithilfe dieser Visualisierung war es möglich den Algorithmus zur Rekonstruktion des Call-Trees teilweise zu validieren. Eine vollständige Validierung ist aufgrund der großen Menge von Funktionen nicht mit vertretbarem Aufwand möglich.

Bei der Validierung wird ein Fehler offensichtlich: Interaktionen sind nicht erkennbar. Nutzerverursachtes Verhalten und Daemon-Funktionen sind nicht voneinander trennbar. Es sollte aber nur Verhalten gemessen werden, welches von einem Nutzer hervorgerufen wurde.

7.2.3 Messen des Nutzerverhaltens

Um ausschließlich das Nutzerverhalten zu messen, wurde eine weitere Funktion in den Aspekt aufgenommen. Die „signifyRoot"-Funktion führt einen weiteren Identifier ein. Bisher gab es nur die Identifier „start" und „stop". Durch die Identifier „startRoot" und „endRoot" ist es möglich einen Call-Tree der Root-Funktion unterzuordnen. Somit werden Daemon-Prozesse nicht in den Call-Tree aufgenommen, sondern nur Verhalten, welches vom Nutzer verursacht wurde.

Somit ist es auch möglich Interaktionen zu filtern, abzuändern und anzupassen, ohne den gesamten Profilgenerationsprozess erneut zu durchlaufen. Diese Änderungen wurde in Listing 7.2 bereits aufgenommen.

7.2.4 Erfassen von RPC

Ein Kernpunkt dieser Thesis ist, dass die Skalierung von Applikationen durch externe Abhängigkeiten in Form von nicht skalierenden Diensten begrenzt sein kann. Diese Abhängigkeiten können manuell in das entstehende Applikationsprofil eingefügt werden, die automatische Erkennung ist aber zu bevorzugen, da es fehlerresistenter und genauer ist.

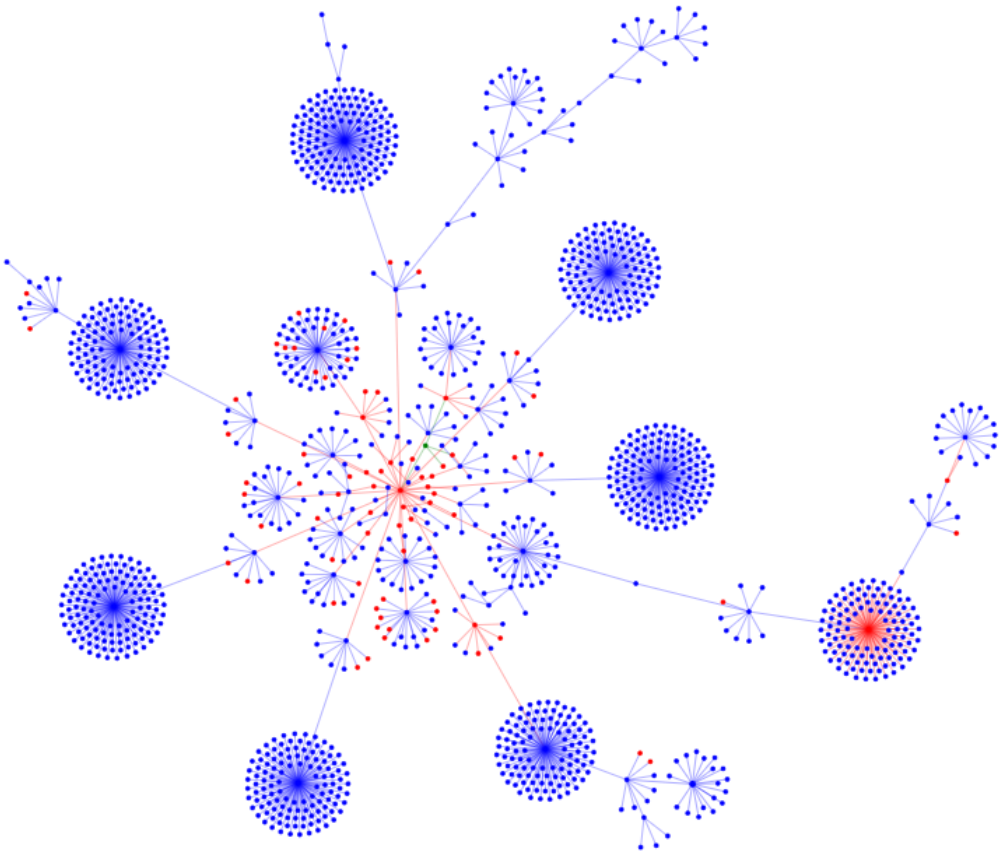

Bild 7.3: Visualisierung des Call-Trees

Um externe Abhängigkeiten automatisch zu erfassen, werden alle bestehenden TCP-Verbindungen in regelmäßigen Abständen erfasst, mit dem TCP-4[1] Tupel identifiziert und gespeichert. Sobald eine Verbindung nicht in den aktuellen Verbindungen enthalten ist, wird die Verbindung mit ihrem Ziel, Start- und Endzeitpunkt in das Log aufgenommen. Mit diesen Daten können Funktionsobjekte erstellt werden, welche in die Interaktionen eingefügt werden können. Ebenso wird das Mapping um diese Funktionen erweitert und Service-Definitionen erstellt, welche vom Nutzer nur noch angepasst werden müssen.

Es ist anzumerken, dass somit nur die IP-Adresse und Port des Ziels erfasst werden, Dienste hinter Proxys können mit diesem Ansatz nicht berücksichtigt werden, da die Hostnamen nicht zugänglich sind. Um Hostnamen zu berücksichtigen wären andere Frameworks und komplexere Verfahren nötig, die über der Transportschicht

[1] Jede TCP-Verbindung kann eindeutig identifiziert werden über die Ursprungs-IP-Adresse, den Ursprungs-Port, die Ziel-IP-Adresse und den Ziel-Port.

eingesetzt werden können.

Alternativ könnte auch ein HTTP-Interceptor entwickelt werden. Dieser könnte alle HTTP-Verbindungen mit ihrem Ziel und der übertragenden Datenmenge erfassen. Allerdings wäre dieser Ansatz programmiersprachenabhängig und müsste genau wie der Aspekt für jede Programmiersprache erneut implementiert werden.

7.2.5 Profilgeneration

Ein Applikationsprofil ist die Abbildung einer Applikation, die sowohl menschenlesbar als auch von der Simulation verwendbar ist. Zum Validieren von Applikationsprofilen wurde ein JSON-Schema erstellt, so können Profile aus verschiedenen Quellen validiert werden und sichergestellt werden, dass diese mit der Simulation kompatibel sind, auch wenn kein direkter Zugang zu dieser besteht. Erstellte Profile könne mit Online-Validieren oder Bibliotheken in verschiedenen Programmiersprachen validiert werden. Das JSON-Schema kann im Anhang in Listing A.2 gefunden werden.

Die Generierung des Profils erfolgt durch die Erzeugung eines Python Dictionary, welches mit der JSON-Bibliothek in einen JSON-String konvertiert und anschließend gespeichert wird. Hierzu werden erst die erhobenen Daten eingelesen und aus dem „CallLog" wird ein Call-Tree gebaut, entsprechend dem in dem im vorigen Unterkapitel beschriebenen Verfahren. Dieser Call-Tree wird anschließend zur spezifizierten Tiefe und minimalen Funktionslaufzeit gestutzt. Das Profilobjekt wird erzeugt, indem aus jedem Node direkt unter dem Root-Node eine Interaktion erzeugt wird.

Die Generierung von Interaktionen erfolgt durch die In-Order-Traversierung des Call-Trees ab dem Root-Node der jeweiligen Interaktion, wobei die Children der Nodes vor der Traversierung nach ihrem Startzeitpunk sortiert werden. Wie bereits beschrieben, kann es Lücken zwischen den Funktionen geben. Diese werden durch Platzhalter gefüllt, insofern die Lücke größer als ein spezifizierter Schwellwert ist. Dieser Schwellwert kann variiert werden, um eine Balance zu finden, zwischen Genauigkeit und Performance der Simulation. Anschließend werden den Funktionen die Ressourcenauslastungen zugeordnet. Die Funktion zum Erzeugen der Interaktion kann in Listing 7.3 und im digitalen Anhang eingesehen werden.

```
def makeInteraction(root, delay, utilLogPath):
interaction = {"name": root.id, "interactionID": root.id,
                "delay": delay, "functions": []}

flattened = []
flattenTree(root, flattened)
markAsyncFunction(flattened)
functions = fillGaps(root, flattened, True)
functions = addUtil(functions, utilLogPath)
functions = addNet(functions, netLogPath)
functions = getFunctionsArray(functions)

if not functions:
    return None

interaction["functions"] = functions

return interaction
```

Code Listing 7.3: Interaktiongeneration

Parallele Funktionen sind hierbei besonders zu beachten, da die erfasste Ressourcenauslastung für das Gesamtsystem oder den Prozess gemessen wurde. Wird über einen Zeitraum eine CPU-Last von 100% gemessen und wurden in diesem Zeitraum aber auch zwei Funktionen gleichzeitig ausgeführt, kann nicht beiden Funktionen eine CPU-Auslastung von 100% zugewiesen werden. Die aufsummierte Ressourcenlast der Funktionen darf zu keinem Zeitpunkt die Messung übersteigen. Hierzu wird ein Gewichtungsfaktor berechnet. Die Funktion zur Berechnung dieses Faktors kann im Anhang in Listing A.3 gefunden werden. Sie implementiert Formeln 7.2 und 7.2.

$$cpuUtil_f = cpuUtil_t * f(X, times) \tag{7.1}$$

$$f(X, times) = \sum_{n=0}^{N} \frac{1}{x_n} * \frac{times_n}{totaltime} \tag{7.2}$$

Die CPU-Auslastung cpuUtil_f der Funktion ergibt sich aus der gemessenen Auslastung in dem Zeitbereich cpu_t mit dem Gewichtungsfaktor multipliziert wird. Die Variable x_n entspricht der Anzahl der Funktionen im Zeitbereich time_n.

Anschließend werden Funktionsobjekte aus den gemessenen TCP-Verbindungen erstellt und in die Abfolge von Funktionen eingefügt.

Abschließend wird die Liste der Funktionen in eine Struktur umgeformt, welche in das Applikationsprofil eingefügt werden kann. Hierzu müssen Callbackfunktionsobjekte durch die ID der Callbackfunktionsobjekte ersetzt werden. Auch hier sind parallele Funktionen gesondert zu betrachten, da nur die längste Funktion Callbacks enthalten darf.

8 Simulation

In diesem Kapitel wird die Implementierung der Simulation und der Visualisierungskomponente entsprechend dem Konzept beschrieben. Hierzu wird das Konzept kurz wiederholt, anschließend folgt eine Beschreibung der Inputs und der implementierten Objekte, welche aus diesen Inputs erzeugt werden. Die Interaktion dieser Objekte wird durch die Engine gesteuert, welche den Kern der Simulation darstellt und ebenfalls beleuchtet wird. Abschließend wird die Entwicklung der Visualisierungskomponente beschrieben.

8.1 Modell

Wie bereits in Kapitel 6 beschrieben, soll ein verteiltes System nachgebildet werden. Hierzu ist es nötig die Komponenten des Systems zu abstrahieren und funktional ähnliche Objekte zu implementieren. Bei diesen Objekten handelt es sich mindestens um Events, Funktionen und Server. Da diese Simulation Event-Basiert modelliert wird, muss jede Funktion als Event in die Simulation eingehen. Da im Idealfall auch (dynamisch) skalierende Services modellierbar sein sollen, wird eine Abstraktion in Form von Services eingeführt. Wie bereits im Konzept beschrieben kann ein Service mehrere Server enthalten und bildet das automatische Loadbalancing und die Skalierung analog zu Kubernetes[1] Pods ab. Funktionen werden dem zuständigen Service zugeordnet, der Service weist die Funktion dem Server mit der geringsten CPU-Auslastung zu. Ist keiner der Services in der Lage den Ansprüchen der Funktion zu genügen, wird die Funktion zurückgestellt, zum nächsten Zeitpunkt, zu dem der Server freie Kapazitäten hat.

Die Simulation besitzt eine Event-Queue, in welcher alle Events, die entsprechend der Input-Verteilung erzeugt wurden, enthalten sind. In diese Queue werden auch während der Laufzeit alle neu erzeugten Events eingefügt. Diese neuen Events können zurückgestellte Funktionen, Neuberechnungen der Endzeitpunkte von Funktionen oder auch die Ausführung von Callback-Funktionen enthalten.

[1] Kubernetes ist eine Container-Orchestrierungslösung

8.2 Inputs

Die Simulation mindestens drei Inputs vom Nutzer: das Applikationsprofil, die Definition der Services, die Definition der Last in Form einer Szenarioverteilung. Optional kann auch ein Mapping angegeben werden.

Diese drei Inputs werden entsprechend Abbildung 8.1 vorverarbeitet, um von der Simulation-Engine verwendet werden zu können. Wie in Abbildung 6.2 dargestellt, muss der Nutzer drei Dateien bereitstellen, eine Definition der Services (diese benötigte keine Vorverarbeitung), mindestens eine Verteilung, aus welcher die Szenarioverteilung (6.3.7) erzeugt wird und das Mapping (6.3.6).

Der vollständige Datenfluss von der Datensammlung bis hin zur Visualisierung kann in Abbildung 8.2 gefunden werden.

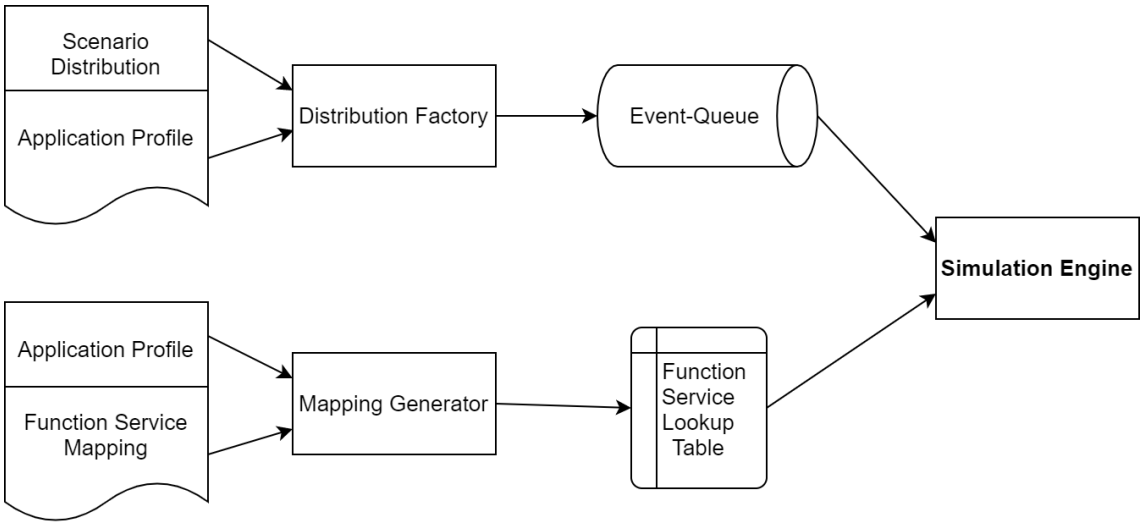

Bild 8.1: Input Vorbereitung

8.2.1 Profil

Während der Profilgenerierung aus den Logdateien, wird ein Applikationsprofil mit einem Szenario erzeugt. Ein Applikationsprofil kann mehrere Szenarien enthalten, hierzu muss ein Nutzer mehrere Profile zusammenführen. Es ist darauf zu achten, dass die ID des Szenarios und der Interaktion im Applikationsprofil einmalig sind.

Das Applikationsprofil kann gegen ein JSON-Schema validiert werden.

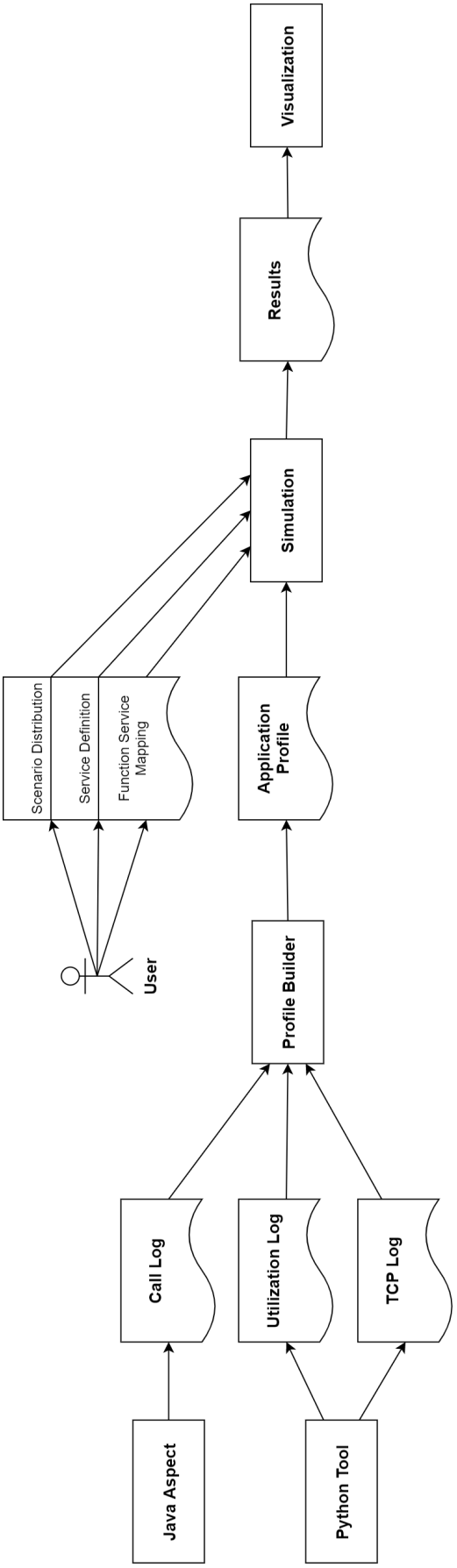

Bild 8.2: Komponenten und Datenflüsse (vollständig)

8.2.2 Mapping

Die Definition eines Mappings ist optional. Es besteht die Möglichkeit die Aufteilung eines Softwaremonolithen in ein verteiltes System zu untersuchen, indem das Mapping variiert wird. Das Mapping erfolgt über den Klassennamen. Aus dem definierten Mapping und dem Applikationsprofil wird eine Lookup-Table für alle Funktionen des Profils generiert, somit ist es effizient möglich den Service zu finden, der für die Funktion zuständig ist.

8.2.3 Service Definition

Es werden automatisch Service-Definitionen für jeden aufgerufenen und beteiligten Service erstellt. Jeder Service besitzt einen „Default Server" - ein Platzhalter für die durch den Nutzer spezifizierte tatsächliche Beschreibung der beteiligten Server, siehe Listing 8.1.

Aus den TCP-Verbindungen, welche während des Profiling aufgenommen wurden, werden Funktionsobjekte erstellt. Diese Funktionsobjekte haben einen Ressourcenbedarf von 1 für die CPU, 0 für RAM und IO und eine Laufzeit entsprechend der Messung. Durch die Anpassung der Kapazitäten des Default Servers kann die Anzahl von Funktionen festgelegt werden, die maximal in einem Zeitraum von diesem Server bearbeitet werden können.

Die Auslastung der CPU oder des RAM könnten als Metrik zum Skalieren des Service verwendet werden. Jeder Service hat mindestens einen Server. Es wird angenommen, dass, wenn ein Service skalieren kann, es keinen Grund für eine Begrenzung gibt. Die automatische Skalierung ist nicht vollständig implementiert.

```
[{ "scaleUpAt": 0.8,
    "scaleDownAt": 0.3,
    "scaleingMetric": "CPU",
    "serviceID": "195.145.109.61:443",
    "scales": false,
    "scale": 1,
    "defaultServer": {
        "maxCPU": 100,
        "maxRAM": 100,
        "maxIO": 100,
        "maxNET": 100
    }
}]
```

Code Listing 8.1: Beispiel Service Definition

8.2.4 Verteilungsgeneration

Bei der DES kann es sich um eine statistische Simulation handeln. Als ein Input für diese Simulation dienen Zufallsverteilungen. Diese Pseudozufallsverteilungen verwenden einen Zufallszahlengenerator, welcher einen Seed verwendet, um den initialen Zustand zu generieren. Wenn dieser Seed bekannt ist und gespeichert wird kann eine Statistische Simulation dennoch reproduzierbar sein.

die Zufallsverteilungen werden mit der NumPy Bibliothek erzeugt. Mit NumPy wereden Zufallszahlen entsprechend der angegebenen Verteilung erzeugt. Diese Zahlen werden als Startzeitpunkte für Szenarien verwendet.

Zu jedem dieser Zeitpunkte wird ein Event mit der ersten Funktion, der ersten Interaktion eines Szenarios, erzeugt und in die Event-Queue eingefügt. Alle Events werden vor dem Beginn der Simulation erzeugt.

Dieses Vorgehen ist ineffizient und kann zu einer hoher RAM-Auslastung führen. Besser wäre die einmalige Auflösung der Funktions-Callbacks und das Nutzen eines Generator Pattern, mit welchem die Funktionsobjekt zur Laufzeit in die Event-Queue eingefügt werden, dies wäre analog zu dem Mechanismus den SimPy verwendet. Für die Beispiele dieser Thesis ist dieser Optimierungsschritt aber nicht notwendig.

8.3 Objekte

Für diese Thesis wurden vier Objekte geschaffen, welche die relevanten Mechanismen abbilden sollen: Funktionen, Events, Server und Services. Die Beziehung dieser Objekte wurde bereits in vorigen Kapiteln erläutert, dieses Kapitel dient der Erläuterung von Details, welche von besonders großer Bedeutung während der Implementierung sind. Die Implementierung kann im digitalen Anhang im Unterordner `/purePy/Application` gefunden werden.

8.3.1 Event

Der Zustand der Simulation kann nur durch Events verändert werden, Events sind somit eine zentrale Datenstruktur der Simulation. Jedes Event gehört zu einem Zeitpunkt, besitzt einen Typ, welcher `None` oder „Recalculation” sein kann, einen

Ziel-Service. Ein Event kann eine Funktion enthalten. Events sind sortierbar, die Sortierung erfolgt nach dem ersten Zeitpunkt, zu dem eine Funktion in die Simulation einging. Somit wird sichergestellt, dass die FIFO-Strategie auch bei zurückgestellten Funktionen eingehalten wird. Enthält ein Event keine Funktion ist die Reihenfolge nicht relevant.

```
class Event:
    def __init__(self, t, type, serviceId, function):
        self.t = t
        self.type = type
        self.serviceId = str(serviceId)
        self.function = function

    def __lt__(self, other):
        if self.function is None:
            return False
        if other.function is None:
            return False

        return self.function.scheduled > other.function.scheduled

    def __eq__(self, other):
        if self.function is None and other.function is None:
            return True
        elif self.function.scheduled == other.function.scheduled:
            return True
        else:
            return False
```

Code Listing 8.2: Event Implementierung

8.3.2 Funktion

Funktionen sind die aktiven Objekte, die ein Server verarbeitet. Jede Funktion hat eine assoziierte Laufzeit und eine Ressourcenauslastung über diese Laufzeit. Funktionsobjekt werden vor dem Start der Simulation aus dem Applikationsprofil erzeugt. Hinzu kommen eine Menge von Laufzeitvariablen, wie die Runtime-ID, start und scheduled. Als „scheduled" wird der ideale Startzeitpunkt bezeichnet. Sind alle Server ausgelastet, wird die Funktion zurückgestellt und „start" auf den neuen Startzeitpunkt gesetzt. Über das Delta zwischen „scheduled" und „start" kann die Verzögerung berechnet werden.

Funktionsobjekte können ihren eigenen Endzeitpunk berechnen, mit den vom Server bereitgestellten Ressourcen. Jede Funktion besitzt auch einen Performance-Gewichtungsfaktor. Benötigt die Funktion Datenmengen, welche nicht in der vorgesehenen Laufzeit von der Festplatte lesbar oder über das Netzwerk übertragbar

sind, wird der effektiv genutzte Anteil der CPU mit diesem Gewichtungsfaktor multipliziert. Hierdurch wird versucht das Verhalten der CPU während des Wartens auf Daten anzunähern.

Funktionen besitzen ebenfalls Vergleichsoperatoren, welche das Sortieren nach dem Startzeitpunkt ermöglicht. So wird sichergestellt, dass das FIFO Prinzip für Funktionen eingehalten wird.

8.3.3 Service

Ein Service ist eine Abstraktion, welche die Abbildung der Skalierung vereinfachen soll. Jeder Service besitzt einen `defaultServer` - ein Server Objekt entsprechend der Implementierung im folgenden Unterkapitel. Mithilfe dieses `defaultServers` werden die Server des Service initialisiert. Services besitzen Funktionen zum Hoch- und Herunterskalieren, zum Monitoring der Auslastung und zum Hinzufügen und Entfernen von Funktionen.

Einem Service kann eine Funktion über die Push-Funktion zugewiesen werden. Besitzt ein Service mehr als einen Server, muss mit jedem Aufruf der Push-Funktion, der Server mit der geringsten Last ermittelt werden. Die Funktion wird dann dem Server mit der geringsten Auslastung zugeordnet. Hat kein Server freie Kapazitäten wird die Funktion zurückgestellt („deferred") und dem frühesten Zeitpunkt zugewiesen, zu dem einer der Server wieder freie Kapazitäten besitzt. Durch die Sortierung der Events und Funktionen wird sichergestellt, dass diese zurückgestellte Funktion so bald wie möglich bearbeitet wird.

8.3.4 Server

Jeder Server hat ein Array mit Funktionsobjekten, welche verarbeitet werden und Ressourcenlimits für CPU, RAM, Netzwerk und IO. Genau wie Services besitzen Server Funktionen für das Zuweisen und Entfernen von Funktionsobjekten.

Beim Zuweisen neuer Funktionen mit der Push-Funktion wird das Funktionsobjekt dem Array aktiver Funktionen hinzugefügt. Mit jedem Push werden neue Endzeitpunkte für alle aktiven Funktionen berechnet. Diese Endzeitpunkte werden an die Simulation-Engine zurückgegeben dort werden Recalculation-Events zu diesen Zeitpunkten erzeugt. Ein Recalculation-Event führt zum Aufruf der Pop-Funktion für alle Server des Service.

Die Pop-Funktion iteriert durch alle Funktionen des Servers und entfernt Funktionen, welche ihren Endzeitpunkt erreicht haben. Insofern diese Funktionen Callbacks besitzen, werden diese an die Simulation-Engine zurückgegeben. Dort werden aus diesen neue Events erstellt, dieser Prozess wird in Kapitel 8.4 genauer erläutert. Nachdem alle fertiggestellten Funktionen entfernt wurden, werden neue Endzeitpunkte für die verbliebenen Funktionen berechnet, diese werden wieder an die Simulation-Engine zurückgegeben. Als Scheduling-Algorithmus wird Round-Robin angenommen.

Da mit jedem Push neue Endzeitpunkte für alle aktiven Funktionen berechnet werden, skaliert die Laufzeit nicht linear mit der Menge der Funktionen der Inputverteilung. Hier bedarf es weitere Optimierung, welche sicherstellen müsste, dass alle Events mit Assoziationen zu fertiggestellten Funktionen aus der Event-Queue entfernt werden. Mit diesem Optimierungsschritt sollte die Laufzeit annähernd linear mit der Anzahl der Events skalieren.

8.4 Simulation-Engine

Im vorigen Kapitel wurden die zentralen Klassen erläutert, welche von der Simulation-Engine verwendet werden. Dieses Kapitel dient der Darstellung der Zusammenhänge der Objekte in der Simulation Loop.

Die Simulation-Loop ist in Abbildung 8.3 dargestellt und besteht aus einer primären Schleife und einer sekundären Schleife. Die Event-Queue und Lookup-Table werden entsprechend Kapitel 7.2 erzeugt und werden der Simulation bereitgestellt. Die primäre Schleife beginnt mit dem Lesen und Entfernen der frühesten Events aus der Event-Queue. Es können mehrere Events zu einem Zeitpunkt gehören. In der sekundären Schleife wird über alle entnommenen Events iteriert, wobei die Liste der aktuellen Events um Callbacks erweitert werden kann. Solange mindestens ein Event in der Liste der aktuellen Events enthalten ist, wird ein Event aus der Queue gelesen. Jedes Event hat einen assoziierten Service. Es wird die Pop-Funktion dieses Services aufgerufen, was dazu führt, dass fertiggestellte Funktionen vom Service entfernt und Callbacks zurückgegeben werden. Wenn Callbacks zurückgegeben werden, werden Events für diese erstellt. Ist das Delay eines Callbacks 0 ns, wird das Event der Liste aktueller Events angehängt. Ist das Delay größer 0 ns wird das Event in die globale Event-Queue eingefügt.

Besitzt das Event eine Funktion, erfolgt ein Lookup der Service-ID mit der ID der Funktion. Dem ermittelten Service wird diese Funktion mit einem Push zugewiesen. Besitzt der Service ausreichend Ressourcen, werden nur die voraussichtlichen Endzeitpunkte zurückgegeben. Hat der Service nicht ausreichende Ressourcen, wird die Funktion mit einem neuen Startzeitpunkt ebenfalls zurückgegeben. In diesem Fall wird für diese Funktion ein neues Event in die globale Event-Queue eingefügt.

Dieser Vorgang wird wiederholt, bis kein weiteres Event in der Liste aktueller Events enthalten ist. Anschließend wird die Auslastung der Services berechnet und zu Monitoring-Zwecken in die Logging-Queue eingefügt.

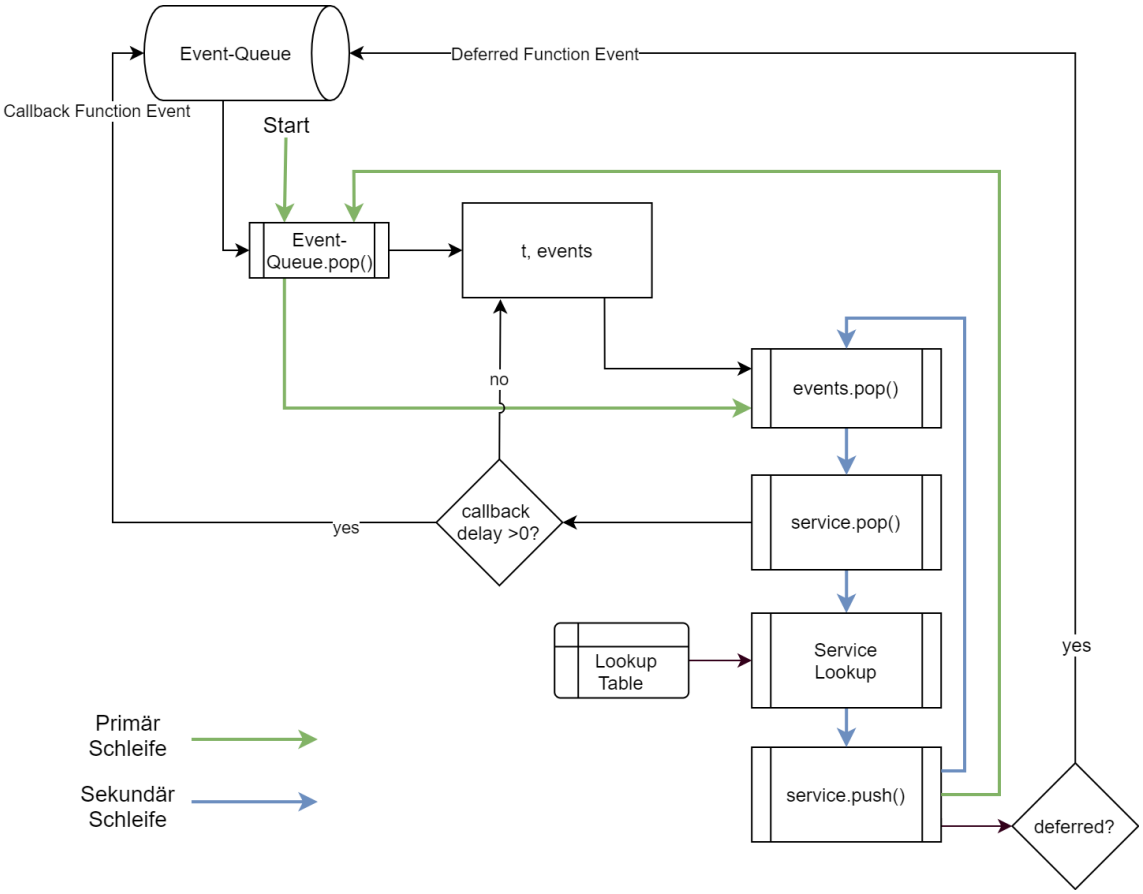

Bild 8.3: Simulation-Engine

8.5 Monitoring

Für das Monitoring der Simulation wird eine Thread-Safe Implementierung einer Queue genutzt. Die Einträge der Queue besitzen einen Zeitstempel, zwei Identifier und ein Array mit gemessenen Werten, siehe Listing 8.3. Somit ist es möglich aus

den Einträgen der Queue ein Dictionary und aus dem Dictionary einen Pandas DataFrame zu erzeugen. Pandas stellt eine Reihe von Datenverarbeitungs- und Visualisierungsfunktionen bereit. Pandas DataFrames können mit dem Dash Framework auch in einem Web-Dashboard dargestellt werden.

Der Key bestimmt in welchem Plot der Graph angezeigt wird und das Filterattribut ermöglicht ein genaueres Filtern über ein Drop-Down Menü. Im Kapitel 8.6.4 wird auf die Integration des Monitoring in die Simulation genauer eingegangen.

```
[
    (key, t, filter, [value1, value2, ...])
]
```

Code Listing 8.3: Monitoring Queue Eintrag

8.6 Visualisierung

In diesem Kapitel wird auf die Visualisierung relevanter Daten aus der Simulation beleuchtet. Es wird mit allgemeinen Überlegungen begonnen, welche Daten visualisiert werden sollen und welche Visualisierungstechniken hierfür geeignet sind. Anschließend wird auf die Implementierung der ausgewählten Techniken mit Dash und Pandas eingegangen. Abschließend folgt eine Erläuterung der Integration der Visualisierungskomponente in die Simulation.

8.6.1 Visualisierungsziele

Bei der Auswahl der visualisierten Metriken wurden sowohl Grundlagen der Queueing Theory als auch Erkenntnisse des Experteninterviews berücksichtigt.

Von besonderer Relevanz sind: die Auslastung der Server, die durchschnittliche Wartezeiten und die Anzahl eingehender und fertiggestellter Funktionen.

Gibt es eine Diskrepanz zwischen eingehenden und fertiggestellten Interaktionen, die über eine zeitliche Verschiebung hinausgeht, deutet dies auf einen Engpass hin, welcher über die Auslastung der Server identifiziert werden kann. Die Auslastung der Server wird nicht nach der Definition der Queueing Theory bewertet, sondern nach den für Server üblichen Metriken CPU-, RAM-, IO- und Netzwerk-Auslastung.

Ebenfalls relevant ist die Zeit, die eine Interaktion benötigen sollte, verglichen mit der Zeit, die sie tatsächlich benötigt. Das Delta wird als Delay bezeichnet, auch an diesem Delta kann eine Systemüberlastung erkannt werden.

8.6.2 Visualisierungstechniken

Da es sich bei den gesammelten Daten um Zeitserien handelt sollten, Visualisierungstechniken angewandt werden, welche dies abbilden können. Gleichzeitig sollte die gewählte Darstellung dem Nutzer vertraut vorkommen. Im Idealfall ist die Entwicklung des Systems über den gesamten Zeitbereich sofort erkennbar. Um dies zu realisieren, werden vier Graphen gleichzeitig auf einem Dashboard abgebildet, entsprechend Abbildung 8.4.

Der linke obere Graph kombiniert die Darstellung der eingehenden und fertiggestellten Interaktionen als einfache Liniendiagramme. Auf demselben Diagramm ist auch der Anstieg der letzten 10 Sekunden als Balkendiagramm dargestellt, sowohl für die Input-Verteilung als auch für die fertiggestellten Funktionen. Die maximale Nutzeranzahl kann an diesem Diagramm abgelesen werden. Kommt es zu einem Engpass während der Simulation, wird die Output-Rate des Gesamtsystems durch die Output-Rate der begrenzenden Komponente bestimmt, diese entspricht dem Maximum des Output Diagramms.

Der Engpass selbst kann über den Plot oben rechts identifiziert werden. Hier kann die Auslastung aller Services ausgelesen werden. Eine qualitative Aussage über die Überlastung des Systems lässt sich aus der Antwortzeit im Plot unten links ableiten. Ist das Delay größer 0 ns ist mindestens ein Service überlastet. Der Plot unten rechts dient nur der Visualisierung von simulationsinternen Daten und ist besonders während der Entwicklung hilfreich.

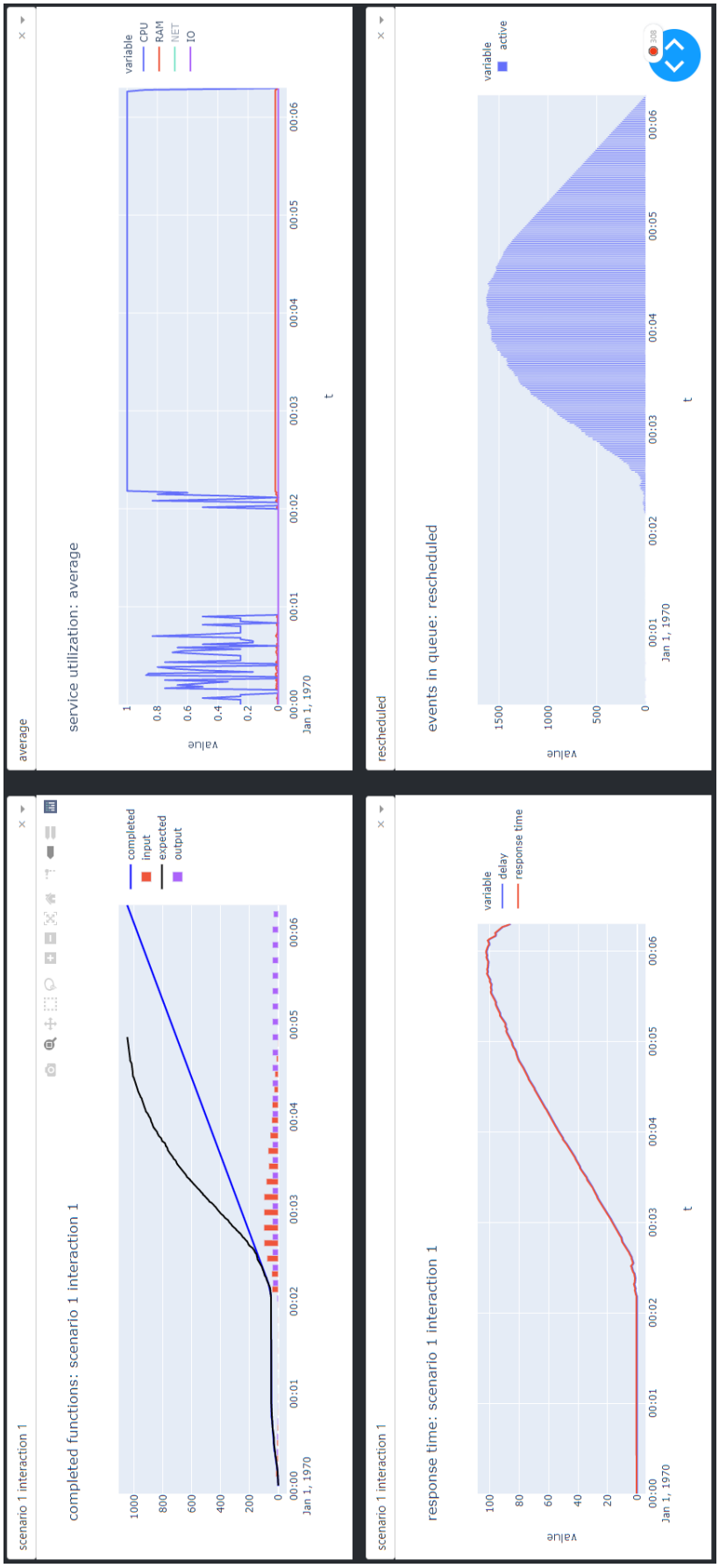

Bild 8.4: Visualisierungs-Dashboard

8.6.3 Implementierung

Die im vorigen Kapitel erläuterten Visualisierungstechniken werden in einem Web-Dashboard kombiniert. Dieses Dashboard wird mit dem Dash Framework implementiert. Dash ermöglicht es Web-Dashboards allein im Backend und in reinem Python zu erstellen. Dash baut auf der Plotly Bibliothek auf, Plotly wiederum verwendet NumPy[2]. All diese Frameworks harmonieren, da sie sich auf Pandas DataFrames als Datenformat stützen. Pandas ist eine Bibliothek zur Datenverarbeitung und Visualisierung.

Dash nutzt React[3] als Front-End Framework, React wiederum ist auf die Verwendung von JSON ausgelegt. Daher werden Layouts und Komponenten als Python Arrays definiert, welche in JSON umgewandelt und an das Front-End weitergegeben werden, wo die Daten von React in HTML-Elemente umgewandelt werden. Dash baut auch auf dem Flask[4] Framework auf, somit kann eine etablierte Annotations-Syntax für interaktive Elemente genutzt werden.

Bevor die gesammelten Daten visualisiert werden können, müssen sie in ein besseres Format konvertiert werden. Hierzu wurde die Funktion in Listing 8.4 implementiert. Die Logging-Queue der Simulation wird in ein Dictionary von Dictionarys konvertiert. Aus diesen Dictionarys werden Pandas DataFrames erzeugt, welche anschließend für das Plotten von Figures verwendet werden können. Die „t"-Spalte wird als Index im DateTime Format festgelegt. Die DataFrames werden in eine globale Struktur „dfs" gespeichert, somit können sie von den getakteten Plotting-Funktionen genutzt werden. Wird die Observation Queue blockweise abgerufen, kann das Plotten parallel zur Laufzeit der Simulation erfolgen.

[2]NumPy ist eine Bibliothek für die Lösung Mathematischer Probleme in Python. NumPy ist in großen Teilen in C implementiert, NumPy Funktionen sind somit oft schneller als Implementierungen in Python.

[3]React ist ein von Facebook entwickeltes und weit verbreitetes Front-End Framework zur Erstellung von User-Interfaces für Web-Applikationen.

[4]Flask ist ein Python WSGI-Mikro-Framework, mit welchem es möglich ist Web-Applikationen in Python zu programmieren.

```
def readFromQueue(sim):
    global dfs
    dfs = dict()

    for value, df in sim.observationQueueToDict().items():
        x = pd.DataFrame.from_dict(df)
        x["t"] = pd.to_datetime(x["t"], unit='ns')
        x.set_index("t", inplace=True)

        if value in dfs:
            dfs[value].append(x)
        else:
            dfs[value] = x
```

Code Listing 8.4: Event Implementierung

In der Funktion „observationQueueToDict" wird die Logging-Queue elementweise ausgelesen, hierbei werden mehrere Dictionarys erzeugt. Welchem Dictionary ein Eintrag hinzugefügt wird, wird durch den Key bestimmt. Nach der Konvertierung in einen DataFrame entsteht hierdurch eine Zeitserie, die nach Filterattributen filterbar ist. Auf das Format der Logging-Queue wurde in Kapitel 8.5 eingegangen.

Die Plots werden beim initialen Laden des Dashboards direkt eingefügt und anschließend durch eine getaktete Funktion alle 2 Sekunden erneuert, insofern neue Daten vorliegen.

Ein Plot besteht aus einem Figure-Objekt und weiterer Konfiguration. Der Code für einen Beispiel-Plot ist in Listing 8.5 zu finden. In diesem Beispiel wird die Kurve der fertiggestellten Funktionen gezeichnet. Der hierzu nötige DataFrame wird über das Keyword identifiziert und aus dem globalen Objekt „dfs" entnommen. Dieser DataFrame kann tausende Events pro Sekunde enthalten, daher erfolgt ein „resampling", wodurch die Anzahl der Events auf eines pro Sekunde reduziert wird, was die Zeit zum Plotten verringert. Sollte es Lücken in dem Datensatz geben, werden die durch das Füllen mit dem letzten Wert aufgefüllt. Dies ist sinnvoll, da es sich um eine Event-basierte Simulation handelt und Werte sich zwischen Events nicht ändern können, auch wenn keine Daten vorliegen, kann der letzte vorliegende Wert angenommen werden.

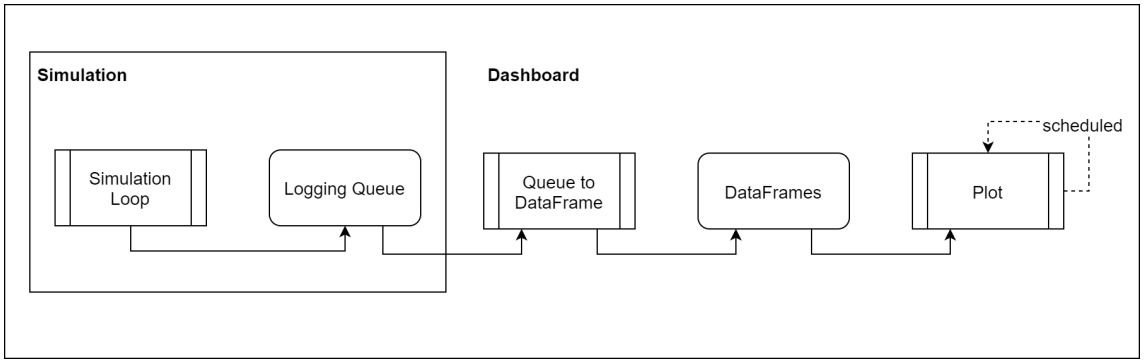

Bild 8.5: Visualisierungsloop der Simulation

```
    df = dfs[keyword]
    data = df.loc[df["identifier"] == identifier].resample(rule = "1s").last().
        interpolate("pad")
    fig = go.Scatter(
        x=data.index,
        y=data["completed"],
        mode="lines",
        line=go.scatter.Line(color="blue"),
        name="completed"
    )
    dcc.Graph(id=f"{keyword} {identifier}",
            figure=fig,
            style={"height": "24rem"}, animate=True)
```

Code Listing 8.5: Beispiel Plot mit Pandas und Dash

Abhängig von den zu visualisierenden Daten werden unterschiedliche Plots generiert, der Prozess ähnelt aber immer dem Demonstrierten.

8.6.4 Integration der Simulation

Die Integration der Simulation in das Dashboard erfolgt indem ein Simulations-Objekt mit dem Starten des Servers erzeugt wird, siehe Abbildung 8.5. Das Simulations-Objekt enthält eine Logging-Queue, welche in DataFrames konvertiert wird, welche wiederum die Grundlage für die weiteren Datenverarbeitungsschritte und das Plotting sind. Das Konvertieren könnte zur Laufzeit der Simulation erfolgen, implementiert wurde es aber sequenziell, das Plotting erfolgt dennoch alle 5 Sekunden. Somit wäre die Erweiterung um eine Online-Visualisierung einer laufenden Simulation einfacher.

9 Verifizierung und Validierung

In diesem Kapitel werden die Ergebnisse der Verifizierung und Validierung der Simulation präsentiert. Es wird mit einer simplen Validierung mittels Queueing Theory begonnen, es folgt die Validierung des Profilers in zwei Schritten und ein einfaches konstruiertes Beispiel. Abschließend wird auf die Erfüllung der Anforderungen des DVZ eingegangen.

9.1 Queueing Theory

Wie im Kapitel 6 bereits erläutert, kann die Simulation als Queueing Network betrachtet werden und kann somit auch mittels Queueing Theory validiert werden. Jeder Server in diesem Queueing Network besitzt eine maximale Verarbeitungsgeschwindigkeit. Ist die Verarbeitungsgeschwindigkeit des Servers geringer als die Frequenz der eingehenden Funktionen, kommt es zu einer Überlastung. Im Fall einer Überlastung entspricht die Output Geschwindigkeit des gesamten Netzwerks der Output Geschwindigkeit der langsamsten Komponente. Ist das Netzwerk nicht überlastet, wenn also die Frequenz der eingehenden Funktionen in jede Komponente geringer ist als die Verarbeitungsgeschwindigkeit der jeweiligen Server, so entspricht die Output-Verteilung der Input-Verteilung, allerdings verschoben um die Verarbeitungsdauer.

Die folgende Untersuchung soll sicherstellen, dass die Simulation die Auslastung von Ressourcen und die Auswirkungen dessen auf die Laufzeit von Funktionen erwartungsgemäß abbildet. Die Dateien zur Definition dieses Experiments können im digitalen Anhang unter„/demoFiles/ValidierungQueueingTheory” gefunden werden.

Es wird ein Service mit einem Server definiert, dieser Server hat einen CPU-Kern und 100 MB RAM. Alle Funktionen werden auf den Default Service gemappt. Es wird eine Dreiecksverteilung[40] von Funktionen mit einer Laufzeit von 100 Sekunden und einem Volumen von 200 Funktionen definiert. Jede dieser Funktionen benötigt eine Sekunde auf der CPU bei einer Auslastung von 100%. Die Output-Frequenz entspricht somit maximal einer Funktion pro Sekunde im Durchschnitt über die

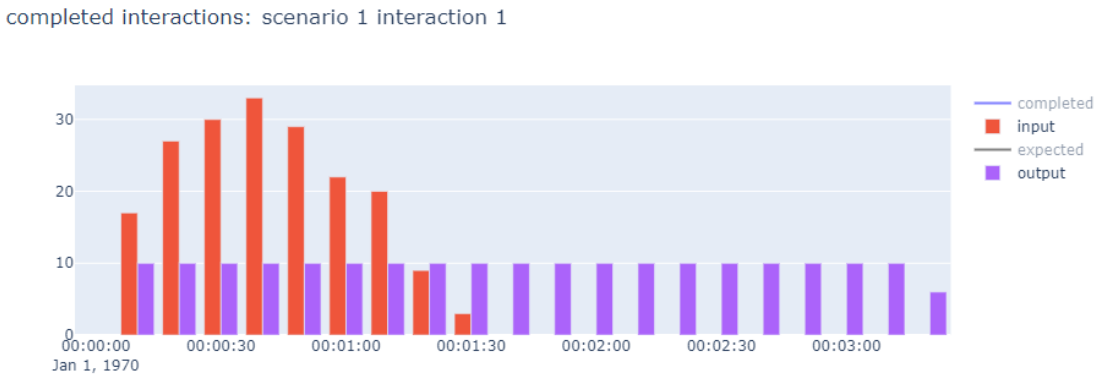

Bild 9.1: Validierung Queueing Theory: Queueing Network mit einer Queue

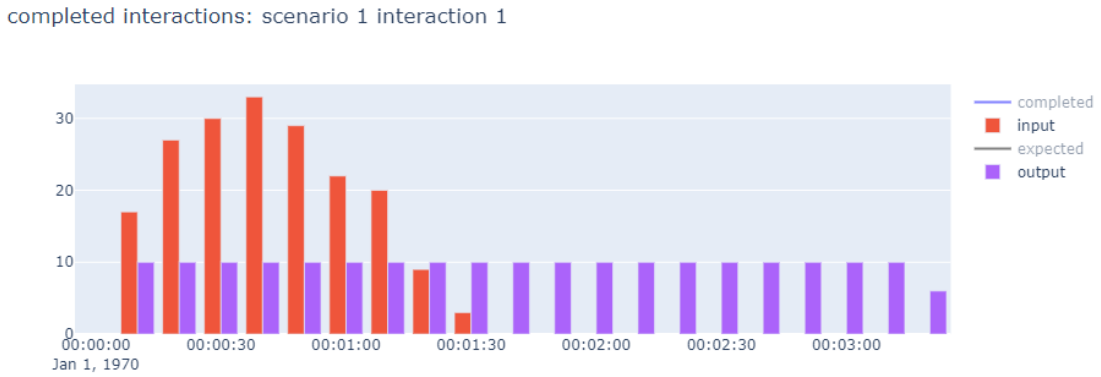

Bild 9.2: Validierung Queueing Theory: Queueing Network mit drei Queues

Laufzeit des Experiments. Die Inputfrequenz liegt bei zwei Funktionen pro Sekunde. Die Laufzeit der Simulation sollte somit bei etwas über 200 Sekunden liegen. Nicht exakt 200, da der Server zu Beginn der Simulation durch die Dreiecksverteilung nicht vollständig ausgelastet ist. Die maximale Output-Frequenz sollte bei einer Funktion pro Sekunde liegen.

Der Versuch bestätigt die Annahme, es wird dieselbe Anzahl von Funktionen verarbeitet, allerdings mit der maximalen Frequenz von einer Funktion pro Sekunde, oder 10 Funktionen in 10 Sekunden, wie es in 9.1 dargestellt ist.

Der folgende Versuch dient der Validierung, dass die maximale Output-Frequenz, der Output-Frequenz der Überlasteten Komponente entspricht. Der Aufbau bleibt ähnlich, allerdings werden 2 Funktionen ergänzt, welche jeweils auf einen weiteren Service gemappt werden. Dieser Service besitzt genug Ressourcen, um keinen

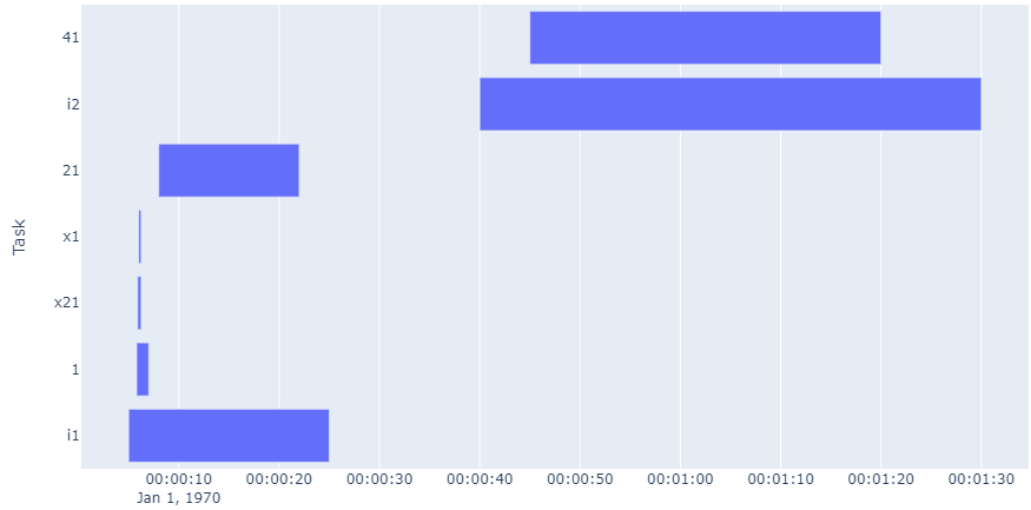

Bild 9.3: Gantt-Diagramm aller Funktionen

Engpass darzustellen. Die Kurve sollte annähernd identisch sein. Das Ergebnis in Abbildung 9.2 entspricht der Erwartung und ist mit einer zeitlichen Auflösung von 10 Sekunden nicht von Abbildung 9.1 zu unterscheiden.

Die Funktionsweise der Simulation als Queueing Network ist somit erwiesen.

9.2 Validierung des Profilers

Zur Validierung der Rekonstruktion des Call-Tree wurde ein Beispiel Call-Log erstellt, siehe Listing A.1. Die Applikation, die hierdurch abgebildet wird, wurde in zwei Interaktionen aufgerufen, eine dieser Interaktionen besitzt auch parallele Funktionen. In Abbildung 9.3 ist ein Gantt-Diagramm der Funktionen abgebildet, wie sie aus dem Log ausgelesen wurden. Es ist klar zu erkennen, dass Funktionen x21 und x1 im Zeitbereich der Funktion 1 liegen und sich überschneiden. Diese Funktionen sollten also als parallel und der Funktion 1 untergeordnet erkannt werden. Funktion 1 und 21 sollten der Interaktion i1 untergeordnet sein, so wie Funktion 41 der Interaktion i2 untergeordnet sein sollte.

In Abbildung 9.4 ist der rekonstruierte Call-Tree abgebildet. Es ist klar zu erkennen, dass die Erwartung erfüllt wurde. Dem grünen Root-Node sind die Interaktionen untergeordnet, die parallelen Funktionen wurden auch als solche erkannt und entsprechend rot markiert.

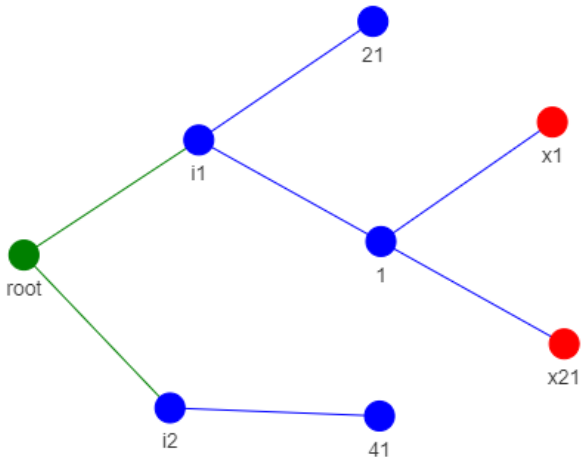

Bild 9.4: Rekonstruierter Call-Tree

Es wurden auch weitere erfolgreiche Tests mit Logdateien des MVSP durchgeführt, welche anhand des Quellcodes teilweise validiert wurden. Eine Vollständige Validierung wäre zu zeitaufwendig, da es sich um 100.000 Funktionen oder mehr handeln kann. Hiermit sollte davon ausgegangen werden können, dass Call-Trees erwartungsgemäß rekonstruiert werden.

9.3 Rekonstruktion einer Messung

Für die Rekonstruktion einer Messung müssen mehrere Aspekte des Profilers korrekt funktionieren. Der Call-Tree muss korrekt rekonstruiert und traversiert werden, die Messung der der Systemauslastung muss akkurat sein und die Auslastung muss den korrekten Funktionen zum korrekten Anteil zugeordnet werden. Die Rekonstruktion und Traversierung des Call-Trees wurde bereits validiert, kommt es zu Abweichungen in der Rekonstruktion eines einzigen Aufrufes, sollte dies durch die Messung der Auslastung oder der Zuweisung der Auslastung zu den Funktionen bedingt sein.

Für diesen Versuch wird der gesamte Prozess wie in Abbildung 6.2 zum ersten Mal durchlaufen. Der Profiling Aspekt wird in das MVSP eingefügt, das Python Tool zur Messung der Auslastung durch Java Prozesse wird gestartet und das MVSP wird auf einem Server deployt. Es wird die Landing Page mehrfach aufgerufen und einige Buttons gedrückt, was weitere Interaktionen für das Nachladen von Inhalten

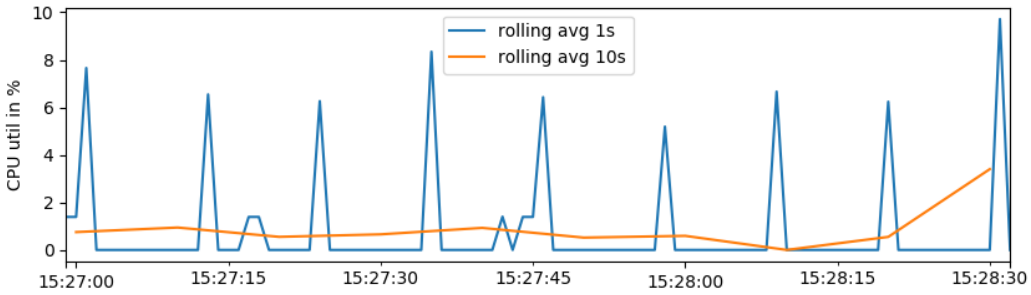

Bild 9.5: Gemessene Service Auslastung

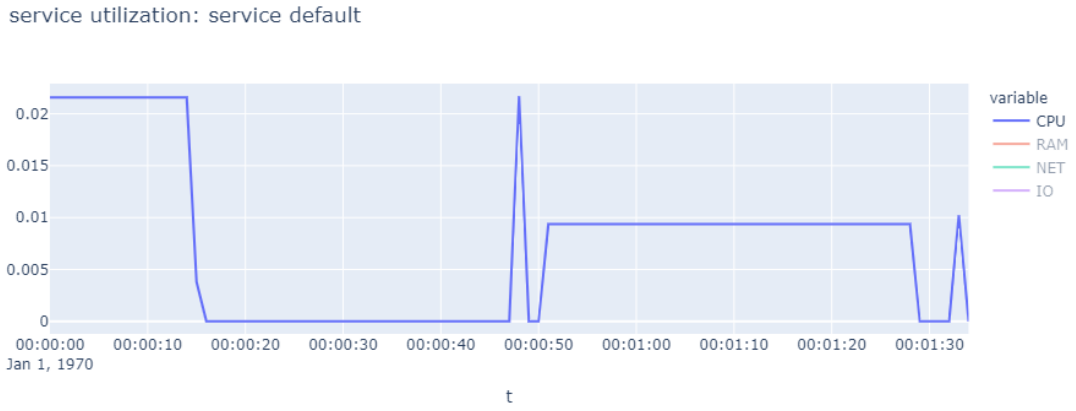

Bild 9.6: Simulierte Service Auslastung

auslöst. Anschließend wird der Server und das Python Tool beendet und der Profile Converter mit den eben erzeugten Logdateien ausgeführt. Somit wurden eine Service-Definition und ein Applikationsprofil erzeugt. Die Service-Definition wird angepasst, um dem Laptop, auf dem die Messung durchgeführt wurde, zu entsprechen. Es wird keine Verteilung erzeugt, stattdessen wird das Szenario nur einmal zu Sekunde 0 erzeugt. Die Auslastung des Default Servers ist in Abbildung 9.6 abgebildet. In Abbildung 9.5 ist die Messung über den selben Zeitraum abgebildet. Im Idealfall wären Abbildung 9.5 und 9.6 annähernd identisch, dem ist nicht so. In Abbildung 9.7 ist das Gantt-Diagramm der Interaktionen abgebildet, in diesem Versuch wurde mit einer Traversierungstiefe von 1 gearbeitet, was bedeutet, dass jede Interaktion nur eine Funktion enthält. Diese Funktionen haben teilweise eine Länge von mehreren Sekunden, die Ressourcenauslastung dieser Funktionen entspricht dem Durchschnitt der gemessenen Auslastung über die Laufzeit der Funktion. Vor diesem Hintergrund kann gesagt werden, dass es Ähnlichkeiten zwischen Abbildungen 9.5 und 9.6, wenn die Durchschnitte über die Zeiträume in 9.7 genommen werden.

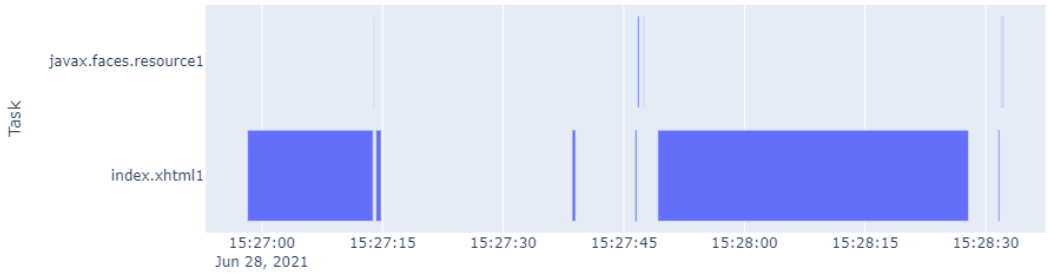

Bild 9.7: Gantt-Diagramm Interaktionen

Leider ist eine höhere zeitliche Auflösung durch eine höhere Traversierungstiefe nicht erreichbar, da die gemessene CPU-Auslastung nicht 10-mal pro Sekunde gemessen wird, sondern circa einmal alle 10 Sekunden. Bei einer höheren Traversionstiefe gäbe es Funktionen mit einer Laufzeit von weniger als 10 Sekunden, wodurch diesen Funktionen eine Auslastung von 0 zugewiesen werden würde. Um das zu vermeiden, müsste ein gleitender Durchschnitt über mehr als 10 Sekunden berechnet werden und die Ressourcenauslastung der Funktionen aus diesem Durchschnitt entnommen werden. Dies verringert die Genauigkeit der Simulation erheblich, die Zuweisung einer höheren Auslastung zu einer Funktion ist somit nicht möglich.

Die Validierung scheitert an der akkuraten Messung der Ressourcenauslastung.

9.4 Konstruiertes Beispiel

Die vorige Simulation konnte die Messung nicht rekonstruieren. Vermutlich lag dies an der zu sporadischen Messung der CPU. Um zu überprüfen, ob dies tatsächlich der Fall ist, wird eine Beispielapplikation entwickelt, welche anschließend simuliert wird. Somit sollte ein Fehler in der Simulation ausgeschlossen werden können.

Für diesen Versuch wurden 2 Server und ein Client entwickelt. Der Client erzeugt 100 asynchrone und parallele GET-Anfragen, welche Server1 Aufrufen. Server1 kann mit maximal 1000 aktiven Verbindungen umgehen und ruft mit jedem Aufruf Server2 auf. Server2 ist Single-Threaded und führt mit jedem Aufruf eine Funktion mit einer Laufzeit von 100 ms auf, welche den CPU-Kern zu 100% auslastet.

Für die Messung der Auslastung des physischen Servers, auf dem Server1 und Server2 ausgeführt werden, wird dasselbe Programm verwendet, wie auch im vorangegangenen Beispiel. Allerdings wird nicht die Auslastung pro Prozess betrachtet, sondern

Bild 9.8: Validierung Versuchsaufbau 3

die Auslastung des gesamten Systems. Bei diesem Ansatz kann die CPU-Auslastung zuverlässig alle 100 ms ausgelesen werden. Da auf dem physischen Server auch andere Prozesse ausgeführt wurden, kann die gemessene Auslastung nicht allein der Applikation zugeordnet werden.

Da alle Funktionen und Zusammenhänge dieses Aufbaus bekannt sind, kann das Applikationsprofil manuell erstellt werden. Die Inputs dieser Simulation können im digitalen Anhang unter /validationTestProgram gefunden werden.

Der Client erzeugt 100 Anfragen mit einem Abstand von jeweils 10 ms. Die Verteilung der Simulation wurde analog erzeugt. Dies sorgt für eine Auslastung der Server-CPU von 100% für circa 10 Sekunden, wie sowohl aus der Messung als auch der Simulation hervorgeht. Auch die Wartezeit der Funktionen ist annähernd gleich. Die Simulation sagt eine maximale Antwortzeit von 10 Sekunden vorher, die gemessenen Ergebnisse zeigen eine maximale Wartezeit von circa 10,17 Sekunden. Die Graphen der Messungen können in Abbildungen 9.8 und 9.10 gefunden werden.

Zumindest für dieses einfache Beispiel konnte die Simulation validiert werden. Die Genauigkeit entspricht den Anforderungen, da die Abweichung bei weniger als 30% lag. Somit kann auch mit Sicherheit gesagt werden, dass die Rekonstruktion der Messung im vorigen Kapitel an der akkuraten Messung der Auslastung scheiterte.

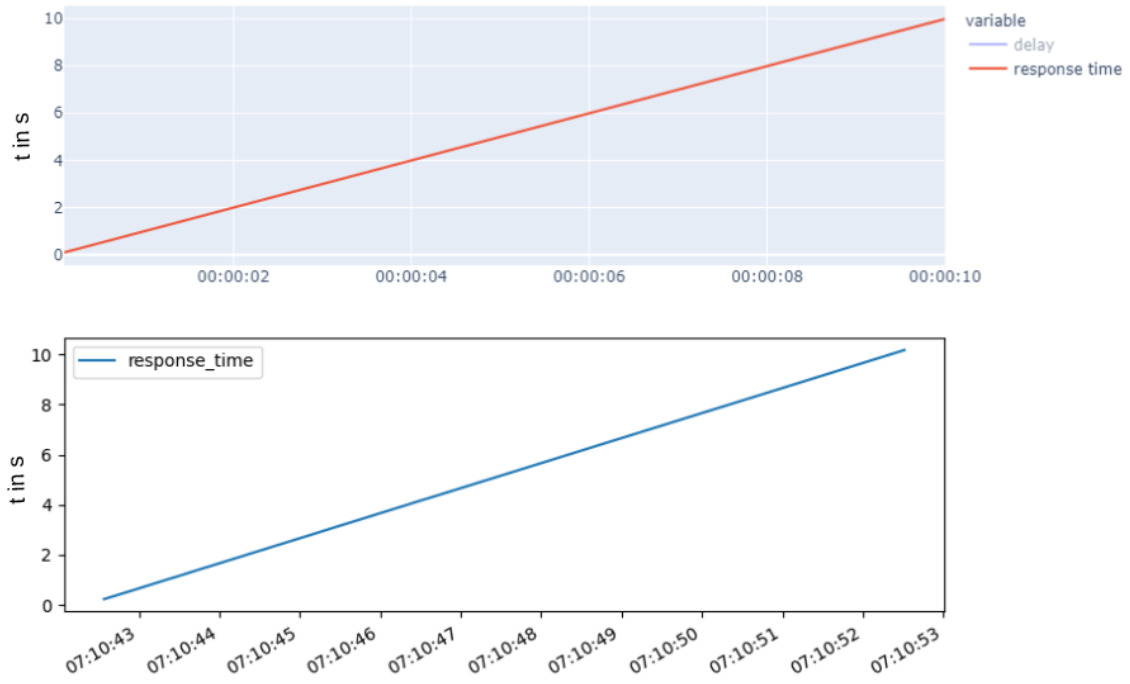

Bild 9.9: Vergleich simulierter Antwortzeit (oben) mit gemessener Antwortzeit (unten)

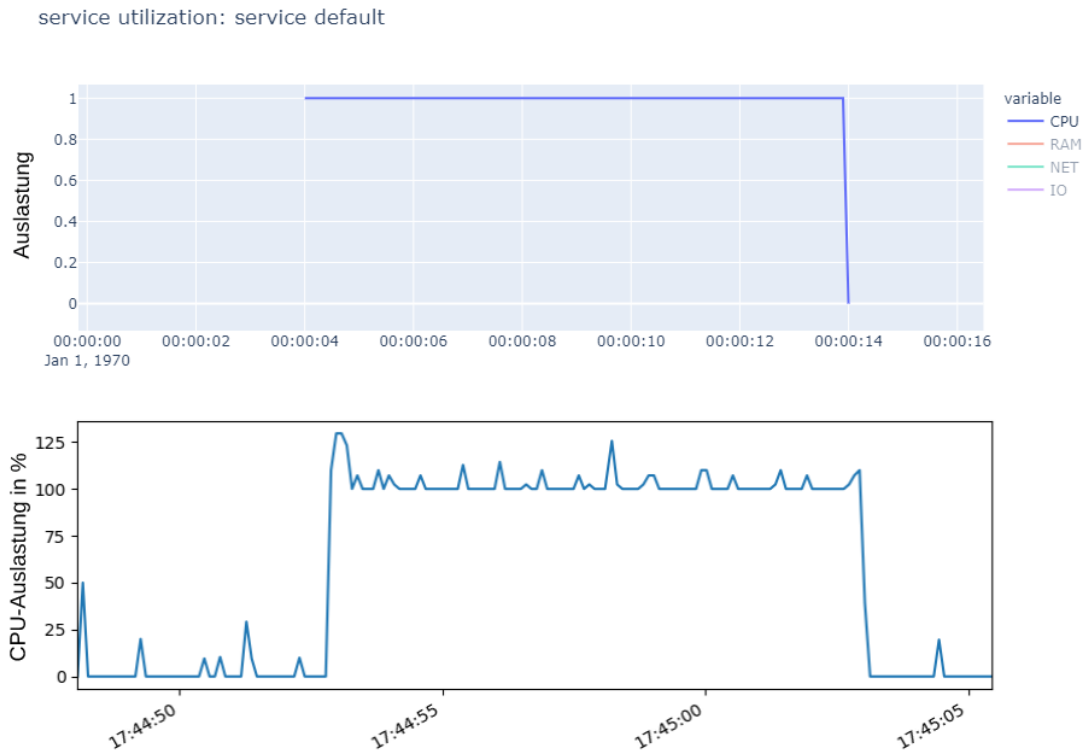

Bild 9.10: Vergleich simulierter CPU-Auslastung (oben) mit gemessener CPU-Auslastung (unten)

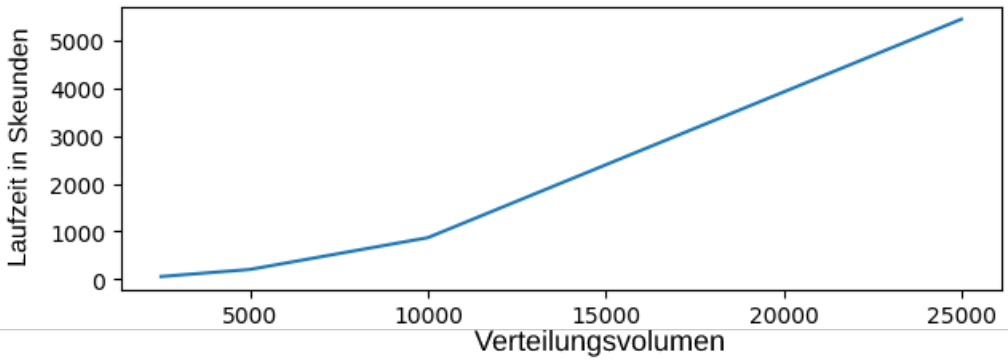

Bild 9.11: Laufzeitgeschwindigkeit der Simulation über Volumen der Szenarienverteilung

9.5 Erfüllung der Anforderungen

Da diese Thesis in Kooperation mit dem DVZ-MV entstanden ist, gab es Anforderungen an die Implementierung, die in Kapitel 4 erläutert wurden. Diese wurden in drei Gruppen unterteilt: Anforderungen an die Simulation, die Visualisierung und an die Benutzbarkeit.

Die Anforderungen an die Simulation sind die Folgenden: es soll die maximale Nutzeranzahl einer Applikation bestimmt werden können, eine Abweichung von weniger als 30% wäre ideal, begrenzende Komponenten sollen identifiziert werden können und die Laufzeitgeschwindigkeit soll ausreichen um, 25.000 Nutzer in 24h zu simulieren.

In den vorigen Tests wurde nur die Laufzeitgeschwindigkeit noch nicht getestet. Hierzu wurde eine Dreieckverteilung mit einem Volumen von 25.000 Szenarien über einen Zeitraum zwischen 9 und 20 Uhr mit einem Höhepunkt um 12 Uhr erzeugt. Genutzt wurde ein Applikationsprofil des MVSP mit der Traversierungstiefe 1, also der geringste Detailstufe. Es wurde mit der Simulation von 2.500 Szenarien in 56,54 Sekunden begonnen, anschließend wurden 5.000 Szenarien in 204,52 Sekunden, 10.000 in 869,73 Sekunden simuliert und abschließend 25.000 in 5443,84 Sekunden simuliert, siehe 9.11. Der Anstieg der Laufzeit ist nicht linear, dies liegt an den Recalculation-Events, welche zur Berechnung der Endzeitpunkt der Funktionen eines Services genutzt werden. Je mehr Funktionen auf einem Service ausgeführt werden, desto mehr Events werden für die Berechnung der Endzeitpunkte erzeugt, siehe Abbildung 3.1. Dieser Zusammenhang ist exponentiell und somit ist dies auch die Laufzeit in Abhängigkeit von der Anzahl an Szenarien in der Verteilung. Die Performance genügt dennoch den Ansprüchen, eine Verbesserung der Implementierung würde die Simulation mit größerer Traversierungstiefe ermöglichen.

Die Visualisierung wurde in Kapitel 8.6 näher erläutert und erfüllt alle Anforderungen.

Mit einem Entwickler des DVZ wurde die Nutzung aller Tools einmal durchlaufen. Das initiale Einrichten des Aspektes und die Installation aller Abhängigkeiten benötigte circa 60 Minuten, die Messdurchläufe wurden einzeln durchgeführt und benötigten jeweils circa 20 Minuten. Somit sind auch die Anforderungen an die Benutzbarkeit erfüllt.

9.6 Auswertung

In diesem Kapitel wurden drei Tests durchgeführt, aus den Ergebnissen dieser können folgende Erkenntnisse abgeleitet werden.

Begonnen wurde mit einem Test der Simulation mittels Queueing Theory. Hierbei wurde erfolgreich getestet, ob die Simulation sich entsprechend dieser Regeln verhält. In diesem Test wurde auch gezeigt, dass die maximale Nutzeranzahl direkt am Plot der verarbeiteten Funktionen abgelesen werden kann. Diese Validierung wurde mit der CPU als begrenzender Ressource durchgeführt.

Für diese Thesis wurden ein Profiler und die Simulation entwickelt. Die Simulation wurde mit dem vorigen Test erfolgreich validiert, aber noch nicht verifiziert. Eine gleichzeitige Validierung und Verifizierung von Profiler und Simulation wäre durch das Reproduzieren einer Messung möglich. Die vollständige Verifizierung beider Komponenten scheiterte an der Messung der Ressourcenauslastung. Um nur Last durch den ausgeführten Service zu berücksichtigen, wurden alle Java Prozesse beachtet. Leider führte dieses prozessbasierte Auslesen der Auslastung dazu, dass die Frequenz von 10 Messungen pro Sekunde nicht eingehalten werden konnte.

Um die Simulation dennoch Verifizieren zu können, wurde eine einfache verteilte Beispielapplikation entwickelt und für die Simulation nachgebaut. Sowohl die maximale Nutzeranzahl als auch die Wartezeit konnte akkurat simuliert werden, mit einer Abweichung der Wartezeit von weniger als 2%, für diese CPU-begrenzte Applikation.

Die Laufzeitgeschwindigkeit erwies sich als ausreichend, um den Anforderungen zu entsprechen. Aufgrund der ineffizienten Event-Verwaltung wächst die Laufzeit aber schneller als linear in Abhängigkeit von der Anzahl der Funktionen in der Szenarioverteilung. Höhere Detailstufen sind somit schwer zu simulieren.

Auf eine vollständige Validierung aller Aspekte der Simulation und des Profilers wurde aus Zeitgründen verzichtet. Die durchgeführten Tests sind aber vielversprechend und eine Weiterentwicklung scheint daher sinnvoll. Es konnten alle Anforderungen des DVZ erfüllt werden.

10 Fazit und Ausblick

In dieser Thesis wurden Erkenntnisse vorangegangener Arbeiten zu Reverse Engineering mittels AOP, Queueing Theory und DES kombiniert, um verteilte Applikation nicht nur zu simulieren, sondern auch automatisch nachzubilden. Hierzu wurden programmiersprachenunabhängige Konzepte und generische Implementierungen für das automatische Profiling und eine Simulation entwickelt, die das einfache Modellieren von komplexen Applikationen ermöglicht.

Nach einer Zusammenfassung der Grundlagen und dem Stand der Forschung wurden die Anforderungen des DVZ durch ein Experteninterview gesammelt. Basierend auf diesen Grundlagen, dem Stand der Forschung und den Anforderungen wurde ein Grobkonzept erstellt und mit der Technologieauswahl für die Modellierung und Implementierung begonnen. Im Feinkonzept wurde auf dem Grobkonzept aufgebaut und eine Reihe von Konzepten und Komponenten beschrieben. Die Implementierung dieser wurde in späteren Kapiteln genau beleuchtet. Abschließend wurden die Konzepte validiert und die Implementierung verifiziert.

Die Verifizierung zeigte, dass die Simulation die Wartezeit von CPU-begrenzten Applikationen mit einer Abweichung von weniger als zwei Prozent simulieren kann. Somit ist die Anforderung an die Genauigkeit erfüllt und mit einer Simulationsgeschwindigkeit von 25.000 Nutzern in circa 90 Minuten können auch die Anforderungen an die Laufzeit Performance erfüllt werden. Weitere Tests sind denkbar. Der Prozess der Datensammlung und Konvertierung in ein Format, welches die Simulation nutzen kann, erfolgte in wenigen Minuten und war somit deutlich schneller als die maximal 60 Minuten, die in den Anforderungen festgehalten wurden.

In dieser Thesis wurden die Möglichkeiten des entwickelten Ansatzes nur ansatzweise ausgeschöpft. In der Zukunft könnte die automatische Skalierung vervollständigt werden, wodurch eine Untersuchung der Service Elastizität möglich wird. Ebenso könnte die bestehende Implementierung verbessert werden, die Laufzeit Performance könnte hierdurch erheblich verbessert werden. Eine Erweiterung des Profilers auf verteilte Applikationen ist ebenfalls denkbar.

Literaturverzeichnis

[1] *The State of Modern App Delivery 2020: Results from Our Annual User Survey.* https://www.nginx.com/blog/state-of-modern-app-delivery-2020-results-annual-user-survey/, Abruf: 28. März 2021

[2] *The State of Microservices.* https://www.redhat.com/en/blog/state-microservices, Abruf: 28. März 2021

[3] *Onlinezugangsgesetz (OZG).* https://www.bmi.bund.de/DE/themen/moderne-verwaltung/verwaltungsmodernisierung/onlinezugangsgesetz/onlinezugangsgesetz-artikel.html, Abruf: 5. Mai 2021

[4] Matz, Patrice: Bachelorarbeit Nutzung von Self-Contained Systems in der elektronischen Antragstellung des MV-Serviceportals. (2020), 02

[5] *Computer Simulations in Science.* https://plato.stanford.edu/entries/simulations-science/, Abruf: 15. Juli 2021

[6] *Folding@Home Reaches Exascale: 1,500,000,000,000,000,000 Operations Per Second for COVID-19.* https://www.anandtech.com/show/15661/folding-at-home-reaches-exascale-1000000000000000000-operations-per-second-for- Abruf: 15. Juli 2021

[7] *IV. Introduction to Modeling and Simulation Systems.* https://uh.edu/~lcr3600/simulation/historical.html#:~:text=The%20history%20of%20computer%20simulation,was%20too%20complicated%20for%20analysis., Abruf: 18. Juli 2021

[8] Ullrich, Oliver ; Lückerath, Daniel: An Introduction to Discrete-Event Modeling and Simulation. In: *Simulation Notes Europe* 27 (2017), 06. http://dx.doi.org/10.11128/sne.27.on.10362. – DOI 10.11128/sne.27.on.10362

[9] Ball, Thomas ; Larus, James: Optimally Profiling and Tracing Programs. In: *ACM Transactions on Programming Languages and Systems* 16 (2000), 04. http://dx.doi.org/10.1145/143165.143180. – DOI 10.1145/143165.143180

[10] *The Fast Guide to Application Profiling.* https://www.red-gate.com/simple-talk/development/dotnet-development/the-fast-guide-to-application-profiling/, Abruf: 12. Juli 2021

[11] Mytkowicz, Todd ; Diwan, Amer ; Hauswirth, Matthias ; Sweeney, Peter: Evaluating the Accuracy of Java Profilers, 2010, S. 187–197

[12] Reiss, Steven: Event-based performance analysis, 2003. – ISBN 0–7695–1883–4, S. 74– 83

[13] Boehme, Marcel ; Cadar, Cristian ; Roychoudhury, Abhik: Fuzzing: Challenges and Reflections. In: *IEEE Software* PP (2020), 08. http://dx.doi.org/10.1109/MS.2020.3016773. – DOI 10.1109/MS.2020.3016773

[14] Engblom, Jakob ; Ermedahl, Andreas ; Sjödin, Mikael ; Gustafsson, Jan ; Hansson, Hans: Worst-case execution-time analysis for embedded real-time systems. In: *STTT* 4 (2003), 08, S. 437–455. http://dx.doi.org/10.1007/s100090100054. – DOI 10.1007/s100090100054

[15] *Aspect-oriented programming.* https://en.wikipedia.org/wiki/Aspect-oriented_programming, Abruf: 18. Juli 2021

[16] *Chapter 6. Aspect Oriented Programming with Spring.* https://docs.spring.io/spring-framework/docs/2.5.x/reference/aop.html, Abruf: 18. Juli 2021

[17] *Documentation.* https://www.eclipse.org/aspectj/docs.php, Abruf: 30. Juli 2021

[18] *QUEUING THEORY.* https://www.whitman.edu/documents/Academics/Mathematics/berryrm.pdf, Abruf: 18. Juli 2021

[19] *QUEUEING THEORY AND MODELING.* https://www0.gsb.columbia.edu/mygsb/faculty/research/pubfiles/5474/queueing%20theory%20and%20modeling.pdf, Abruf: 18. Juli 2021

[20] Members, MONARC: Models of Networked Analysis at Regional Centres for LHC Experiments (MONARC) PHASE 2 REPORT. (2000), 03

[21] *Chapter 8, Queueing Models FU Berlin Prof. Dr. Mesut Güneş.* https://www.mi.fu-berlin.de/inf/groups/ag-tech/intern/19540-V-Simulation/08_Queueing_Models.pdf, Abruf: 18. Juli 2021

[22] *Models of Networked Analysis at Regional Centres for LHC Experiments Project Execution Plan*. http://www0.mi.infn.it/~perini/monarc_pep/sld025.htm, Abruf: 28. März 2021

[23] Alam, Ferdous ; Mohan, Srimathy ; Fowler, J.W. ; Gopalakrishnan, Mohan: A discrete event simulation tool for performance management of web-based application systems. In: *Journal of Simulation* 6 (2012), 02. http://dx.doi.org/10.1057/jos.2011.8. – DOI 10.1057/jos.2011.8

[24] Buyya, Rajkumar ; Ranjan, R. ; Calheiros, Rodrigo: Modeling and Simulation of Scalable Cloud Computing Environments and the CloudSim Toolkit: Challenges and Opportunities, 2009, S. 1 – 11

[25] Dobre, Ciprian ; Pop, Florin ; Cristea, Valentin: A Simulation Framework for Dependable Distributed Systems, 2008. – ISBN 978–0–7695–3375–9, S. 181–187

[26] Dobre, Ciprian ; Cristea, Valentin: A Simulation Model for Large Scale Distributed Systems, 2007. – ISBN 978–1–4244–1841–1, S. 526 – 530

[27] Dragos Andrei, 354 C.: MONARC SIMULATION OF DISTRIBUTED SYSTEMS. TECHNIQUES OF PERFORMANCE IMPROVEMENT. TEST CASES. (2007)

[28] Legrand, Iosif ; Newman, Harvey: The MONARC toolset for simulating large network-distributed processing systems, 2000

[29] *Validation of the MONARC Simulation Tools*. https://monarc.web.cern.ch/MONARC/, Abruf: 30. Juli 2021

[30] Gschwind, Thomas ; Oberleitner, Johann: Improving Dynamic Data Analysis with Aspect-Oriented Programming. (2003), 01

[31] *Stack trace*. https://en.wikipedia.org/wiki/Stack_trace#:~:text=In%20computing%2C%20a%20stack%20trace,the%20stack%20and%20the%20heap., Abruf: 30. Juli 2021

[32] Briand, L.C. ; Labiche, Y. ; Leduc, J.: Tracing distributed systems executions using AspectJ. In: *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, S. 81–90

[33] Weingaertner, Elias ; Lehn, H. ; Wehrle, Klaus: A Performance Comparison of Recent Network Simulators, 2009, S. 1 – 5

[34] *Mesa: Agent-based modeling in Python 3+.* https://github.com/projectmesa/mesa, Abruf: 13. Juni 2021

[35] Mehlhase, Alexandra: A Python framework to create and simulate models with variable structure in common simulation environments. In: *Mathematical and Computer Modelling of Dynamical Systems* 20 (2014), Nr. 6, S. 566–583. http://dx.doi.org/10.1080/13873954.2013.861854. – DOI 10.1080/13873954.2013.861854

[36] *SimPy.* https://en.wikipedia.org/wiki/SimPy, Abruf: 13. Juni 2021

[37] *SimPy.* https://simpy.readthedocs.io/en/latest/, Abruf: 13. Juni 2021

[38] *Student's t-distribution.* https://en.wikipedia.org/wiki/Student%27s_t-distribution, Abruf: 30. Juli 2021

[39] *CPU design : answers to frequently asked questions.* https://archive.org/details/cpudesignanswers00thim_0/page/68/mode/2up, Abruf: 12. Juli 2021

[40] *Dreiecksverteilung.* https://de.wikipedia.org/wiki/Dreiecksverteilung#:~:text=Die%20Dreiecksverteilung%20(oder%20Simpsonverteilung%2C%20nach,Wahrscheinlichkeitstheorie%20und%20Statistik%20verwendet%20wird., Abruf: 30. Juli 2021

[41] *Simulation simple models and comparisonwith queueing theory.* https://monarc.web.cern.ch/docs/monarc_docs/1999-08.pdf, Abruf: 30. Juli 2021

[42] Ullrich, Oliver ; Lückerath, Daniel: An Introduction to Discrete-Event Modeling and Simulation. In: *Simulation Notes Europe* 27 (2017), 06. http://dx.doi.org/10.11128/sne.27.on.10362. – DOI 10.11128/sne.27.on.10362

Danksagung

An dieser Stelle möchte ich mich bei Allen bedanken, die einen Anteil an dieser Thesis hatten und mich während der Anfertigung unterstützten.

Als erstes möchte ich mich bei meinen Gutachtern Prof. Dr. Kresueler und Prof. Dr. Pawletta bedanken, die sich bereit erklärten meine Thesis zu begleiten.

Über die letzten 3,5 Jahre hatte ich die Freude neben meinem Studium für des DVZ-MV im Sachgebiet E-Government Entwicklung an verschiedenen interessanten Themen im Rahmen meines Praktikumsberichtes, meiner Bachelorthesis und nun auch meiner Masterthesis zu arbeiten.

Ein besonderer Dank gilt dem Sachgebietsleiter Herr Uwe Gärtitz für seine jahrelange Unterstützung und Herr Oliver Roggelin, der mir als betrieblicher Betreuer unterstützte und auf dem richtigen Weg hielt.

Außerdem möchte ich mich bei Herr Marvin Lehmann für das Korrekturlesen und das Feedback zu meiner Thesis bedanken.

Patrice Matz

Abbildungsverzeichnis

3.1 MONARC: Ressourcenallozierung [28] 21
3.2 ARE Architektur [30] 22

5.1 Flowchart Komponenten und Datenflüsse (vereinfacht) 25

6.1 Profiling und Simulation des Call-Tree 31
6.2 Komponenten und Datenflüsse 32
6.3 Kardinalitäten der Objekte 33
6.4 Komposition eines Szenarios 34

7.1 Relevante Daten und Komponenten für die Erstellung eines Applikationprofils 39
7.2 Rekonstruktion des Call-Trees 43
7.3 Visualisierung des Call-Trees 46

8.1 Input Vorbereitung 51
8.2 Komponenten und Datenflüsse (vollständig) 52
8.3 Simulation-Engine 58
8.4 Visualisierungs-Dashboard 61
8.5 Visualisierungsloop der Simulation 64

9.1 Validierung Queueing Theory: Queueing Network mit einer Queue . . 66
9.2 Validierung Queueing Theory: Queueing Network mit drei Queues . . 66
9.3 Gantt-Diagramm aller Funktionen 67
9.4 Rekonstruierter Call-Tree 68
9.5 Gemessene Service Auslastung 69
9.6 Simulierte Service Auslastung 69
9.7 Gantt-Diagramm Interaktionen 70
9.8 Validierung Versuchsaufbau 3 71
9.9 Vergleich simulierter Antwortzeit (oben) mit gemessener Antwortzeit (unten) 72
9.10 Vergleich simulierter CPU-Auslastung (oben) mit gemessener CPU-Auslastung (unten) 72
9.11 Laufzeitgeschwindigkeit der Simulation über Volumen der Szenarienverteilung 73

Code Listings

7.1 Call Interceptor Aspekt 40
7.2 Funktion zum Aufbau des Call-Trees 44
7.3 Interaktiongeneration 48

8.1 Beispiel Service Definition 53
8.2 Event Implementierung 55
8.3 Monitoring Queue Eintrag 59
8.4 Event Implementierung 63
8.5 Beispiel Plot mit Pandas und Dash 64

A.1 Beispiel CallLog für Validierung 86
A.2 JSON Schema für ein Applikation Profil 86
A.3 Berechnung des Gewichtungsfaktors der Ressourcenauslastung einer von mehrere parallelen Funktionen 88

A Anhang

A.1 Experteninterview

F Das MVSP ist ja eine Plattform mit wachsender Nutzeranzahl. Wie viele aktive Nutzer hat die Plattform an einem durchschnittlichen Tag momentan? Was ist die absehbare maximale Nutzeranzahl in 5 Jahren?

A Das ist Tagesabhängig, am Wochenende haben wir weniger Nutzer als an einem Wochentag. Man muss auch zwischen anonymen Nutzern, Bots und registrierten Nutzern unterscheiden. Tatsächliche individuelle Nutzer haben wir ca. 25 - 50 pro Tag. Momentan befindet sich das MVSP aber auch in einem produktiven Beta-Betrieb. Die vollen 500 Leistungen werden ja erst ab 2022 angeboten. Ab dann erwarten wir auch deutlich mehr, eher 10 – 25 Tausend Nutzer pro Tag, besondere Events ausgenommen. Bei einer Million Einwohner, die MV hat sind 10 – 25 Tausend vielleicht noch konservativ geschätzt.

F Kann das MVSP mit seiner jetzigen Architektur diese Nutzermengen bedienen?

A Nein. Eine vertikale Skalierung reicht für solche Nutzermengen nicht aus. Die digitale Infrastruktur wurde darauf nicht getestet und ist dem vermutlich nicht gewachsen.

F Welche Herausforderungen gibt es bei der horizontalen Skalierung?

A Ein Replizieren des gesamten Deployments wäre ineffizient. Die Infrastruktur für das Monitoring wäre ein Single-Point-of-Failure. Komponenten wie das Nutzerkonto lassen sich nicht skalieren. Auch Lizensierungen stellen eine Herausforderung beim horizontalen Skalieren da.

F Das MVSP steht ja nicht allein, wie viele Abhängigkeiten hat die Plattform circa?

A Wir haben 7 – 10 direkte Abhängigkeiten, von denen ca. 5 in jedem Antrag aufgerufen werden und diverse weitere für einzelne Verfahren. Theoretisch wäre bis zu einer Abhängigkeit pro Antrag denkbar, das wird schnell unüberschaubar, besonders wenn einzelne Kommunen besondere Wünsche haben.

F Können diese Abhängigkeiten die Skalierung begrenzt?

A Ja definitiv.

F Ist diese Grenze realistisch abschätzbar?

A Das MVSP als Plattform ist mittlerweile so groß, dass nur wenige Entwickler den vollen Überblick besitzen. In der Zukunft wird das System nur komplexer. Dort Grenzen abzuschätzen ist kaum möglich.

F Wenn ja, wie wäre das Vorgehen? Ist dieses Vorgehen praktikabel?

A Nein, das wird nur komplexer. Mit unserer Rule-Engine die von nötigen Capabilities eines Antrages weiß wäre das in der Zukunft vielleicht möglich, aber nicht momentan.

F Was wären Ihre Anforderungen an die Performance einer Simulation?

A Es sollte ein Tag mit der erwarteten Nutzerzahl mindestens in Echtzeit simulierbar sein.Das wären 3 – 4 Nutzer pro Minute / 200 – 300 pro Stunde, das wäre ausreichend. Die Laufzeit ist eher weniger relevant.

F Eine Simulation ist immer nur eine Annäherung an ein reales System, es wird daher zu Abweichungen kommen, wie groß sollten diese maximal sein, um die Abschätzung unterstützen zu können?

A Abweichung +- 30%, für maximale Nutzer pro Zeiteinheit, wären vollkommen akzeptabel, es geht mehr um die Richtung, also die Größenordnung und die Identifizierung der Schwachstelle, als den exakten Wert.

F Wie würden Sie sich die ideale Anwendung der Datenerhebung und Simulation vorstellen?

A Initial Aufwand darf vorhanden sein, pro Messdurchlauf sollten nicht mehr als 60min an vor und Nachbereitung nötig sein. Das „Druchklicken“ ausgenommen

F Welche Metriken sollten auslesbar sein?

A Anzahl der Nutzer, Auslastung der Server (CPU, RAM), (http-)Aufrufe die ein Nutzer in einem Szenario verursacht, Antwortzeit der Funktionen, Filterung nach Szenario sollte möglich sein

```
    startRoot i1 5000000000
    start 1 5800000000
    start x 6000000001
    end x 6250000000
    start x2 5900000001
    end x2 6250000000
    end 1 7000000000
    start 2 8000000000
    end 2 22000000000
    endRoot i1 25000000000
    startRoot i2 40000000000
    start 4 45000000000
    end 4 80000000000
    endRoot i2 90000000000
```

Code Listing A.1: Beispiel CallLog für Validierung

```
{
"$schema": "https://json-schema.org/draft/2020-12/schema",
"$id": "/Matz/Patrice/Master-Thesis/Profile.schema.json",
"title": "Profile",
"description": "An Applications measure Profile in different Scenarios",
"type": "object",
"properties": {
    "name": {
        "description": "user defined name, optional",
        "type": "string"
    },
    "scenarios": {
        "description": "",
        "type": "array",
        "minItems": 1,
        "uniqueItems": true,
        "items": {
            "type": "object",
            "properties": {
                "name": {
                    "description": "user defined name, optional",
                    "type": "string"
                },
                "scenarioID": {
                    "description": "The unique identifier for a scenario",
                    "type": "integer"
                },
                "interactions": {
                    "description": "",
                    "type": "array",
                    "minItems": 1,
                    "uniqueItems": true,
                    "items": {
                        "type": "object",
                        "properties": {
                            "name": {
                                "description": "user defined name, optional",
                                "type": "string"
```

```
        },
        "interactionID": {
            "description": "The unique identifier for a
                interaction",
            "type": "string"
        },
        "delay": {
            "description": "Time between two interactions in
                seconds, cause by human interaction",
            "type": "integer"
        },
        "functions": {
            "description": "array of functions with measured
                ressource utilization",
            "type": "array",
            "minItems": 1,
            "uniqueItems": true,
            "items": {
                "type": "object",
                "properties": {
                    "functionID": {
                        "description": "The unique identifier
                            for a function",
                        "type": "string"
                    },
                    "cpu": {
                        "description": "cpu utilization in
                            percent",
                        "type": "number"
                    },
                    "cpu_time": {
                        "description": "amount of time function
                             takes to execute in seconds",
                        "type": "number"
                    },
                    "ram": {
                        "description": "amount of RAM used
                            while executign the function",
                        "type": "number"
                    },
                    "io": {
                        "description": "amount of data read
                            from Disk",
                        "type": "number"
                    },
                    "net": {
                        "description": "amount of data gotten
                            over the network",
                        "type": "number"
                    },
                    "delay": {
                        "description": "delay in nano seconds",
                        "type": "integer"
                    },
                    "callbacks": {
                        "description": "The ID of the Callback
```

```
                                        Function. -1 means no callback",
                                    "type": "array",
                                    "minItems": 1,
                                    "uniqueItems": true,
                                    "items":{
                                        "type": "string"
                                    }
                                }
                            },
                            "required": [
                                "functionID"
                            ]
                        }
                    }
                },
                "required": [
                    "interactionID",
                    "functions",
                    "delay"
                ]
            }
        }
    },
    "required": [
        "scenarioID",
        "interactions"
    ]
  }
}
}
```

Code Listing A.2: JSON Schema für ein Applikation Profil

```
def getFunctionUtilFactor(function, functions):
    if not function.isAsync:
        return 1

    relevantFuncs = [function]
    for f in functions:
        if f == function:
            continue
        if not f.isAsync:
            continue
        if not function.overlaps(f):
            continue

        relevantFuncs.append(f)

    relevantFuncs.sort(key=lambda x: x.start)

    vals = {}

    for f in relevantFuncs:
        if f.start < function.start:
            vals[function.start] = "inc"
```

```
23          else:
                vals[f.start] = "inc"
25
            if f.end > function.end:
27              vals[function.end] = "dec"
            else:
29              vals[f.end] = "dec"

31      val2 = {}
        val2[function.start] = 0
33
        counter = 0
35      for key in sorted(vals.keys()):
            val = vals[key]
37          if val == "inc":
                counter += 1
39          if val == "dec":
                counter -= 1
41          val2[key] = counter
        val2[function.end] = 0
43
        keys = sorted(val2.keys())
45      functionLength = function.end - function.start

47      factor = 0

49      for i in range(len(keys)-1):
            duration = keys[i+1] - keys[i]
51          count = val2[keys[i]]
            factor += (duration / functionLength) * ( 1 / count )
53
        return factor
```

Code Listing A.3: Berechnung des Gewichtungsfaktors der Ressourcenauslastung einer von mehrere parallelen Funktionen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig verfasst und nur unter Verwendung der aufgeführten Hilfsmittel angefertigt habe. Die Stellen der Arbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.

Ich erkläre ferner, dass ich die vorliegende Arbeit in keinem anderen Prüfungsverfahren als Prüfungsarbeit eingereicht habe oder einreichen werde.

Die eingereichte schriftliche Fassung entspricht der auf dem Medium gespeicherten Fassung.

Ort, Datum Unterschrift