HAW HAMBURG

BACHELOR THESIS
Julian Friedemann

Optimierung von automatischen Skalierungs- und Lastverteilungsmechanismen am Beispiel einer ungeeigneten Applikation

FAKULTÄT TECHNIK UND INFORMATIK
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Julian Friedemann

Optimierung von automatischen Skalierungs- und Lastverteilungsmechanismen am Beispiel einer ungeeigneten Applikation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Marina Tropmann-Frick

Eingereicht am: 30. November 2022

Julian Friedemann

Thema der Arbeit

Optimierung von automatischen Skalierungs- und Lastverteilungsmechanismen am Beispiel einer ungeeigneten Applikation

Stichworte

Skalierung, Cloud-Systeme, Cloud-Anwendung, Lastverteilung, Warteschlangentheorie

Kurzzusammenfassung

Energiekrise, Klimawandel und Kosteneffizienz spielen eine immer größer werdende Rolle in unserer Zeit [34]. Auch im Kontext von Cloud-Systemen und Anwendungen in der Cloud gilt es, effizient und umweltschonend zu agieren. Um Energie- und Kosteneffizienz zu erreichen bieten Lastverteilungs- ung Skalierungsmechanismen eine Lösung [8]. In dieser Arbeit werden Vorgehensweisen für die Skalierung und Lastverteilung eines Cloud-Systems analysiert. Eine Grundlage für die Analyse bilden Konzepte aus der Warteschlangentheorie. Ziel dieser Arbeit ist es, Optimierungsmöglichkeiten für eine Anwendung zu erörtern, welche nicht für den hochverfügbaren Einsatz konzipiert wurde. Eine ungeeignete Anwendung stellt in dieser Arbeit eine Applikation dar, welche hohe Antwortzeiten hat und keine gleichzeitigen Nutzer:innen ermöglicht. Diese Anwendung wird als Black Box angesehen - ein System welches nur von außen betrachtet werden kann. Anhand der Grundlagen der Warteschlangentheorie wird das Warteschlangen-Modell von Vilaplana et al. besprochen. Die Ergebnisse von Vilaplana et al. beschreiben, wie ein System aus Wartschlangen agiert. Durch Anpassung des Modells auf den Anwendungsfall dieser Arbeit kann festgestellt werden, dass eine hohe Serveranzahl nicht zwingend eine geringere Antwortzeit zur Folge hat. In diesem Fall kann die Servicezeit so hoch sein, dass die Ankunftsrate keine effektive Nutzung der übrigen Server ermöglicht. Hierbei entstehen Leerläufe, die es zu vermeiden gilt. In den Versuchen dieser Arbeit werden Lastverteilung und Skalierung genutzt, um diese Leerläufe zu minimiern. Deutlich wird, dass der im theoretischen Teil der Arbeit besprochene Lastverteilungsalgorithmus Advanced weighted Round Robin (AWRR) 2.4.6, die effizienteste Lastverteilung ermöglicht. Durch die Kombination des AWRR und Skalierung entsteht ein System, welches eine ungeeignete Anwendung hochverfügbar machen kann.

Julian Friedemann

Title of Thesis

Optimization of automatic scaling and load balancing mechanisms using the example of an unsuitable application

Keywords

Scaling, cloud systems, cloud application, load balancing, queueing theory

Abstract

Energy crisis, climate change and cost efficiency play an ever increasing role in our time [34]. In the context of cloud systems and applications in the cloud, it is also important to act efficiently and in an environmentally friendly manner. To achieve energy and cost efficiency, load balancing and scaling mechanisms provide a solution [8]. In this thesis, approaches for scaling and load balancing of a cloud system are analyzed. Concepts from queueing theory form a basis for the analysis. The goal of this work is to discuss optimization options for an application that was not designed for high-availability use. An unsuitable application in this work is an application that has high response times and does not allow concurrent users. This application is considered a black box - a system that can only be viewed from the outside. Based on the basics of queuing theory, the queuing model of Vilaplana et al. is discussed. The results of Vilaplana et al. describe how a system of queues acts. By adapting the model to the use case of this thesis, it can be found that a high number of servers does not necessarily result in a lower response time. In this case, the service time may be so high that the arrival rate does not allow effective use of the remaining servers. This results in idle time, which must be avoided. In the experiments of this thesis, load balancing and scaling are used to minimize these idle times. It becomes clear that the Advanced weighted Round Robin (AWRR) load balancing algorithm 2.4.6, discussed in the theoretical part of the thesis, provides the most efficient load balancing. The combination of the AWRR and scaling creates a system that can make an unfit application highly available.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Durch verschiedenste äußerliche Einflüsse wird es in der heutigen Zeit immer relevanter, Serveranwendungen effizient zu skalieren. Denn sowohl die Systemleistung - im Sinne der Verfügbarkeit - als auch die Kosten, um die Server zu betreiben, spielen eine größere Rolle als je zuvor [8].

Im Rahmen des dualen Studiums arbeitete ich an einem Projekt zur effizienten Bereitstellung einer Datenvisualisierungsanwendung. Hierbei sollte die Serverumgebung für eine ¨single threaded¨ Anwendung realisiert werden, sodass mehrere Nutzer:innen die Anwendung ohne Leistungverluste oder längere Wartezeiten gleichzeitig verwenden können.

In dieser Arbeit wird die Thematik der Skalierung und Lastverteilung um eine Serveranwendung, deren Laufzeit nicht für den hochverfügbaren Einsatz geeignet ist, analysiert. Hierbei soll auf die verschiedenen Techniken von Lastverteilungsalgorithmen und Skalierungsmechanismen eingegangen werden, sowie praktische Anwendung finden.

Ziel dieser Arbeit ist daher zu veranschaulichen, welche Techniken und Algorithmen verwendet werden können, um Skalierung und Lastverteilung zu optimieren und Anwendungen für den hochverfügbaren Einsatz zu realisieren.

Zu Beginn werden die grundlegenden Techniken und Systeme erläutert, die in dieser Arbeit Verwendung finden. Weiterführend werden die theoretischen Konzepte analysiert und daraus erste Aussagen formuliert, wie Lastverteilung und Skalierung Optimierung ermöglichen. Ebenso werden zur Bewertung der Lastverteilungs- und Skalierungsmechanismen aus dem Bereich der Warteschlangentheorie das System der M/M/1 und M/M/m Warteschlangen näher betrachtet. Hierzu wird ein Modell eines Netzwerks aus Warteschlangen analysiert und an diese Arbeit angepasst. Die theoretischen Resultate werden im folgenden Kapitel in Versuchen unter Verwendung der vorgestellten Algorithmen veranschaulicht. Die daraus entstehenden Ergebnisse sowie die theoretischen Konzepte sollen die Frage beantworten, wie eine Applikation - die nicht für den hochverfügbaren Einsatz gedacht ist - optimiert werden kann.

2 Theorie

2.1 Motivation

Die Wirtschaft entwickelt sich zu einer digitalen Informationverwaltung. Serverlandschaften, die Schlüsselpunkte unserer derzeitigen Datenverarbeitung und –verwaltung sowie Kommunikationsnetzwerke darstellen, sind in fast jedem Wirtschaftszweig zu finden und erfahren derzeit eine steigende Anfrage. Daraus ergibt sich ein Anstieg des Energieverbrauchs [32]. Ein steigender Energieverbrauch führt zu einem Anstieg der Betriebskosten und der Treibhausgasemissionen [34]. Durch Server im Leerlauf entstehen so unnötige Kosten und Emissionen. Im Idealfall sollte die Kapazität der Serverfarm daher dynamisch angepasst werden - auf der Grundlage der eingehenden Nachfrage [22]. Nicht nur ökologische Faktor tragen zur Relevanz des Themas bei. Laut Rina A. Doherty [18] wird für ein:e Nutzer:in die Relevanz durch die UX (User Experience) widergespiegelt. Wenn ein:e Nutzer:in sich im Flow (Fluss) einer Anwendung befindet, richtet sich der Fokus auf die Arbeit der Person. Die Anwendung wird ausgeblendet. Dies führt zur positiven Wahrnehmung der Anwendung und deren Leistung. Entsteht eine Unterbrechung dieses Flusses (durch Ladezeit, lange Anwortzeiten), führt dieser Umstand dazu, dass die Aufmerksamkeit des/der Nutzer:in auf die Anwendung gelenkt wird. Dadurch entsteht eine negative Wahrnehmung der Anwendung. Das Konzept des „Flow“ ist ein wesentlicher Faktor für die Zufriedenheit, insbesondere bei hoch interaktiven Geräten oder Anwendungen [18].

Aus der Sicht des Auftraggebenden ist eine Optimierung der Kosten von Relevanz [31]. Anbieter von Cloud Computing-Diensten (z.B. Amazon EC2) stellen Infrastruktur auf Anfrage zur Verfügung und berechnen diese nach Verbrauch. Auf diese Weise wird, genau wie beim Verbrauch von anderen Versorgungsunternehmen (z. B. Wasser, Strom oder Gas), eine Rechnung erstellt. Es muss das bezahlt werden, was verbraucht wurde. Weniger Nutzungszeiten resultieren in weniger Kosten [31].

Wie bereits einleitend geschildert ist aus ökologischer Sicht die Optimierung der Laufzeit und Serverzeiten von Relevanz. Der Anstieg von Serverzeiten führt zu einem Anstieg der Emission von Treibhausgasen [34]. Wie Anwendungen möglichst optimal auf eine wechselnde Nachfrage/Anzahl an Nutzenden regieren können, ist somit von hoher Relevanz. Aus diesem Grund widmet sich die vorliegende Arbeit der Erforschung von Lastverteilung und Skalierung, von Anwendungen für den hochverfügbaren Einsatz zu realisieren.

In den folgenden Abschnitten werden die Techniken eingeführt, die in dieser Arbeit verwendet oder erwähnt werden.

2.2 Allgemeine Einführung

2.2.1 DevOps

Vereinfacht beschreibt der Begriff der DevOps die Kombination aus Entwicklung und Betrieb von Software – meist Cloud Systemen. DevOps ist eine neue Denkweise im Bereich der Softwareentwicklung, die in letzter Zeit viel Aufmerksamkeit erhalten hat [28]. Ein Großteil dieser Arbeit bewegt sich im Kontext der DevOps.

2.2.2 Container und Docker

Die in dieser Arbeit untersuchte Anwendung (2.3) wird in einem Docker-Container realisiert.

Docker ist ein Open-Source-Projekt, das auf vielen längst bekannten Technologien aus der Betriebssystemforschung aufbaut wie zum Beispiel LXC-Container (eine leichtgewichtige Virtualisierungstechnologie), Virtualisierung des Betriebssystems und ein hash-basiertes oder git-ähnliches Versionierungs- und Differenzierungssystem. Dabei versucht Docker – oder Container im Allgemeinen – die technischen Herausforderungen der ¨Dependency Hell¨, unpräzise Installation/Setup Dokumentation, ¨Code-Rot¨ (Veraltung von Abhängigkeiten oder Code) und die Fähigkeit zur Adaption und Wiederverwendbarkeit zu lösen [2].

Docker besteht im Wesentlichen aus vier Komponenten: dem Docker-Client und Docker-Server/Daemon, Docker-Images, der Docker-Registries sowie Docker-Containern. Docker kann als Client-Server basierte Applikation verstanden werden.

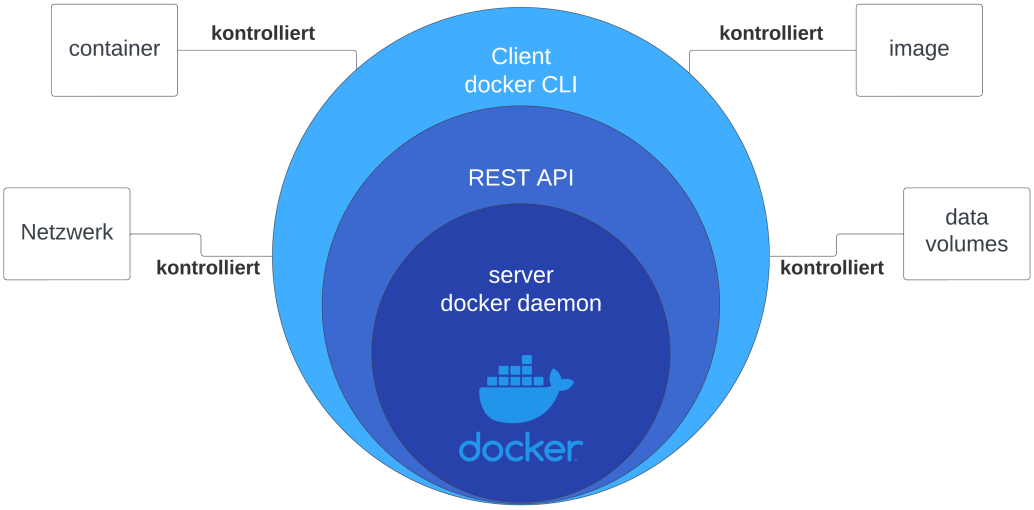


Abbildung 2.1: Beispielhafte Darstellung der Bestandteile von Docker nach [29]

Ein Docker-Client stellt dem entsprechenden Docker-Server/Daemon über RESTful API eine Anfrage und verarbeitet diese. Dieser Server/Daemon kann sowohl lokal als auch von ¨außen¨ über das Internet bei entsprechender Konfiguration erreicht werden. Docker-Images sind vereinfacht Abbilder eines kompletten Systems. Die Images können Betriebssysteme sein bis hin zu vollständigen Anwendungen. Images beinhalten in der Regel alle nötigen Anforderungen und können unabhängig von einem System ausgeführt werden. Mithilfe des ¨docker build¨-Befehls werden die Images nach den Instruktionen gegeben durch ein Dockerfile erstellt. In einer Registry werden Images gesammelt und unter anderem öffentlich zugänglich gemacht (zum Beispiel durch Docker-Hub). Über Docker-Images können Docker-Container erstellt werden. Die Container enthalten das gesamte - für eine Anwendung erforderliche - System, sodass die Anwendung isoliert ausgeführt werden kann. Wird beispielsweise ein Abbild des Ubuntu-Betriebssystems mit einem SQL-SERVER erstellt und dieses Image mit dem Befehl ¨docker run¨ ausgeführt, dann wird ein Container gestartet, welcher einen SQL-SERVER auf dem Ubuntu-Betriebssystems ausführt.

Die Container werden durch Docker verwaltet und als Methode zur Virtualisierung auf Betriebssystemebene verwendet. Lediglich ein ¨Kontrollhost¨ verwaltet die isolierten Container. Ressourcen wie Netzwerk, Speicher und CPU werden vom Host-Kernel zugewiesen [1].

2.2.3 Hypertext Transfer Protocol

Das Hypertext Transfer Protocol ist ein Anfrage/Antwort-Protokoll [20]. Ein Client sendet eine Anfrage an den Server in Form einer Anfragemethode, einer URI und einer Protokollversion, gefolgt von einer MIME-ähnlichen [21] Nachricht, die Request Modifikatoren, Client-Informationen und möglichen Body-Inhalten über eine Verbindung mit einem Server. Der Server antwortet mit einer Statuszeile, einschließlich der Protokollversion der Nachricht und eines Erfolgs- oder Fehlercodes, gefolgt von einer MIME-ähnlichen Nachricht, den Serverinformationen, Entity Metainformationen und möglichen Entity-Body-Inhalt enthält [20].

Im praktischen Teil dieser Arbeit wird durch Python-Bibliothek "Flask"verwendet. Flask verwendet standardmäßig die HTTP-Version 1.1. Die Funktionsweise dieser Version wird im folgenden, durch das RFC-2616 und RFC-9110 genauer betrachtet.

Die meisten Kommunikationsvorgänge mittels HTTP bestehen aus einer Anfrage (GET) für eine Darstellung einer durch einen URI identifizierten Ressource. Im einfachsten Fall kann dies über eine einzige bidirektionale Verbindung (===) zwischen dem User Agent (UA) und dem Origin Server (O) erfolgen [20].

Abbildung 2.2: HTTP-Kommunikation [20]

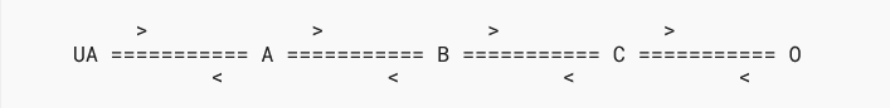


Abbildung 2.3: HTTP-Kommunikation mit Zwischenschaltungen [20]

HTTP ermöglicht den Einsatz von Vermittlern, um Anfragen über eine Kette von Verbindungen zu erfüllen. Es gibt drei gängige Formen von HTTP-"Vermittlern": Proxy, Gateway und Tunnel. In einigen Fällen kann ein einziger Vermittler als Ursprungsserver, Proxy, Gateway oder Tunnel fungieren und je nach Art der Anfrage sein Verhalten ändern [19].

Die Abbildung 2.3 zeigt eine Kommunikation zwischen dem User Agent und O über die Entitäten A, B und C. Eine Anfrage oder Antwort, die die gesamte Kette durchläuft, durchläuft vier verschiedene Verbindungen. Diese Unterscheidung ist wichtig, weil einige HTTP-Kommunikationsoptionen nur für die Verbindung mit dem nächstgelegenen, nicht getunnelten Nachbarn, oder nur für die Endpunkte der Kette oder für alle Verbindungen entlang der Kette gelten. Obwohl das Diagramm linear ist, kann jeder Teilnehmer an mehreren gleichzeitigen Kommunikationen beteiligt sein. Zum Beispiel kann B Anfragen von anderen Clients als A erhalten und/oder Anfragen an andere Server als C weiterleiten, während er gleichzeitig die Anfrage von A bearbeitet [20].

2.2.4 Reverse-Proxy/HTTP-Loadbalancer

Um Lastverteilungssystem genauer analysieren zu können, wird im Folgenden der Begriff des Lastverteilers erklärt.

Die meisten Lastverteiler (Loadbalancer) basieren auf dem Prinzip eines Reverse-Proxy-Servers. Hierbei werden verschiedene Peers auf Basis definierter Algorithmen ausgewählt, um die eingegangenen Anfragen zu bearbeiten [38]. Der zugrundeliegende Reverse-Proxy-Server ist in der Regel eine Anwendung, welche als Vermittler bei der Netzwerkkommunikation zwischen Client und Ziel-Anwendung dient. Die Kommunikation wird im Kontext des Webs über HTTP-Anfragen und –Antworten realisiert.

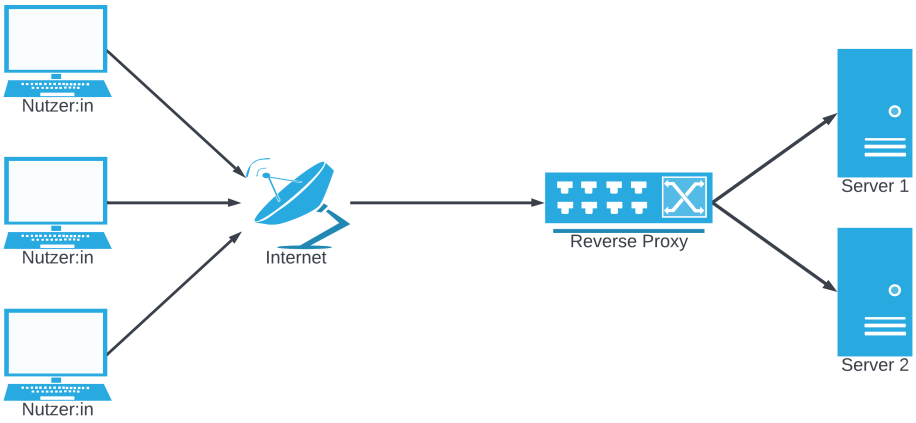


Abbildung 2.4: Vereinfachte Sequenz eines Reverse-Proxys. Eigene Darstellung orientiert an [7]

Als gängige Softwarelösungen sind Nginx und Traefik bekannt. Nginx, ein Reverse-Proxy, wird unter anderem auch von Webseiten wie Wikipedia und WordPress genutzt [12].

2.3 Ungeeignete Applikation

Eine Cloud-Applikation bei einem Cloud Computing-Dienst bereitzustellen, ist keine Garantie für hohe Leistung, Skalierbarkeit und ähnliche Qualitätsmerkmale. Die Bedeutung eines angemessenen Designs einer Cloud-Applikation ist groß und die Cloud selbst nicht die Lösung [24]. In diesem Abschnitt werden die Probleme und Herausforderungen, die mit dem Design einer Cloud-Applikation verbunden sind (Durchsatz, Reaktionszeit, etc.), erörtert und der Begriff einer ungeeigneten Applikation definiert.

Im folgenden wird die Applikation als „Black Box“ betrachtet, es wird davon ausgegangen, dass die Applikation als solches nicht verändert werden kann.

2.3.1 Antwortzeiten

Antwortzeiten oder Latenzen sind im Kontext von Cloud-Applikationen zu differenzieren zwischen Netzwerk-Latenzen und Antwortzeiten der Applikation. In diesem Fall werden die Antwortzeiten einer Applikation betrachtet und Netzwerk-Latenzen vernachlässigt. Ein Indikator für eine ungeeignete Applikation sind lange Antwortzeiten.

Die Gründe für die Verschlechterung von Antwortzeiten liegt, laut David Gesvindr [24], meist beim Entwickelnden oder schlechten Designentscheidungen. Aufwändige Berechnungen und wiederholenden Prozesse sollten vermieden werden, da hierbei lediglich die Reaktivität des Dienstes negativ beeinflusst wird. Caching ist besonders für Datenbankzugriffe ein Weg, um repetitive Anfrage an eine Datenbank zu vermeiden und schnell das gewünschte Ergebnis zu liefern. Dadurch verringert sich nicht nur die Anwortzeit sondern auch der rechentechnische Aufwand. Jedoch müssen Caching Algorithmen so implementiert werden, dass möglichst viele Information durch den Cache gespeichert werden können, aber die Diversität der Information dadurch nicht beeinträchtigt wird.[24]

Ähnlich haben im Kontext von Datenbanken synchrone Schreib-Operationen einen negativen Einfluss auf Antwortzeiten. Denn durch die grundsätzliche Funktionsweise von z.B. NoSQL-Datenbanken (ACID – Isolation) werden Schreib-Operationen erst ausgeführt,

wenn keine andere Schreib-Operation aktiv ist. Eine Lösung für dieses Problem ist, Änderungen zu sammeln und in einer Anfrage mehrere Schreib-Operationen auszuführen, anstatt für jede Änderung an einer Tabelle einen Aufruf zu versenden. Zwar resultiert diese Lösung im gleichen rechnerischen Aufwand, aber gleichzeitige Schreib-Operationen werden verringert, da die Wahrscheinlichkeit sinkt, dass eine Anfrage die Datenbank im gleichen Moment erreicht. [24]

Eine negative Beeinflussung der Antwortzeiten findet ebenfalls durch externe Service-Calls statt. Größere Anfragen gilt es zur vermeiden. Das ist der Fall, wenn beispielsweise bei jedem Aufruf eines Services eine vollständige Tabelle angefragt, aber nur ein Bruchteil der Daten benötigt wird. Hier sollten die Anfragen effizienter gestaltet werden, um nur die Daten anzufragen, welche benötigt werden. Ebenso können parallele Anfragen Antwortzeiten verschlechtern. Wenn Anfragen in einer synchronen Weise abgearbeitet werden, ergibt sich für die Antwortzeit aller Anfragen als Gesamtzeit die addierte Zeit eines jeden Aufrufes. Asynchrone Aufrufe sollten hierbei verwendet werden, wenn die Anfragen voneinander unabhängig sind [24].

2.3.2 Durchsatz

Der Durchsatz kann definiert werden als ein Maß für die Menge an Arbeit, die eine Applikation in einer Zeiteinheit leisten kann [25]. Im Folgenden soll der durchschnittliche Durchsatz einer Cloud-Applikation nach diesem Maß betrachtet werden.

Die Relevanz des Durchsatzes wird durch eine Online-Tracking Anwendung zu den 2014 US-Wahlen deutlich, welche auf einer Microsoft Azure Cloud Plattform bereitgestellt wurde. Ein weit verbreiteter Irrglaube ist, dass jede in der Cloud gehostete Anwendung, einen hohen Durchsatz besitzt. Zu Beginn der Wahlen, wurde eine Spitzenlast mit 1667 Anfragen pro Sekunde erwarten. Durch Änderungen in der Applikationsarchitektur (weniger Anfragen durch Caching-Layer bearbeitet) wurde eine Spitzenlast von 44893 Anfragen pro Sekunde gemessen. Dieser Umstand hätte zu einem mit hoher Sicherheit zu einem Ausfall der Anwendung geführt, wenn nicht eine überdimensionierte Caching Infrastruktur bereitgestellt worden wäre.

Im Kontext des Durchsatzes sollte also folgendes beachtet werden. Es sollten reale Schätzung für den Durchsatz des Systems gemacht werden. So können - durch historische Daten oder Schätzungen im Vorfeld - Aussagen getroffen werden über die Eignung der Cloud-Applikation. Analog dazu sollte Ressourcen analysiert werden, welche den Durchsatz fest

limitieren, da dieser den Gesamt-Durchsatz der Cloud-Applikation mitbestimmt. Service-Calls und Datenbank-Zugriffe sollten minimiert oder vereinheitlicht werden durch Caching oder Batch-Processing, um möglichst viele kleinere Anfragen zu vermeiden [24].

2.3.3 Skalierbarkeit und Elastizität

Die wichtigsten Leistungsmerkmale - Durchsatz und Antwortzeit - bestimmen das Applikationsverhalten für eine Cloud-Applikation. Andererseits definiert die Skalierbarkeit einer Applikation, wie sich die Applikation bei steigender Last und dynamischen Serverressourcen verhält und ob sie in der Lage ist, neu bereitgestellte Ressourcen effektiv zu nutzen. Die Elastizität der Plattform beschreibt, wie schnell und genau die Umgebung in der Lage ist, sich an Änderungen der Arbeitslast anzupassen, indem sie die Menge der bereitgestellten Ressourcen ändert [25]. Für die Skalierbarkeit gilt,

- dass Last verteilt werden sollte, sodass einzelne Ressourcen nicht überlastet werden, während andere im Leerlauf sind.
- dass die Applikation so designt sein sollte, sodass Anfragen Zustandslos bearbeitet werden können, denn es kann für eine horizontale Skalierung nicht davon ausgegangen werden, dass Anfragen vom gleichen Client gesendet werden.
- dass die Architektur und Infrastruktur der Cloud-Applikation für Skalierung offen sein sollte. So führen z.B. Hardwarelimitationen bei Änderung der Anforderungen zu Problemen oder einen kostenaufwändigen Re-Design der Architektur.

Für die Elastizität gilt:

- Horizontales skalieren: Der Prozess eine neue Instanz einer Ressource/Anwendung zu starten erfordert keinen Neustart im Gegensatz zur vertikalen Skalierung. Das System bleibt verfügbar und die neue Ressource kann eingehende Anfragen bearbeiten.
- Entscheidungen Treffen anhand von Metriken: Durch die Interpretation der Systemmetriken können Schwachpunkte des Systems identifiziert und gegebenenfalls behoben werden.
- Für eine Art der Skalierung entscheiden: Eine geplante Skalierung kann unter gewissen Bedingungen die Kosten um 30-50% reduzieren im Vergleich zur automatisierten Skalierung einiger Cloud-Anbieter.[24]

2.3.4 Verfügbarkeit

Die Verfügbarkeit eines Systems sei definiert durch einen prozentualen Zeitanteil, den eine Applikation in einem operativen Zustand ist. Viele Cloud-Anbieter versprechen Verfügbarkeiten von bis zu 99.99%, was vier Minuten Unterbrechung innerhalb eines Monats entspricht. Eine Applikation bei einem Cloud-Anbieter bereitzustellen, bedeutet im Umkehrschluss jedoch nicht 99,99% Verfügbarkeit. Die Applikation ist ausschlaggebend für die Verfügbarkeit. Grundsätzliche Konzepte, um hohe Verfügbarkeit zu erreichen bezüglich einer Cloud-Applikation und derer Bereitstellung sind laut David Gesvindr und Barbora Buhnova [24]:

- die Implementation einer ¨Retry-Logik¨, um Fehler während eines Skalierungsprozesses oder Ähnlichem aufzufangen.
- Recovery-Szenarien: Wenn Fehler auftreten sollten oder der Service nicht mehr verfügbar sein sollte, gilt es, das System, mithilfe von Daten-Backups oder einer Neustart Logik, wiederherzustellen.
- Ebenso kann die Applikation so designt werden, dass sie erkennt, wenn ein Fehlerzustand erreicht ist und entsprechende diese Funktionalitäten automatisch blockiert, um einen kompletten Ausfall zu vermeiden.

2.3.5 Qualität des Gesamtsystems

Die erörterten Faktoren einer Cloud-Applikation sind kritisch zu bewerten und das Vernachlässigen (wie von David Gesvindr [24] erläutert) kann zu erheblichen Qualitätseinbußen führen. Leistungsoptimierung von Cloud-Applikationen ist ein kontinuierlicher Prozess und sollte schon in der Entwicklungsphase berücksichtigt werden. Dabei sollten auch die Leistungsanforderungen der Applikation vollständig geplant und analysiert werden [24]. Das gleiche gilt für die Betriebsphase. In dieser Phase sollten ebenso Leistungsmetriken und Kennzahlen gesammelt werden, damit jede Abweichung der im Voraus geplanten Anforderungen aufgenommen und angepasst werden können [24].

2.3.6 Definition einer ungeeigneten Applikation

Auf Basis der beschriebenen Einflussgrößen soll eine ungeeignete Applikation in dieser Arbeit folgendermaßen definiert sein:

- Die Applikation hat keine Skalierungsmechanismen – Skalierbarkeit und Elastizität.
- Die Applikation kann nur ein:e gleichzeitige:n Nutzer:in zulassen - Durchsatz.
- Die Antwortzeiten der Applikation betragen 2-5 Sekunden.
- Alle Nutzer:innen reihen sich in der Applikation in einer Warteschlange ein, bis die Applikation wieder frei ist. Somit steigert sich die Antwortzeit um die Warteschlangenlänge mal der Antwortzeit pro Nutzer:in.

Die folgenden Kapitel beschreiben, mit welchen Techniken ein System um eine Cloud-Applikation - die der genannten Definition einer ungeeigneten Applikation entspricht – realisiert werden kann, um diese hochverfügbar zu machen und nach Qualitätsmerkmalen aus Abschnitt 2.3 zu optimieren.

2.4 Lastverteilungsalgorithmen

2.4.1 Wie kann Lastverteilung die Bereitstellung einer Cloud-Applikation verbessern

Lastverteilung ist ein fundamentaler Bestandteil in Server/Datacenter Infrastrukturen und maßgebend für die Leistung von Applikationen gehostet auf Servern/Datacentern [23]. Durch die Verteilung der Anfragen auf mehrere Systeme, replikative Server oder Server getrennt in ihren Aufgaben, ermöglichen Lasterverteiler eine Entlastung der einzelnen Systeme. Als Resultat erhöht sich der Durchsatz und verringern sich Antwortzeiten [33]. Im Folgenden soll die Grundlagen von Lastverteilung erörtert werden sowie die verschiedene Algorithmen zur Realisierung von Lastverteilung vorgestellt werden.

2.4.2 Grundlagen

Anfänglich gilt es folgende Unterscheidungen zu treffen:

Hardware- und Software-Lastverteilern. Hardware-Lastverteilung sind meist dedizierte Server oder Switches (z.B. OpenFlow Switch) mit entsprechender Firmware. Diese sind stark leistungsorientiert und für den jeweiligen Einsatz optimiert. Daher sind sie aber auch Kosten intensiver, dennn es wird schlichtweg Hardware benötigt. Das gleiche Ziel

erfüllen Software-Lastverteiler. Diese sind darauf ausgelegt eine flexiblere Lösung zu bieten [14].

Layer-4- und Layer-7-Lastverteiler. Lastverteilung kann im OSI-Referenzmodell [35] auf verschiedenen Ebenen angesiedelt werden. Lastverteiler in Layer-4 (Transport-Layer) verwalten den eingehenden Traffic anhand der Netzwerkinformationen wie Applikationsports und Protokolle, ohne dabei den Inhalt der Nachrichten nachzuvollziehen. Dadurch können Nachrichten schnell und sicher weitergeleitet werden, da diese weder entschlüsselt noch inspiziert werden müssen. Es kann aber auch keine Entscheidung bezüglich des Inhalts der Nachrichten getroffen werden.

Layer-7-Lastverteiler arbeiten auf dem Applikations-Layer. Der Lastverteiler kann auch anhand des Medientypen entscheiden, an welche Ressource die Anfrage weitergeleitet wird. Durch die Entschlüsselung und das Inspizieren der Nachrichten ist die Leistung des Layer-7-Lastverteilers schlechter im Vergleich zum Layer-4-Lastverteiler.

Lastverteilung auf dem Applikations-Layer ermöglicht es aber intelligentere Entscheidungen zu treffen. Durch das Setzen von Cookies und die daraus resultierende Identifikation von einzelnen Nutzenden, können Sitzungen beispielsweise persistiert werden. Layer-7-Lastverteiler ermöglichen zudem die Implementierung von Caching-System, da die Inhalte der Nachrichten bekannt sind [5].

Im Weiteren werden Lastverteiler oder Loadbalancer als Layer-7 Software-Lastverteiler verstanden. Für Lastverteiler existieren verschiedene Herangehensweisen, um Anfragen möglichst effizient zu verteilen. In den nächsten Kapiteln werden eine Auswahl an Algorithmen zur Lastverteilung genau beleuchtet und verglichen.

Vorbemerkung. Es wird davon ausgegangen, dass der Lastverteiler Zugriff auf eine Sammlung an replizierten Servern in Form einer Liste (Serverliste) hat. Der Lastverteiler kann jeden Server aus dieser Liste auswählen und die eingegangene Anfrage an diesen Server verteilen. Ebenso wird für jeden Server die aktuelle Auslastung – Anzahl der Nutzer:innen – festgehalten.

2.4.3 Zufallsstrategie

Der Lastverteiler wählt hierbei zufällig einen Server aus der Serverliste. Die Anfrage wird an den gewählten Server weitergeleitet. Die Anzahl der Nutzer:innen wird um eins inkrementiert. Wenn die Sitzung abgelaufen ist, wird die Anzahl wieder um eins reduziert.

2.4.4 Round-Robin

Der Lastverteiler wählt den Server anhand des Serverindexes. Dieser wird errechnet, durch $S_x = i \mod len(Serverliste)$, mit i gleich dem Index des zuletzt genutzten Servers, $i_0 = 0$. Die Anzahl der Nutzer:innen wird um eins inkrementiert. Wenn die Sitzung abgelaufen ist, wird die Anzahl wieder um eins reduziert.

2.4.5 Gewichteter Round-Robin

Hierbei wird jedem Server eine Gewichtung zugewiesen, die auf Kriterien basiert, die administrativ für das System ausgewählt werden. Das am häufigsten verwendete Kriterium ist die Kapazität des Servers zur Verarbeitung des Datenverkehrs. Je höher die Gewichtung desto größer ist der Anteil der Client-Anfragen, die der Server erhält. Wenn beispielsweise Server A eine Gewichtung von 3 und Server B eine Gewichtung von 1 zugewiesen wird, leitet der Lastverteiler 3 Anfragen an Server A für je eine, die er an Server B sendet [13].

2.4.6 Erweiterter gewichteter Round-Robin

Der erweiterte gewichtete Round Robin ist ein probabilistischer, dynamischer Algorithmus. Der Advanced Weighted Round Robin (AWRR) setzt die Wahrscheinlichkeiten für jede Ressource auf die gleiche Weise fest, wie es der WRR für seine Gewichte realisiert. Der Hauptunterschied zwischen den beiden Methoden besteht darin, dass die AWRR-Strategie die Wahrscheinlichkeiten jeder Ressource anhand ihres Zustandes dynamisch ändert. Genauer gesagt, beobachtet die AWRR-Strategie die Auslastung jeder Ressource und wenn diese höher wird, senkt sie die Wahrscheinlichkeit, dass diese Ressource ausgewählt wird und umgekehrt. Der Faktor, um den die Wahrscheinlichkeit steigt oder sinkt, basiert auf der Gewichtung der Ressource. Die Gewichte der jeweiligen Server werden folgendermaßen berechnet:

Wir betrachten ein System mit m als Gesamtzahl der Nodes und $N_1, ..., N_m$ als Anzahl der Server der jeweiligen Node. Die durchschnittliche Service Zeit ist $u_1, ..., u_m$ und $U_1, ..., U_m$ die aktuelle Auslastung des Servers i. P_i ergibt die Wahrscheinlichkeit, dass ein Server gewählt wird. [37]

$$TotalWeight = \sum_{i=1}^{m}(\frac{N_i}{\mu_i}) \tag{2.1}$$

$$Weight_i = \frac{\frac{N_i}{\mu_i}}{TotalWeight} \tag{2.2}$$

$$DynamicTotal = \sum_{i=1}^{m}(Weight_i * (1 - U_i)) \tag{2.3}$$

$$P_i = \frac{Weight_i * (1 - U_i)}{DynamicTotal} \tag{2.4}$$

2.4.7 Pseudo-Code Lastverteilungsalgorithmen

Zufallsstrategie

1: **for** *task* in *jobs* **do**
2: $random \leftarrow$ getRandomNumber$(0, len(Servers))$
3: $selectedServer \leftarrow Servers[random]$
4: DispatchTask(task, selectedServer)
5: **end for**

Round-Robin

1: **for** *task* in *jobs* **do**
2: $i \leftarrow$ GetNextIndex
3: $selectedServer \leftarrow Servers[i]$
4: DispatchTask(task, selectedServer)
5: **end for**

Gewichteter Round-Robin

1: **for** *task* in *jobs* **do** ▷ Wahrscheinlichkeit bezogen auf das Gewicht
2: $selectedServer \leftarrow$ getServerByWeight
3: DispatchTask(task, selectedServer)
4: **end for**

Erweiterter gewichteter Round-Robin

1: **for** *task* in *jobs* **do**
2: calculateProbabilities
3: *selectedServer* ← getSiteBasedOnProbabilities ▷ mit höchster Wahrscheinlichkeit wählen
4: DispatchTask(task, selectedServer)
5: UpdateWeights
6: **end for**

Aufgrund der unterschiedlichen Arbeitsweisen und nähe am System der Algorithmen, werden die Unterschiede im praktischen Teil dieser Arbeit, getestet und die Metriken der jeweiligen Algorithmen verglichen.

2.5 Skalierungsmechanismen

2.5.1 Wie kann Skalierung die Bereitstellung einer Cloud-Applikation verbessern

Wie bereits erörtert sind der Durchsatz und die Antwortzeit einer Cloud-Applikation die maßgebenden Faktoren zur Bestimmung der Leistung der Applikation. Durch effizientes Skalieren einer Cloud-Applikation können bei steigender Last die Serverressourcen dynamisch angepasst werden, um weiterhin die bereitgestellten Services effektiv nutzen zu können. Im Folgenden werden sowohl die angewandten Mechanismen zum Skalieren als auch die durch die Mechanismen resultierende Elastizität der Applikation, genauer erörtert [25].

2.5.2 Grundlagen

Im Kontext von Skalierungsmechanismen muss zwischen vertikaler und horizontaler Skalierung unterschieden werden. Vertikale Skalierung ermöglicht mehr Rechenleistung durch das Hinzufügen von mehr Ressourcen (im Sinne von CPU, RAM oder ganzen Servern). Horizontale Skalierung erreicht mehr Rechenleistung durch das Hinzufügen weiterer Applikationen oder virtueller Maschinen. In dieser Arbeit wird im Weiteren hauptsächlich

von horizontaler Skalierung gesprochen. Die nächsten Abschnitte erörtern die verschiedenen gängige Techniken, um Skalierung zu erreichen. [9]

Vorbemerkung. Es wird davon ausgegangen, dass ein Observer der die Skalierung steuert Zugriff auf die Metriken und Zahl der Nutzer:innen aller Subsysteme hat.

2.5.3 Geplante Skalierung

Skalierung kann bereits durch gute Planung erzielt werden. Wenn die Nutzungszeiten gut abgeschätzt werden können, werden anhand dieser Nutzungsdaten die benötigte Anzahl von Applikations-Replikationen gestartet oder gegebenenfalls gestoppt [6]. Um geplante Skalierung jedoch effizient nutzen zu können müssen entweder ausreichend historische Daten der Nutzung vorliegen oder die Hauptnutzungszeiten verlässlich geschätzt werden. Dadurch wird jedoch die Elastizität des Systems eingeschränkt, da feste Zeiten keine Reaktion auf außerordentliche Last zulassen [6].

2.5.4 Vorhersagen-basierte Skalierung

Durch Vorhersagen, kann Skalierung einer Cloud-Applikation erreicht werden. Am Beispiel der Google Cloud Compute Engine [11] prognostiziert die Compute Engine die zukünftige Last basierend auf dem Verlauf der Container/Instanzen und skaliert diese für die vorhergesagte Last, sodass neue Instanzen für den Empfang bereit sind. Es können jedoch - ähnlich wie bei der geplanten Skalierung - erst verlässliche Prognosen erstellt werden, wenn ausreichend Daten gesammelt wurden [11].

2.5.5 Skalierung basierend auf lokalen Schwellenwerten

Bei der Verwendung von einer lokalen Schwellenwert-Technik [26] wird die aktuelle Auslastung des Hosts mit einem unterem und oberen Schwellenwert verglichen. Werden diese Schwellenwerte innerhalb einer Reihe von Messungen unter- oder überschritten, wird der Host beim Überschreiten als überlastet und beim Unterschreiten als unterlastet gekennzeichnet.

Ist der Host unterlastet, werden alle eingegangen und noch nicht bearbeiteten Anfragen auf andere Hosts umgeleitet. Wenn die aktuell bearbeitete Anfrage beendet ist, wird der Host gestoppt und die Ressourcen freigegeben.

Ist der Host überlastet, werden noch nicht bearbeitete Anfragen zu neu gestarteten Instanzen weitergeleitet.

Um zu häufige Skalierungsentscheidungen zu vermeiden, werden Schonfirsten für die skalierten System eingerichtet, sodass diese nicht automatisch erneut überlasten. Ebenso werden Skalierungsentscheidung erst getroffen, wenn ein Schwellenwert von einem Host n aufeinanderfolgende Male verletzt wird [26].

2.5.6 Skalierung basierend auf globalen Schwellenwerten

Im Gegensatz zu den lokalen Schwellenwerten wird der globale schwellenwertbasierte Ansatz auf der Grundlage der durchschnittlichen Last aller laufenden Hosts definiert. Überschreitet die durchschnittliche Last des gesamten Systems den oberen Schwellenwert, wird das System als überlastet gekennzeichnet. Für jeden Host wird eine Lastmenge bestimmt, welche umverteilt und gegebenenfalls neuen Host zugeteilt wird. Ein globaler Schwellenwert hat weniger Skalierungsaufwände zur Folge, jedoch kann sich eine übermäßige Last auf einem einzelnen Host negativ auf die Antwortzeit des Systems auswirken [26].

2.5.7 Reinforcement Learning

Durch Reinforcement Learning wird anhand einer Nachschlag-Tabelle entschieden, welche Aktion basierend auf den historischen Werten der Auslastung ausgeführt wird. Die Tabelle beschreibt eine gegebene Auslastung und den entsprechenden zu erwartenden Erfolg einer Skalierungsentscheidung. Die möglichen Aktionen sind ¨Scale in¨, ¨Scale out¨ und ¨keine Aktion¨. Die Aktion mit dem höchsten zu erwartenden Erfolg wird basierend auf der Auslastung gewählt. Die Nachschlagtabelle wird durch den Sarsa-Algorithmus (siehe [17] und [4]) anhand der vorangegangenen Aktion, dem eingetretenen Erfolg der Aktion sowie dem erwarteten Erfolg errechnet.

$$Q(s_i) : Q(s_{i-1}) = (1 - \alpha(s_{i-1})Q(s_{i-1}) + \alpha(s_i)(r + \beta * (Q(s_i)))) \tag{2.5}$$

wobei $\alpha(s_{i-1})$ und b die Lernfaktoren bezeichnen, welche den Einfluss des direkten und zu erwartenden Erfolgs beschreiben. Der direkte Erfolg ist basiert auf dem Auslastung-Fehler, welche beschrieben wird, durch die Differenz zwischen gemessener und erwarteter Auslastung. Wenn der Fehler zwischen $Q(s_{i-1})$ und $Q(s_i)$ sinkt, ist der direkte Erfolg

größer als 1. Wenn der Fehler jedoch steigt, ist der direkte Erfolg niedriger als 1. $\alpha(s_i)$ soll definiert sein als:

$$\alpha(s_i) = imp * (\frac{dur}{dur + visits(s_i)}) \tag{2.6}$$

wobei $visits(s_i)$ die Anzahl der Lernschritte entspricht, welche das System gemacht hat, um den jetzigen Stand zu erreichen. *imp* beschreibt die Einfluss-Rate (Impact) des Lernens. *dur* (Duration) zeigt an, wie lange das System lernt. Die Standard-Werte hierbei sind $imp = 0,2$ $dur = 80$ [26].

Im praktischen Teil dieser Arbeit soll der Skalierungmechanismus der globalen Schwellenwert-Skalierung getestet werden und die Metriken der jeweiligen Einflüssen erörtert werden. Die geplante Skalierung, Skalierung durch Prognose sowie Reinforcement Learning sind im Kontext dieser Arbeit nicht geeignet, da nicht ausreichende historische Werte in einer Testumgebung ausgewertet werden können.

2.6 Wartenschlangenmodelle

2.6.1 Warteschlange in Cloud Systemen

Cloud-Computing soll mithilfe der Verlagerung der Recheninfrastruktur in das Internet eine Reduzierung der Kosten für Management und Wartung erreicht werden. Jedoch bleibt die Frage, ob die dadurch erreichte Leistung einen QoS (Qualtity of Service) garantieren kann. Das Warteschlangen-Modell beschrieben von Jordi Vilaplana et al. [39] ist designt, um ein QoS basierend auf der Antwortzeit der Services zu garantieren. In diesem Kapitel werden die Warteschlange Modelle M/M/m und M/M/1 erläutert, das Modell von Vilaplana et al. besprochen und auf das System dieser Arbeit angepasst.

Um Vilaplanas Modell anwenden zu können müssen folgende grundlegende Konzepte etabliert werden.

2.6.2 Grundlagen

Little´s Theorem. definiert die durchschnittliche Zahl von Nutzer:innen (N). Das Theorem gilt Systemen in stetigem Zustand und ist durch folgende Gleichung bestimmt [3]:

$$N = \lambda * T \tag{2.7}$$

wobei λ die durchschnittliche Ankuftsrate der Nutzer:innen ist und T die durchschnittliche Servicezeit beschreibt.

Dadurch können drei grundlegende Beziehungen abstrahiert werden:

- N wird wachsen, wenn λ oder T wachsen.
- λ wird wachsen, wenn N wächst oder T sinkt.
- T wird wachsen, wenn N wächst oder λ sinkt.

Durch Littles Theorem können grundsätzliche Prozesse in Warteschlagensystemen verstanden werden. Vertiefend werden folgende Charakteristiken definiert:

- Ankuftsprozesse
  - Eine Wahrscheinlichkeitsdichtefunktion bestimmt die Ankünfte der Nutzer:innen im System.
  - In einem Nachrichtensystem beschreibt dies die Ankunft der Nachrichten und deren Wahrscheinlichkeitsverteilung.
- Serviceprozess
  - Die Wahrscheinlichkeitsverteilung bestimmt die Servicezeit des Systems.
  - In einem Nachrichtensystem beschreibt dies die Nachrichtenübertragungszeit. Die Übertragung ist direkt proportional mit der Länge der Nachricht. Dieser Parameter bezieht sich indirekt auf die Nachrichtenlängenverteilung.
- Anzahl der Server
  - Die Anzahl der Server die für die Nutzer:innen verfügbar sind.

  - In einem Nachrichtensystem beschreibt dies die Anzahl der Verbindungen zwischen der Quell-Node und den Ziel-Nodes.

Häufig wird für Warteschlangen-Modelle folgende Notation (verallgemeinerte Kendall´s Notation) verwendet: A/B/C [36]

welche abgeleitet aus den beschriebenen Charakteristiken die verwendeten Konventionen für jeden Prozess beschreibt. A beschreibt den Ankuftsprozess, B den Serviceprozess und C die Anzahl der Server. A und B können zum Beispiel durch die folgenden Zeichen beschrieben werden [36]:

- M (Markov oder memoryless) –> exponentielle Wahrscheinlichkeitsdichte (Poisson Ankunft, exponentielle Servicezeit).
- D (Degenerierte Verteilung) –> alle Nutzer:innen haben den gleichen Wert.
- G (Generell) –> jede beliebige Wahrscheinlichkeitsverteilung.

2.6.3 Poisson-Ankuftsprozess

M/M/1-Warteschlangensysteme gehen von einem Poisson-Ankunftsprozess aus. Um M/M/1 weiter vertiefen zu können, muss der Begriff des Poisson-Ankunftsprozesses etabliert werden.

Der Poisson-Ankunftsprozess beschreibt die Anzahl der Vorkommnisse eines Phänomens in einem kontinuierlich Zeitintervall. Zum Beispiel die Anzahl der bearbeiteten Google-Anfragen in einer Sekunde. Die Events sind hierbei zählbar und weisen einen messbaren Verzug zwischen den Anfragen auf.

Der Poisson Ankunftsprozess sei definiert durch:

Sei $\lambda > 0$ gegeben. Der kontinuierliche Prozess N(t),t Element aus [0, unendlich) ist ein Poisson Prozess mit der Rate λ, wenn alle folgenden Bedingungen gelten [30]:

- N(0) = 0
- N(t) hat unabhängige Inkremente
- Die Anzahl der Ankünfte eines jeden Intervalls der Länge $\tau > 0$ ist $Poisson(\lambda\tau)$ verteilt.

Wenn angenommen wird, dass der Poisson Ankunftsprozess als eine eindimensionalen Poisson Streuung – z.B. die Anzahl der Regentropfen, die auf einen gegebenen Quadratmeter eines Daches in einer bestimmten Zeitspanne fällt, ist Poisson verteilt - mit Ankünften in einem gegebenen Zeitintervall und die Formel für die Poisson-Verteilung betrachten [16], resultiert folgende Gleichung für die Wahrscheinlichkeit von n Ankünften in einem Zeitintervall t [30]:

$$P_n(t) = \frac{(\lambda t)^n}{n!} e^{-\lambda t} \tag{2.8}$$

Beweis siehe: MIT OpenCourseWare - Theorem 2.2.3 - Proof 1.

Um diesen Zusammenhang zu verdeutlichen, wird im folgenden ein Spezialfall dieser Verteilung betrachtet und durch ein Beispiel erläutert.

Gegeben sei n = 0, daraus entsteht folgende Gleichung:

$$P_0(t) = e^{-\lambda t} \tag{2.9}$$

Hierdurch resultiert die Wahrscheinlichkeit, dass in einem gegebenen Zeitintervall keine Ankünfte eintreten und, dass die Funktion negativ exponentiell zur Länge des Intervalls ist. Anhand eines Beispiels wird dies verdeutlicht:

Angenommen, dass auf einer Autobahn durchschnittlich alle 10 Sekunden ein Auto die Autobahn erreicht ($\lambda = 0.1$). Durch die Gleichung 2.8 kann angenommen werden, dass die Wahrscheinlichkeit kein Auto zu sehen innerhalb des Zeitintervalls stark sinkt. Wird die Abbildung 2.5 betrachtet, wird deutlich, dass innerhalb eines 20 Sekunden Zeitfenster eine 10-prozentige Wahrscheinlichkeit besteht, dass man kein Auto sieht [10].

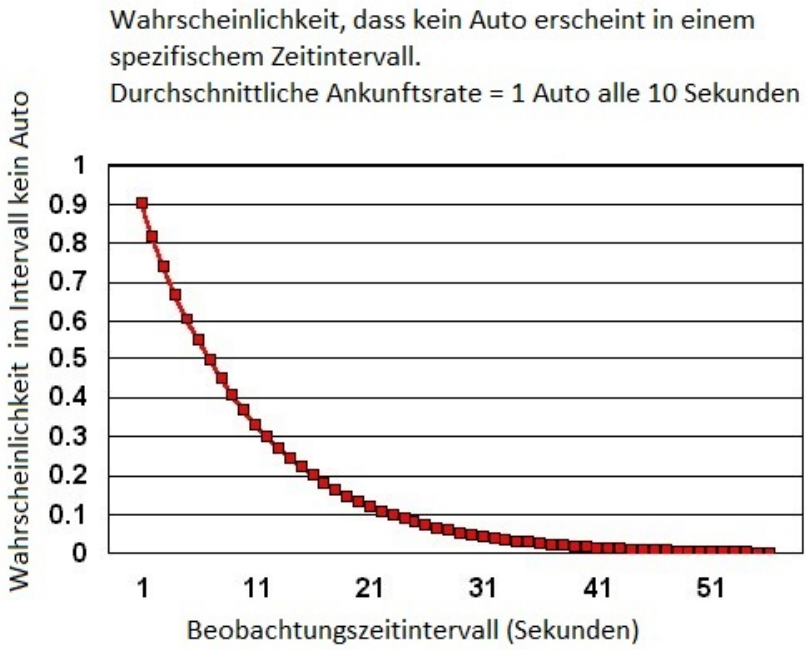


Abbildung 2.5: Wahrscheinlichkeitsverlauf der Ankunft eines Autos [10]

Hierdurch können jetzt Systeme modelliert werden, welche den Anforderungen eines Poisson-Prozesses und somit der Grundlage der genannten Warteschlagensysteme entsprechen.

2.6.4 M/M/1 - Warteschlange

M/M/1 bezieht sich auf negativ exponentielle Ankunfts- und Servicezeiten (Poisson verteilt) mit einem einzigen Server. Durch den Forschungsstand, ist es das in Analysen am häufigsten verwendete Warteschlangensystem.

Zunächst soll die durchschnittliche Antwortzeit einer M/M/1 Warteschlange definiert werden. Da sich die M/M/1 Warteschlange in einem stetigen Zustand befindet (per Definition [3]), kann hierbei Little´s Theorem angewandt werden.

$$N = \lambda * T \tag{2.10}$$

dadurch ergibt sich für die durchschnittliche Antwortzeit:

$$T = \frac{N}{\lambda} \tag{2.11}$$

$$T = \frac{\frac{\lambda}{\mu-\lambda}}{\lambda} \tag{2.12}$$

$$T = \frac{1}{\mu - \lambda} \tag{2.13}$$

wenn für $N = \frac{\lambda}{\mu-\lambda}$ und $\mu =$ die durchschnittliche Servicerate gilt.

2.6.5 M/M/m - Warteschlange

Eine M/M/m Warteschlange unterscheidet sich von einer M/M/1 Warteschlange grundsätzlich nur in der Anzahl der Server. Ausgegangen wird hierbei nicht von einem Server sondern m identischen Servern.

Um Vilaplanas Modell anwenden zu können, muss auch hier die durchschnittliche Antwortzeit definiert werden. Durch m Server können mehrere Nutzer:innen gleichzeitig bedient werden, wodurch sich eine Wahrscheinlichkeit eine:r Nutzer:in ergibt, dass diese/r in der Warteschlange warten muss und eine Anzahl an wartenden Personen.

Die Wahrscheinlichkeit für eine wartende Person wird wie folgt berechnet[27, 3]:

$$P_i = \begin{cases} P_0 \frac{(m\rho)^i}{i!}, \text{ für } i \leq m \\ \\ P_0 \frac{m^m \rho^i}{m!}, \text{ für } i > m \end{cases} \tag{2.14}$$

Für $\rho = \frac{\lambda}{m\mu}$. Dadurch gilt: $\sum_{i=0}^{\infty} P_i = 1$ und:

$$P_0 = \big[\sum_{i=0}^{m-1} \frac{(m\rho)^i}{i!} + \frac{(m\rho)^m}{m!(1-\rho} \big]^{-1} \tag{2.15}$$

$$\begin{aligned} P_Q &= \sum_{i=m}^{\infty} P_i \\ &= \sum_{i=m}^{\infty} \frac{P_0 m^m \rho^i}{m!} \\ &= \frac{P_0 (m\rho)^m}{m!} \sum_{i=m}^{\infty} \rho^{i-m} \\ &= \frac{P_0 (m\rho)^m}{m!(1-\rho)} \end{aligned} \tag{2.16}$$

Die Anzahl der wartenden Personen ergibt sich durch:

$$\begin{aligned} N_Q &= \sum_{i=m}^{\infty} (i-m) P_i = \sum_{i=0}^{\infty} i P_{i+m} \\ &= \sum_{i=0}^{\infty} i P_0 \frac{m^m \rho^{i+m}}{m!} = \frac{P_0 (m\rho)^m}{m!} \sum_{i=0}^{\infty} i\rho^i \\ &= \frac{P_0 (m\rho)^m}{m!} \frac{\rho}{(1-\rho)^2} \\ &= P_Q \frac{\rho}{1-\rho} \end{aligned} \tag{2.17}$$

Dadurch kann jetzt die durchschnittliche Wartezeit berechnet werden:

$$W = \frac{N_Q}{\lambda} = \frac{\rho P_Q}{\lambda(1-\rho} \tag{2.18}$$

Die durchschnittliche Antwortzeit ergibt sich durch:

$$T = \frac{1}{\mu} + W = \frac{1}{\mu} + \frac{\rho P_Q}{\lambda(1-\rho)} = \frac{1}{\mu} + \frac{P_Q}{m\mu - \lambda} \tag{2.19}$$

2.6.6 Jackson Netzwerke

Ein Jackson Netzwerk ist ein System aus m Serivces. Für jeden Service i gilt (i = 1,2,. . . ,m):

1. eine unendliche Wartschlange
2. Nutzer:innen treten in das System in einem Poisson-Ankunftsprozess ein

3. Die Servicezeit eines Servers s_i ist exponentiell verteilt nach dem Parameter μ.

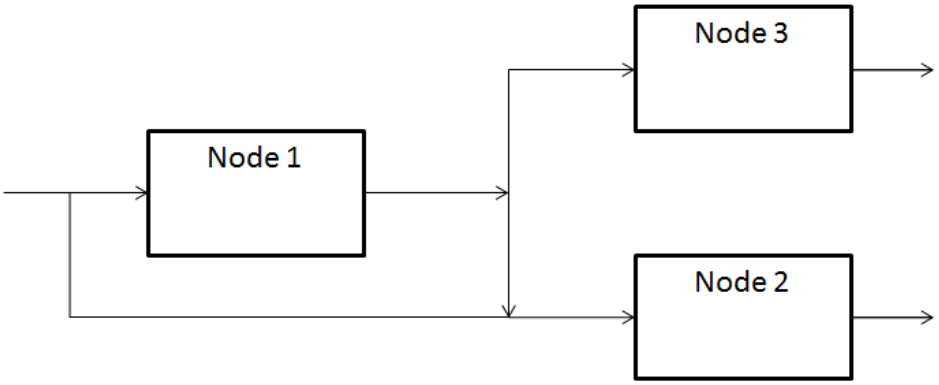


Abbildung 2.6: Vereinfachte Darstellung eines Jackson Netzwerks [15]

Ein solches Netzwerk hat folgende Eigenschaft:

In stetigem Zustand verhält sich jeder Service $j(j = 1, 2, ..., m)$ in einem Jackson Netzwerk als eine unabhängige M/M/s_j Warteschlange mit der Ankunftsrate:

$$\lambda_j = a_j + \sum_{i=1}^{m} \lambda_i \rho_{ij} \text{ , wobei } s_j, \mu_j > \lambda_j \tag{2.20}$$

Darüber hinaus kann zwischen offenen und geschlossenen Jackson Netzwerken unterschieden werden. Ein offenes Jackson Netzwerk hat im Gegensatz zu einem geschlossenen Pfade, welche von außerhalb in das Netzwerk und aus dem Netzwerk führen. Weiterführend werden die Warteschlangen in einem offenen System betrachtet, wodurch keine rückläufigen Wartezeiten oder Schleifen entstehen. [3]

Durch diese fundamentalen Begriffe und Formel der Warteschlangen-Theorie, kann nun auf Vilaplanas Modell eingegangen werden.

2.7 Warteschlangen Modell für Cloud Applikationen

Jordi Vilaplana et al. [39] bilden ein Multi-Server-System auf Grundlage eines offenen Jackson Netzwerks. Das Netzwerk wird über einen ¨Entering Server¨ (ES) betreten. Dieser Server wird als Lastverteiler betrachtet, welcher die Anfragen an die restlichen Server des Netzwerks (¨Processing Server¨, PS_i) weiterleitet. Der ES wird als M/M/1-Warteschlange verstanden. Die Service- und Ankunftsrate sind modelliert als exponentielle Verteilung (siehe 2.6.3) mit den Parameter λ und N, wobei $\lambda < N$. Der Lastverteiler koordiniert die eingehenden Anfragen anhand der durchschnittlichen Auslastungen der Server im Netzwerk.

Ein PS wird als Node betrachtet, welche alle von Nutzer:innen angefragten Services bearbeitet. Somit kann ein PS_i als eine M/M/m-Wartschlange betrachtet werden. Jede PS_i ist identisch und verarbeitet alle Anfragen mit gleicher Servicerate und ist gleich zu μ, für $\mu = \mu_i, i = 1, ..., m$. Jede PS_i Node hat mit einer Wahrscheinlichkeit delta Zugriff auf einen Datenbank-Server DS. Der DS repräsentiert alle I/O Zugriffe. Für den DS wird ebenfalls eine M/M/1 Warteschlage modelliert, mit exponentiellen Ankunfs- und Serviceraten beschrieben durch die Regeln eines Jackson Netzwerks $\delta * \gamma$ und N_{DS}.

OS repräsentiert einen ¨Output Server¨, welche eine Node bildet, um die Anfrage übers Internet zurück zur/zum Nutzer:in zu senden (CS).

CS ist der ¨Client Server¨, welcher Anfragen entsprechend der Ankunftsrate (λ) zum Lastverteiler (ES) versendet. Der CS empfängt ebenfalls die Anfragen des OS, bis die Anfrage komplett abgefertigt wurde. OS und CS werden auch als M/M/1-Wartschlange modelliert. Diese können im Gegensatz zu den anderen Nodes des Systems verschiedene Serviceraten haben.

Die Verbindungen der Nodes durch exponentielle Verteilung in Form von feedforward (keine rückläufigen Pfade) erfüllen die Anforderung an den Poisson-Ankunftsprozess. Somit können die etablierten Definitionen angewandt werden.

λ beschreibt die Rate der Nutzer:innen, die den Lastverteiler verlassen. Durch Jackson können wir annehmen, dass die Ankunftsraten von außerhalb des Systems sowie aller Nodes innerhalb des Systems addiert werden können. Dadurch ergibt sich (klein) gamma. Wenn τ die Wahrscheinlichkeit des Verlassens des offenen Jackson Netzwerks beschreibt, erhalten wir $\gamma = \lambda/(1 - \tau)$.

Die Gesamt-Antwortzeit des Systems (T) kann durch Jackson ebenfalls wie folgt beschrieben werden.

$$T = T_E S + T_P S + T_D S + T_O S + T_C S \tag{2.21}$$

Wobei $T_E S$:

$$T_{ES} = \frac{1/N}{1 - \lambda\ N} \tag{2.22}$$

$T_P S$ repräsentiert die tatsächliche Bearbeitungszeit der Anfrage. m beschreibt das Maximum der PS Nodes und kann als die maximale Rechenleistung interpretiert werden. Durch die Modellierung als M/M/m-Warteschlange ergibt sich für die Antwortzeit von PS folgendes:

$$T_{PS} = \frac{1}{\mu} + \frac{C(m, \rho)}{m\mu - \gamma} \tag{2.23}$$

Wobei C(m,p) der Erlang-C-Formel entspricht (siehe Gleichung 2.16) und $\rho = \gamma/\mu$.

$T_D S$ entspricht der Antwortzeit des als M/M/1 modellierten Datenbank-Servers (DS). Hierbei gilt:

$$T_{DS} = \frac{1/N_{DS}}{1 - \delta\gamma/N_{DS}} \tag{2.24}$$

Die Ankunftsrate des OS wird als Summe der Ankunftsraten des DS und aller PS an der Zusammenführung (siehe Abbildung 2.7) angesehen. Wenn ein PS beim DS anfragt, ergibt sich $\delta\gamma$ als Ankunftsrate. Für den Fall, dass der DS nicht berücksichtigt wird, muss das Komplement dazu summiert werden. Für die totale Ankunftsrate gilt dadurch folgendes:

$$(1 - \delta)\gamma + \delta\gamma = \gamma \tag{2.25}$$

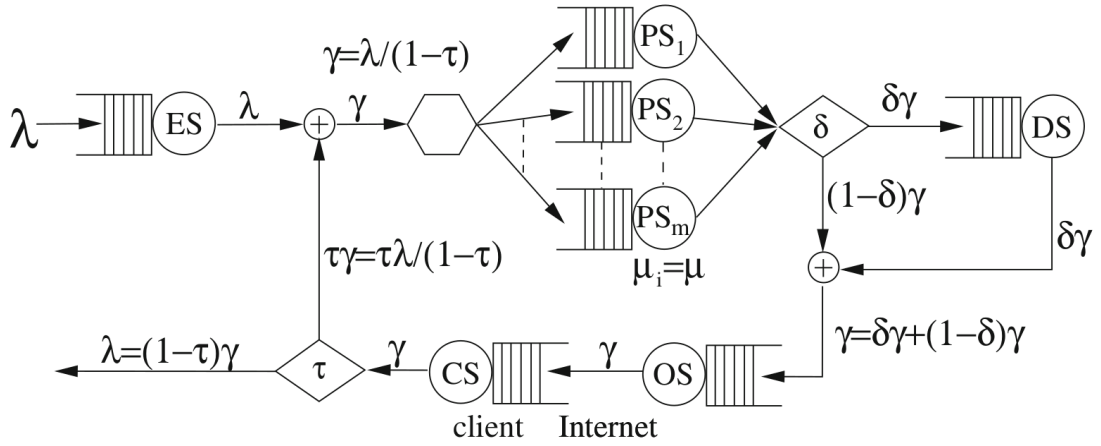


Abbildung 2.7: Modell einer Cloud-Architektur nach Vilaplana [39].

Für die Antwortzeit des OS gilt somit:

$$\begin{aligned} T_{OS} &= \frac{F/O}{1-\gamma/(O/F)} \\ &= \frac{F}{O-\gamma F} \end{aligned} \tag{2.26}$$

wobei die Servicerate als O/F definiert sei. O beschreibt die durchschnittliche Bandbreite des Output-Servers und F die durchschnittliche Antwortgröße des Systems.

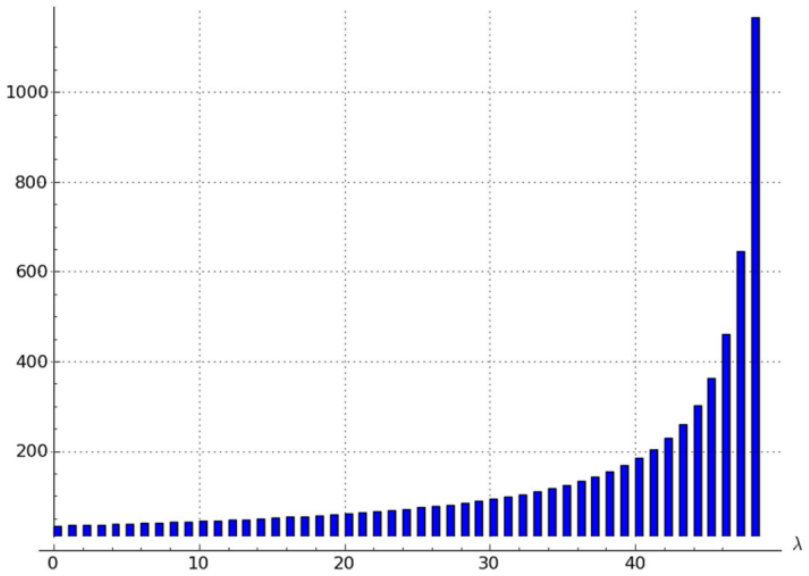
Für CS, welcher als M/M/1-Warteschlange modelliert ist, gilt ebenfalls:

$$\begin{aligned} T_{CS} &= \frac{F/C}{1-\gamma/(C/F)} \\ &= \frac{F}{C-\gamma F} \end{aligned} \tag{2.27}$$

wobei C der Bandbreite des Client-Servers entspricht und F der durchschnittlich empfangen Bytes.

2.7.1 Ergebnisse

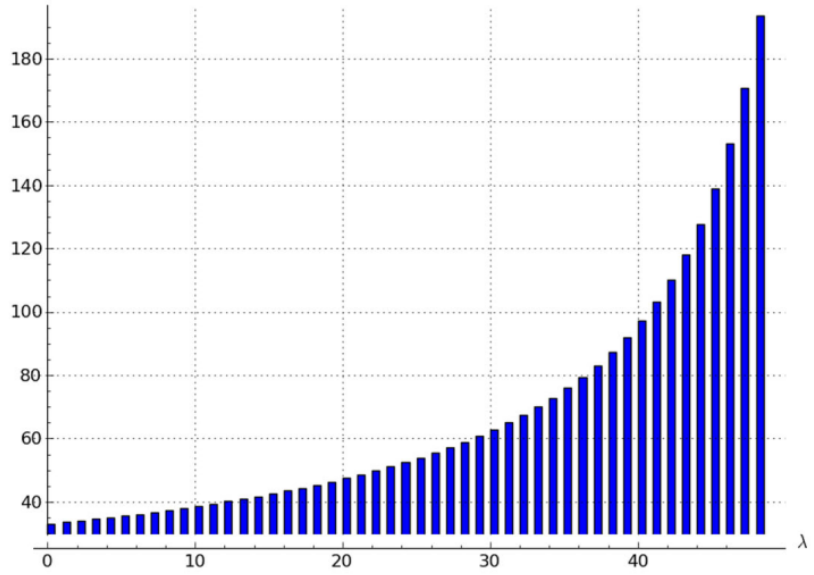
Im Folgenden werden die Ergebnisse Vilaplana et al. [39] vorgestellt und diskutiert. Die Simulation des Modells wurde mit der mathematischen Software Sage 5.3 durchgeführt (siehe Sagemath). Für die verwendeten Grafiken von Vilaplana et al. beschreibt y die Zeitachse.



(a) $m = 1$.

Abbildung 2.8: Totale Antwortzeit (T) des Modells verglichen mit der Ankunftsrate λ. m =1 [39].

Abbildung 2.8 und Abbildung 2.9 zeigen, wie die Ankunftsrate λ die totale Antwortzeit des Systems beeinflusst. Durch den Vergleich der Graphen (a) und (b) aus Abbildung 2.8 und Abbildung 2.9 wird klar, wie sich die Anzahl der Server auf die totale Antwortzeit auswirkt. Hierbei ist eine fast 10-fache Verbesserung abzulesen. Jedoch ist für dieses Modell laut Vilaplana keine Verbesserung mehr zu sehen, wenn mehr als fünf Server eingesetzt werden, da die Systeme sich hier schnell stabilisieren. Dieser Umstand ergibt sich laut Vilaplana vermutlich durch die Auslastung der Nodes PS_i (definiert durch: $\rho = \gamma/\mu$).



(b) $m = 2$.

Abbildung 2.9: Totale Antwortzeit (T) des Modells verglichen mit der Ankunftsrate λ. m = 2 [39].

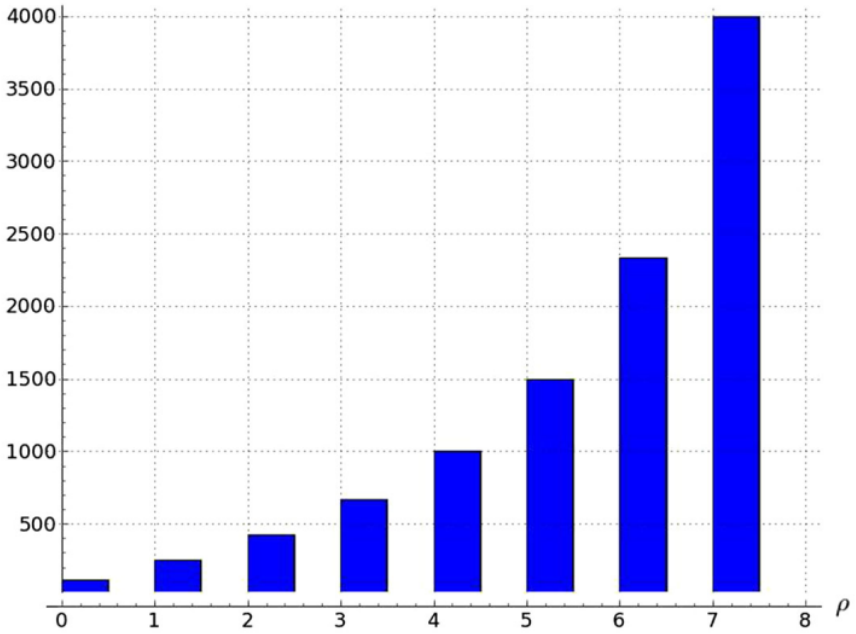
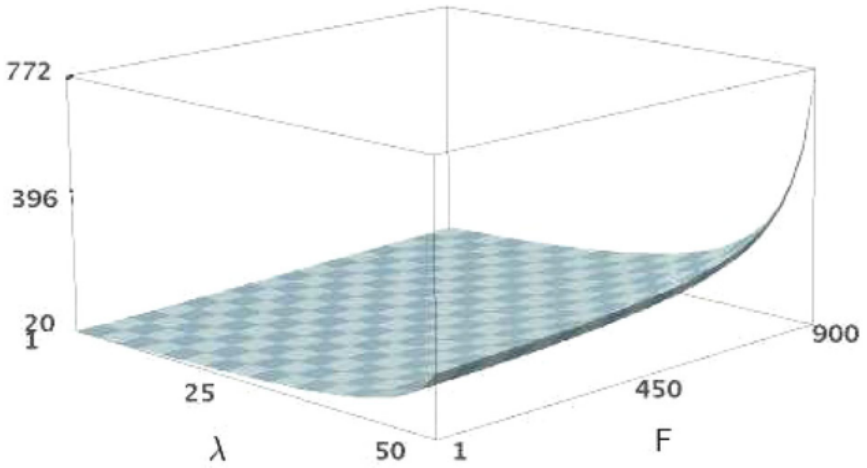


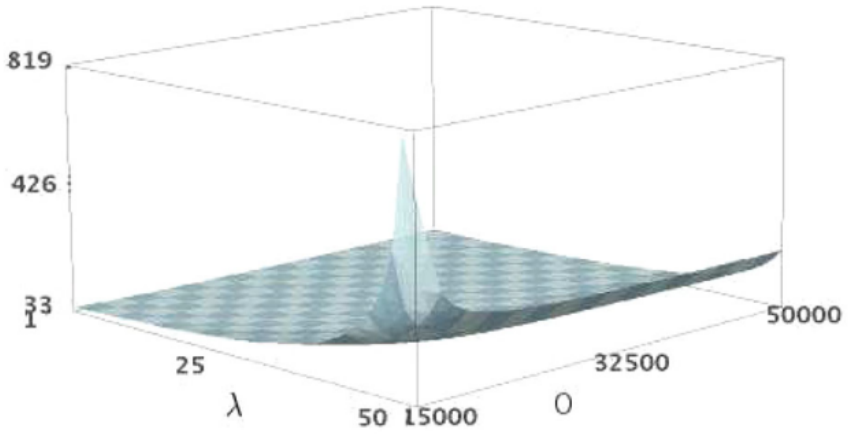
Abbildung 2.10: Totale Antwortzeit (T) des Modells verglichen mit der Auslastung ρ [39].

Bei geringer Serverzahl der PS_i Nodes ist die Antwortzeit abhängig von der Auslastung ρ und der M/M/m-Warteschlangen der PS_i Nodes. Wenn jedoch die Serverzahl bis zu einem Equilibriums-Punkt steigt, ist die M/M/1-Warteschlange des Lastverteilers maßgeblich für die totale Antwortzeit des Systems. Da die Ankunftsrate durch die M/M/1-Wartschlange des Lastverteilers zu Beginn geschleust wird, entsteht dieser Effekt (wenn λ hoch, dann ist für die Ankunftsrate $\gamma = \lambda/(1 - \tau)$) ebenfalls ein hoher Wert abzusehen.). In Abbildung 2.10 ist zu sehen, wie sich die Auslastung der PS_i Nodes auf die totale Antwortzeit des Modells auswirkt. Die Antwortzeit der einzelnen PS Nodes steigt in exponentieller Form, wenn die Auslastung der Node steigt. Wenn die Auslastung sich jedoch Null annähert, entspricht die Antwortzeit fast der Servicezeit. Hierbei ist ein ähnliches Verhalten zu beobachten wie in Abbildung 2.8 und Abbildung 2.9.



(a) T given λ and F.

Abbildung 2.11: Totale Antwortzeit (T) des Modells verglichen mit der Ankunftsrate λ der durchschnittlichen Dateigröße (F) [39].



(b) T given λ and O.

Abbildung 2.12: Totale Antwortzeit (T) des Modells verglichen mit der Ankunftsrate λ der durchschnittlichen Bandbreite (O) [39].

In Abbildung 2.11 und Abbildung 2.12 wird gezeigt, wie sich das System verhält, wenn die Ankunftsrate (λ), durchschnittliche Dateigröße (F) und Bandbreite des ¨Output-Servers¨ verändert werden. Hierbei wird eine grundsätzliche Vermutung bestätigt. Wenn die Dateigröße steigt oder die Bandbreite sinkt, steigt die totale Antwortzeit des Systems exponentiell.

2.7.2 Bottlenecks - Die Engpässe des Systems

Die Abbildungen 2.13 und 2.14 veranschaulichen die Simulation, wenn im System Bottlenecks integriert werden und zeigen, wie diese sich auf die totale Antwortzeit des Systems auswirken. Hierbei wird unterschieden, ob wie in Abbildung 2.13 die Server ein Bottleneck darstellen oder wie in Abbildung 2.14 die Bandbreite.

Im ersten Fall sehen wir eine klare Verbesserung der totalen Antwortzeit durch Verdopplung der Servicerate. Die Erhöhung der Bandbreite hat keinen signifikanten Einfluss, da letztlich die Leistung des Servers ausschlaggeben für die Servicezeit und somit für die Antwortzeit ist. Für den zweiten Fall gilt die Verdopplung der Bandbreite als Lösung des Bottlenecks.

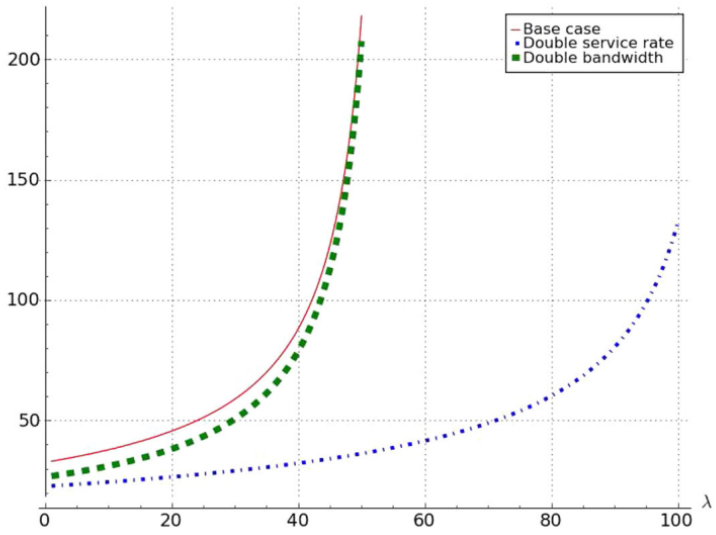


Abbildung 2.13: Totale Antwortzeit (T) verglichen mit der Ankunftsrate λ. Server sind Engpässe [39].

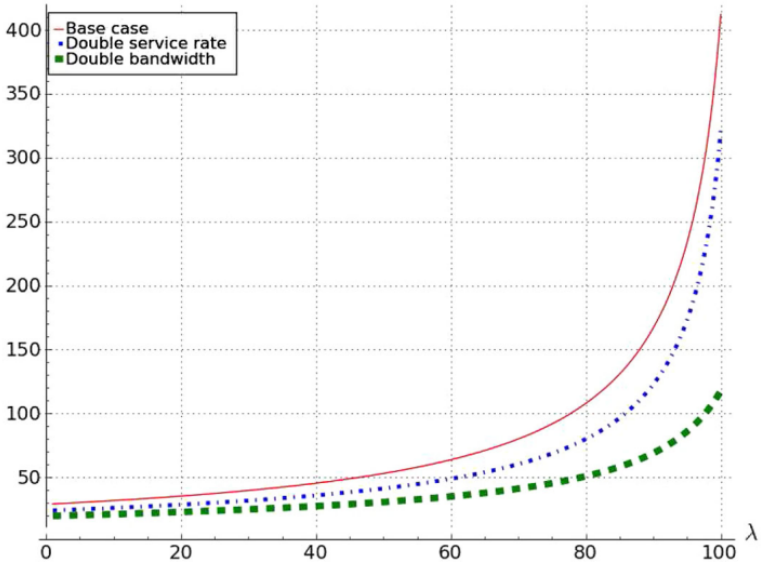
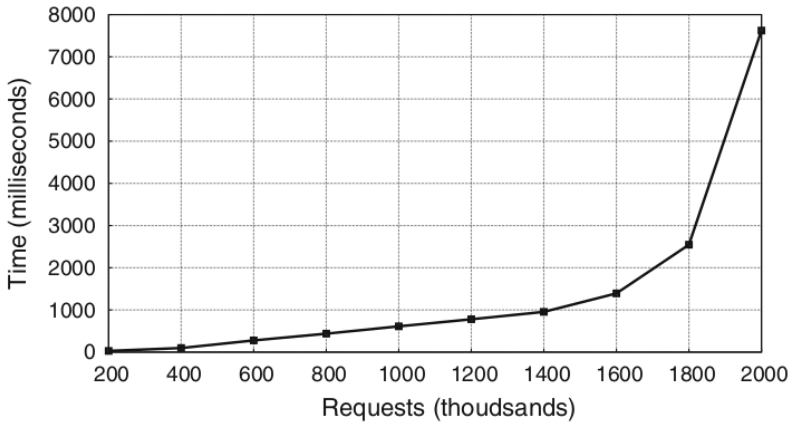


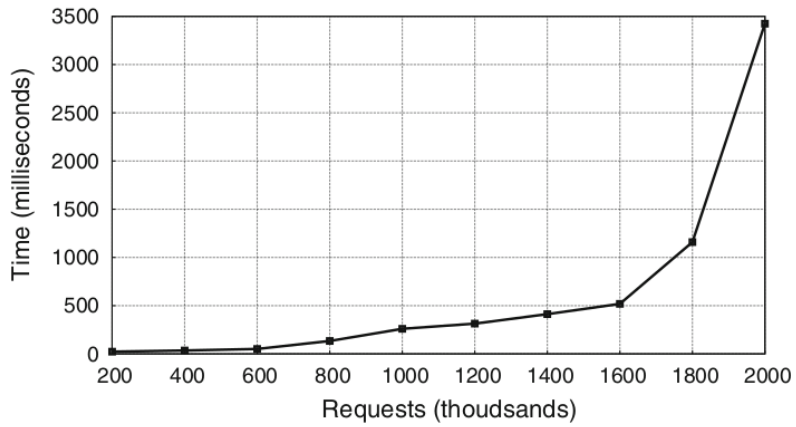
Abbildung 2.14: Totale Antwortzeit (T) verglichen mit der Ankunftsrate λ. Bandbreite ist der Engpass [39].

2.7.3 Validierung

Vilaplana et al. [39] validieren das Modell durch die Implementierung eines Testsystems auf der Opensource-Plattform OpenStack. Openstack erlaubt die dynamische Erstellung und Einstellung von virtuellen Maschinen entsprechend den Anforderungen des Modells. Ein automatischer Skalierungsmechanismus steuert das Starten und Stoppen der Ressourcen. Die Architektur wurde durch verschiedene virtuelle Maschinen (VM) realisiert, welche die einzelnen Knoten des Systems repräsentieren wie den ES (¨Entering Server¨). Jede VM verfügt über 4 GB RAM und zwei Kerne eines AMD Opteron 6100 Prozessors mit je einer Taktfrequenz von 2,1 GHz.



(a) Response time (T) with 1 server.



(b) Response time (T) with 2 servers.

Abbildung 2.15: Antwortzeiten erzeugt durch das OpenStack-System [39].

In Abbildung 2.15 lassen sich bereits Ähnlichkeiten zu den theoretisch simulierten Graphen erkennen. Der größte Gewinn wurde durch die Inkrementation von einem auf zwei Server erreicht – wie in der Simulation. Ebenfalls wie in den Simulationen stabilisiert sich das Modell ab drei Servern und höher. Dadurch ist laut Vilaplana et al. das theoretische Modell validiert und bietet eine größere Relevanz [39].

2.7.4 Anpassung des Modells

Für dieses Arbeit wird im Folgenden das Modell von Vilaplana angepasst nach den Anforderungen einer ungeeigneten Applikation aus Unterabschnitt 2.3.6.

Modell

Wie auch im Modell von Vilaplanas et al. [39] wird der Eintritt in das System über einen Lasterverteiler (¨Entering Server¨ bei Vilaplana) ermöglicht. Dieser wird als M/M/1-Warteschlange realisiert. Alle Anfragen werden vom Lastverteiler an eine Applikation weitergeleitet. Die Sammlung der Applikationen wird als M/M/m Warteschlange modelliert, wobei nur eine Anfrage pro Server gleichzeitig bearbeitet wird, um die Voraussetzung für eine M/M/m-Warteschlagen zu gewährleisten. Im Gegensatz zu Vilaplanas Modell wird auf einen CS, einen DS sowie OS verzichtet, da in dieser Arbeit die Optimierung der Applikationsbereitstellung im Fokus liegt. Der Fokus soll wie im Modell Vilaplanas et al. auf der totalen Antwortzeit des Systems liegen, da hier die beste Aussage über die Qualität des Systems getroffen werden kann.

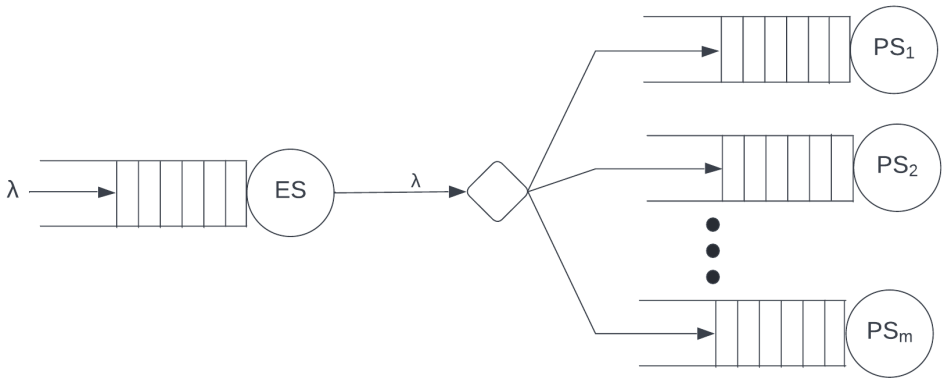


Abbildung 2.16: Anpassung des Modell. Eigene Darstellung orientiert an Vilaplana

Ergebnisse

Ähnlich wie Vilaplana et al. konnte in den Simulationen festgestellt werden, dass das Vergrößern der Serverzahl auf über zwei keinen großen Einfluss auf die totale Antwortzeit nimmt. Der größte Unterschied ist durch die Steigerung von einem Server auf zwei Server zu erkennen siehe Abbildung 2.17 und 2.18). Dieses Phänomen lässt sich durch die Ankunfts- und Servicerate begründen, da das Hinzufügen von Rechenleistung keine Auswirkungen mehr hat, wenn die Ankunftsrate und Servicerate sich in einem solchen Verhältnis befinden, sodass eine Anfrage bereits bearbeitet ist sobald die nächste Anfrage eintrifft. Bei einer hohen Anzahl an Servern werden nur wenige Server berücksichtigt und die Mehrheit befindet sich in einem Leerlauf.

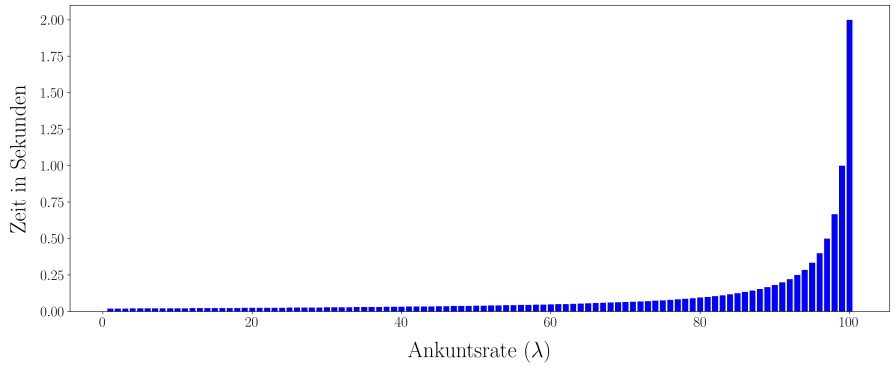


Abbildung 2.17: Totale Antwortzeit (T) des Modells verglichen mit der Ankunftsrate λ. Anzahl der Server 1

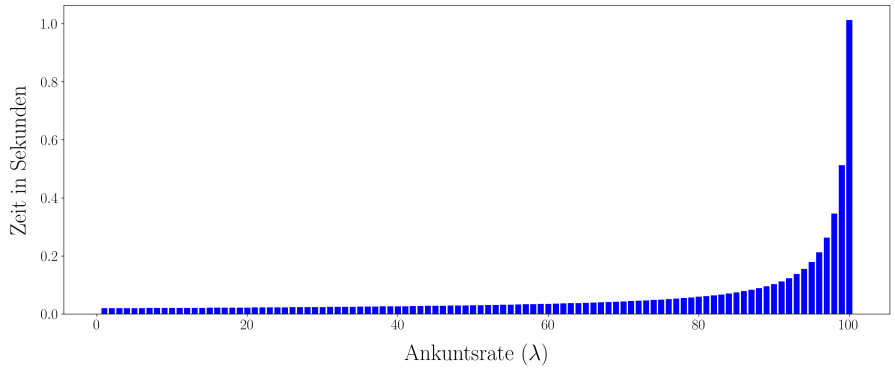


Abbildung 2.18: Totale Antwortzeit (T) des Modells verglichen mit der Ankunftsrate λ. Anzahl der Server 2

In den Abbildungen 2.17 und 2.18 ist eine klare Abnahme der totale Antwortzeit zu erkennen. Hierbei gilt wie bei Vilaplana et al., dass die Steigerung der Systemleistung eine Verbesserung zur Folge hat. Klarer wird dieser Zusammenhang, wenn die M/M/m-Wartschlange als einzelnes System betrachtet wird. In den Abbildungen 2.19, 2.20 und 2.21 lässt sich deutlich erkennen, welchen Einfluss auf die totale Antwortzeit die Anzahl der Server auf das System hat. Ebenso wird klar, dass die Servicerate die Ankunftsrate tilgt und somit einer höhere Serverzahl keinen Einfluss auf die totale Antwortzeit nimmt. Dadurch kann angenommen werden, dass die Serverzahl gleich der Anzahl der Anfragen ist, keine Wartezeiten entstehen. Somit kann das Potential für eine effiziente Skalierung anhand der Anfragen abstrahieren werden. Jedoch ist zu berücksichtigen, dass sich Server auch im Leerlauf befinden können.

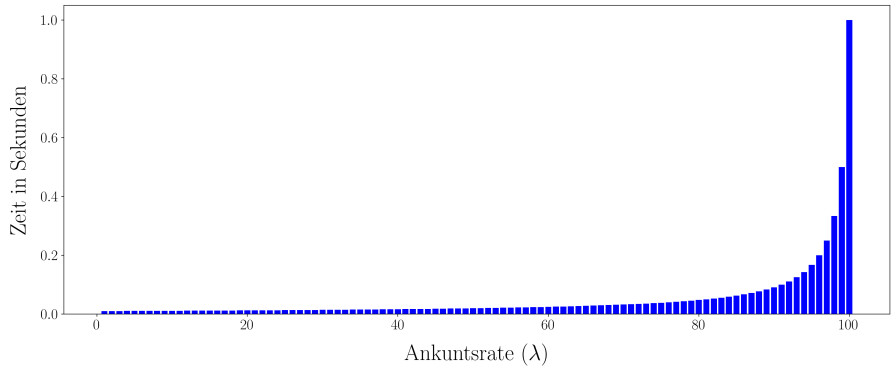


Abbildung 2.19: Totale Antwortzeit (T) der M/M/m-Warteschlange verglichen mit der Ankunftsrate λ. Anzahl der Server 1

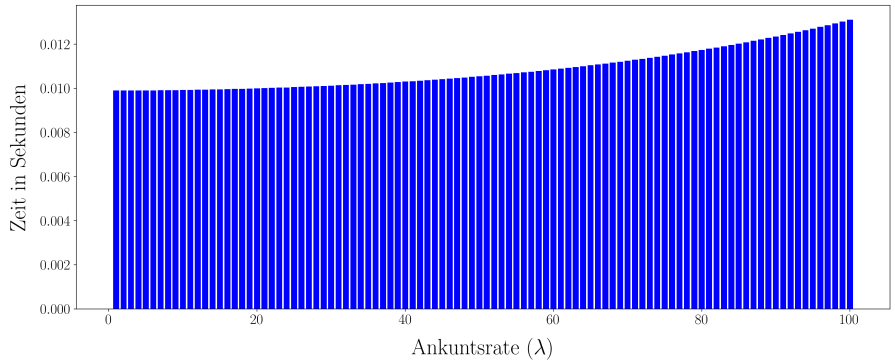


Abbildung 2.20: Totale Antwortzeit (T) der M/M/m-Warteschlange verglichen mit der Ankunftsrate λ. Anzahl der Server 2

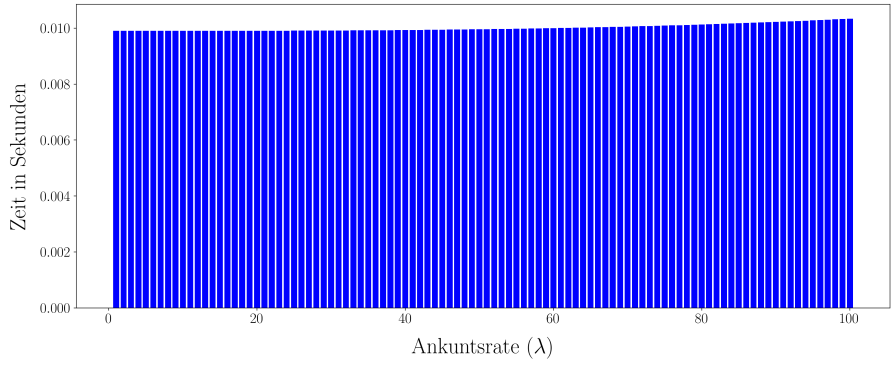


Abbildung 2.21: Totale Antwortzeit (T) der M/M/m-Warteschlange verglichen mit der Ankunftsrate λ. Anzahl der Server 3

Die Wahrscheinlichkeit, dass sich das System bei hoher Ankunftsrate (λ) im Leerlauf befindet, steigt stark und stabilisiert sich je höher die Anzahl der Server ist (siehe Abbildung 2.22. Daher kann angenommen werden, dass das System schnell genug arbeitet, sodass eine höhere Serverzahl überflüssig ist. Besonders im Hinblick auf den Aspekt des Energieverbrauchs ist hier Optimierungsbedarf, da auch Server im Leerlauf Kosten erzeugen. Das bedeutet, dass Skalierung die totale Antwortzeit des Systems zwar deutlich verbessern kann, Server im Leerlauf grundsätzlich aber als negativ betrachtet werden sollten.

Abbildung 2.22: Wahrscheinlichkeit für einen Server im Leerlauf

Im praktischen Teil werden diese Thesen geprüft und durch Lastverteilung und dynamische Skalierung weiter optimiert, sodass auch bei zwei oder mehr Servern die Antwortzeit des Systems optimiert werden kann und Leerlauf-Zeiten vermieden werden.

3 Evaluation

3.1 Systembeschreibung

Die Versuche werden auf einem einzelnen System/Rechner durchgeführt. Der Rechner ist ausgestattet mit einer AMD Ryzen 5 5600G CPU und 32 GB Arbeitsspeicher. Die Applikation – implementiert nach der Definition einer ungeeigneten Applikation - wird in einem Docker-Container gestartet und ist dadurch von den weiteren Replikationen der Applikation unabhängig. Der Lastverteiler wird als alleinstehende Applikation realisiert welcher je nach Konfiguration verschiedene Lastverteilungsalogrithmen verwendet. Das Lastverteiler ermöglicht es gezielt die Applikation durch das Starten oder Stoppen von Containern zu skalieren. Die Nutzer:inen werden durch eine Applikation repräsentiert, welche in einem bestimmten Zeitintervall HTTP-Anfragen an den Lastverteiler sendet, um den Dienst der ungeeigneten Applikation zu erreichen. Jede Anfrage symbolisiert eine:n Nutzer:in durch einen eigenen Thread im Programmablauf. Eine Anfrage gilt als abgeschlossen, sobald sie beantwortet wurde. Hierbei werden in jedem Schritt Zeitstempel aufgezeichnet. Desweiteren soll während der Laufzeit die Effizienz des Systems aufgezeichnet werden.

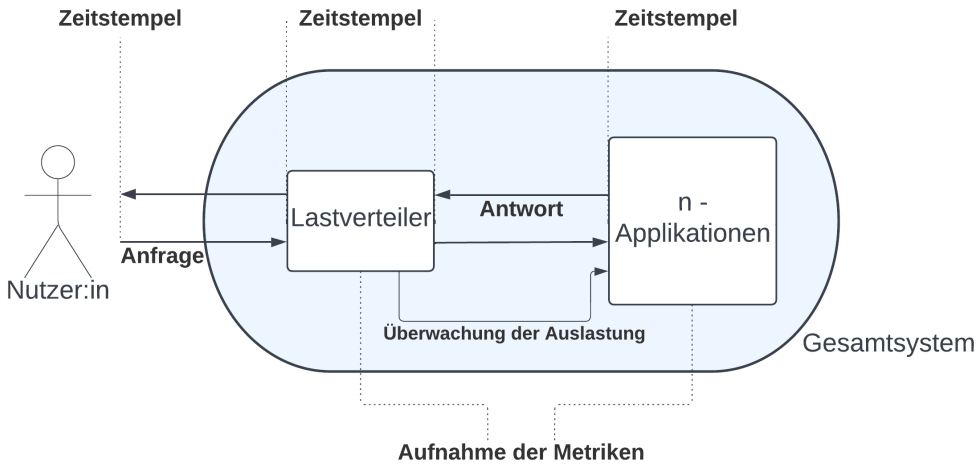


Abbildung 3.1: Skizzierung des Systems. Eigene Darstellung

Im Kontext des Quellcodes und in der Implementierung wird Englisch verwendet.

3.1.1 Applikation

Die Ziel-Applikation wurde nach der Definition einer ungeeigneten Applikation umgesetzt. Zwei Endpunkt sind über den Lastverteiler erreichbar. Der erste Endpunkt der Applikation („GET /calc“) errechnet bei jeder eintreffenden Anfrage den Wert von Pi durch die Leibniz-Formel mit 10.000.000 berechneten Brüchen (siehe Definition Leibniz-Reihe). Wenn die Rechnung beendet ist, sendet die Applikation an den:die Sender:in eine Antwort zurück (HTTP-Status 200). Mit der Antwort werden im HTTP-Body die Eintritts- und Austrittszeitpunkte versendet. Um Parallelisierung zu vermeiden, ist der Endpunkt durch einen Semaphor nur für eine Anfrage gleichzeitig verwendbar. Jede weitere Anfrage wartet, bis die vorherige Anfrage beendet ist. Der zweite Endpunkt („GET /getUtilization“) ermöglicht es dem Lastverteiler, die aktuelle Auslastung der Applikation abzufragen. Die Applikation antwortet mit dem Durchschnittswert der prozentualen Auslastung der CPU und des Arbeitsspeichers (HTTP-Status 200). Jede Instanz der Applikation wird in einem Docker-Container gestartet. (Implementierung siehe A.1.2)

3.1.2 Lastverteiler

Der Lastverteiler ist ein einfacher Reverse-Proxy, der alle eintreffenden Anfragen auf eine Sammlung an angeschlossenen Instanzen der ungeeigneten Applikation aus Abschnitt 2.3 (UA) verteilt. Der Lastverteiler überwacht während der Laufzeit den Zustand des Systems und stößt gegebenenfalls Skalierungsentscheidungen an.

Zu Beginn der Laufzeit startet der Lastverteiler die in der Konfigurationsdatei festgelegte Anzahl an Instanzen der UA. Der Port und Container-ID werden hierbei gespeichert. Über den Endpunkt „GET /calc“ leitet der Lastverteiler die eingehenden Anfragen an die UA weiter. Beim Eintreffen einer Anfrage wird ein Zeitstempel aufgezeichnet und die nächste Instanz der UA ausgewählt. Eine Instanz kann auf vier Wege gewählt werden. Ist für den Lastverteiler kein bekannter Algorithmus konfiguriert, wird die erste Instanz aus der Sammlung gewählt. Wenn „random“ eingestellt ist, wird eine zufällige Instanz aus der List gewählt. Durch „RR“ (siehe Unterabschnitt 2.4.4) wird eine Instanz anhand des Round-Robin-Algorithmuses und durch „AWRR“ mittel des erweiterten gewichtetet Round Robin (siehe Unterabschnitt 2.4.6) gewählt. Der „AWRR“ verwendet den Endpunkt „GET /getUtilization“ der UA, um die Auswahlwahrscheinlichkeit einer Instanz zu bestimmen.

Wenn für den Lastverteiler Skalierung konfiguriert wurde, wird in einem fünf Sekunden Zyklus über den Endpunkt „GET /getUtilization“ der UA die aktuelle Auslastung der jeweiligen Instanzen abgefragt. Wenn die gesamte Auslastung des Systems einen konfigurierten Schwellenwert unter- oder überschreitet, werden neue Instanzen der UA gestartet oder gestoppt. Der Schwellenwert setzt sich zusammen aus der Summe der Ressourcen-Auslastung (CPU und Arbeitsspeicher) und der Anzahl der aktuellen User, die die jeweilige Instanz beanspruchen, geteilt durch 100 mal zwei. Hierbei wird eine prozentuale Gewichtung der Anzahl der Nutzer:innen errechnet. Die Applikation stoppt durch die Skalierungsentscheidung nie alle Container, um die UA stets erreichbar zu halten. Für das Starten und Stoppen der Instanzen in den entsprechenden Docker-Containern wird der Docker SDK für Python verwendet.

Während der Laufzeit des Lastverteilers zeichnet der Lastverteiler den CPU- und Arbeitsspeicherverbrauch über die eigene Prozess-ID auf.

(Implementierung siehe Anhang A.1.1)

3.1.3 Client-Simulationsanwendung

Um die entsprechende Anzahl der Anfrage gemäß der Versuchsbeschreibung zu simulieren, wird die Client-Simulationsanwendung (CSA) verwendet. Hierbei wird für die konfigurierte Anzahl der Anfragen je ein Thread erstellt, welcher als hypothetische:r Nutzer:in fungiert. Die Threads werden in zufällig gewählten Zeitintervallen gestartet. Die Ober- und Untergrenzen dieser Zeitintervalle sind durch die Konfigurationsdatei einstellbar. Dieser Wert bestimmt die Ankunftsrate (λ) des Systems. Jeder Thread führt eine asynchrone Funktion, aus welche die HTTP-Anfrage („GET /calc“) an den Lastverteiler sendet. Hierbei werden vor dem Versenden und nach dem Erhalt einer Antwort Zeitstempel aufgezeichnet. Sobald der Lastverteiler eine Antwort erhält, wird der Nachrichteninhalt mit den Zeitstempeln der einzelnen Stationen der Anfrage in eine Datei geschrieben und diese gesichert.

(Implementierung siehe Anhang A.1.4)

3.1.4 Konfigurationsdatei

In der Konfigurationsdatei werden folgende Parameter festgelegt:

- numberOfInstances – die Anzahl der erstellten Container der UA
- numberOfClients – die Anzahl der Nutzer:innen, Anfragen
- portLoadbalancer – der Port, über den der Loadbalancer erreichbar ist
- path – der Endpunkt der UA
- imageName – der Name des Docker-Images der UA
- dockerPort – der interne Port des Docker-Containers
- loadbalancingAlgorithm – der Lastverteilungsalgorithmus welcher verwendet werden soll
- scalingMechanism – Skalierungsmechanismus (wenn „none“ dann keine Skalierung, sonst immer)
- filePath – Pfad des Ordners, in dem die Messungen gespeichert werden
- lowerUtilizationLimit - unterer Schwellenwert für Skalierungentscheidungen

- upperUtilizationLimit – oberer Schwellenwert für Skalierungentscheidungen
- maxContainer – Anzahl der maximalen Container
- timeBetweenClientsMSMax – maximaler Abstand für den Start eines Client-Threads
- timeBetweenClientsMSMin – minimaler Abstand für den Start eines Client-Threads

3.1.5 Sequenzdiagramm

In Abbildung 3.2 wird der Ablauf einer Anfrage eines Threads der CSA vereinfacht dargestellt.

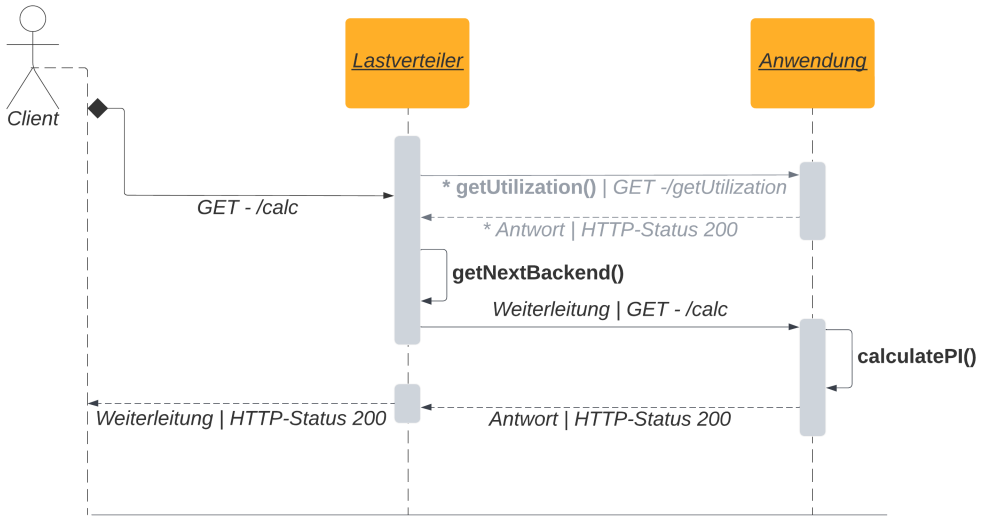


Abbildung 3.2: Sequenz Diagramm einer Anfrage. Eigene Darstellung

* Wenn Skalierung aktiv oder „AWRR“ verwendet wird.

3.1.6 Aufzeichnung der Metriken / Verwendete Techniken

Für die Aufzeichnung der Metriken der UA zeichnet ein Bash-Skript in einem zwei Sekunden Zyklus die prozentuale CPU-Auslastung und Arbeitsspeicherauslastung des Containers auf. Hierfür wird die Docker-API verwendet, welche durch den Befehl „docker stats“

alle Container bezogenen Informationen ausgibt. Diese Ausgabe wird formatiert und mit einem Zeitstempel in eine Ausgabedatei geschrieben. Das Bash-Skript wird mit Start des Lastverteilers gestartet. Die Metriken des Lastverteilers werden durch psutil abgefragt. psutil (process and system utilities) ist eine plattformübergreifende Bibliothek zum Abrufen von Informationen über laufende Prozesse und Systemauslastung (CPU, Speicher, Festplatten, Netzwerk, Sensoren) in Python. Die UA verwendet ebenfalls psutil für den Endpunkt „GET /getUtilization“. Hierbei werden nicht die Docker Container-Statistiken verwendet, da der Aufruf von „docker stats“ einen größeren Aufwand darstellt als die Abfrage über psutil. Jedoch ist der „docker stats“ Aufruf außerhalb der UA durch ein Bash-Skript am präzisesten, da hierdurch die gesamten Metriken der Container-Laufzeit zurückgegeben werden.

3.2 Versuchsaufbau

Das Zeitintervall für die Ankunftsrate ist in allen Versuch zufällig gewählt. Um Vergleichbarkeit der Versuche zu gewährleisten, ist der Bereich zwischen 100 Millisekunden und 500 Millisekunden konfiguriert. Zwischen diesen Werten wird eine pseudo-zufällige Zahl gewählt, welche die Zeit bestimmt, bis ein neuer Client-Thread der CSA gestartet wird.

3.2.1 Versuch 1

In Versuch 1 wird eine Instanz an den Lastverteiler angeschlossen. Die CSA erstellt 100 Client-Threads welche jeweils eine Anfrage an den Lastverteiler senden. Es werden weder Lastverteilungsalgorithmen noch Skalierungsmechanismen verwendet.

Die vorgenommenen Konfigurationen für Versuch 1:

numberOfInstances	1
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	none
scalingMechanism	none
filePath	Messungen/Versuch1/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

3.2.2 Versuch 2

Für Versuch 2 wird jeweils einer der in Abschnitt 2.4 erörterten Lastverteilungsalgorithmen verwendet. Im ersten Durchlauf wird Round-Robin („RR“, Versuch 2.1) und im zweiten Durchlauf der erweiterte gewichtete Round-Robin („AWRR“, Verusch 2.2) angewendet. Die Versuche werden nacheinander ausgeführt und sollen den Unterschied zwischen RR und AWRR verdeutlichen. An den Lastverteiler werden zehn Instanzen der UA angeschlossen. Die CSA erstellt 100 Client-Threads, welche jeweils eine Anfrage an den Lastverteiler senden. Es wird kein Skalierungsmechanismus verwendet.

Die vorgenommenen Konfigurationen für Versuch 2.1:

numberOfInstances	10
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	RR
scalingMechanism	none
filePath	Messungen/Versuch2_1/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

Die vorgenommenen Konfigurationen für Versuch 2.2:

numberOfInstances	10
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	AWRR
scalingMechanism	none
filePath	Messungen/Versuch2_2/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

3.2.3 Versuch 3

Versuch 3 soll einen Grenzfall verdeutlichen. Im Theorieteil dieser Arbeit wurde folgende Vermutung aufgestellt: Wenn die Anzahl der Instanzen gleich der Anzahl der Anfragen ist, wird die Antwortzeit einer jeden Anfrage durch die Servicezeit bestimmt. Mit Versuch 3 soll diese Vermutung überprüft werden. Die Versuche sind wie in Versuch 2 aufgeteilt in der Verwendung der Lastverteilungsalgorithmen RR und AWRR. An den Lastverteiler werden 100 Instanzen der UA angeschlossen. Die CSA erstellt 100 Client-Threads, welche jeweils eine Anfrage an den Lastverteiler sendet. Es wird kein Skalierungsmechanismus verwendet.

Die vorgenommenen Konfigurationen für Versuch 3.1:

numberOfInstances	100
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	RR
scalingMechanism	none
filePath	Messungen/Versuch3_1/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

Die vorgenommenen Konfigurationen für Versuch 3.2:

numberOfInstances	100
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	AWRR
scalingMechanism	none
filePath	Messungen/Versuch3_2/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

3.2.4 Versuch 4

Durch Versuch 4 soll die Effektivität von Skalierungen verdeutlicht werden. Skalierungsentscheidungen werden anhand der konfigurierten Schwellenwerte getroffen. Die Schwellenwerte sind ein zusammengesetzter prozentualer Wert des Gesamtsystems aus CPU-, Arbeitsspeicherauslastung und der Anzahl der Nuter:innen, die die Instanz beanspruchen. Die gewählten Schwellenwerte sind 15% als unterer Schwellenwert (ein Container ist im Leerlauf bei einer Auslastung von ca. 1%) und als oberer Schwellenwert 30% Auslastung des Gesamtsystems. Die Werte sind durch wiederholtes Beobachten des Skalierungsverhaltens und Metriken gewählt worden. Um zu verhindern, dass bei einer hohen Auslastung über einen langen Zeitraum eine unbegrenzte Anzahl an neuen Containern gestartet wird, wird eine maximal Anzahl an erstellbaren Containern konfiguriert. An den Lastverteiler ist zu Beginn eine Instanz der UA angeschlossen. Die CSA erstellt 100 Client-Threads welche jeweils eine Anfrage an den Lastverteiler senden. Es wird kein Lastverteilungsalgorithmus verwendet. Die Wahl der Instanzen ist zufällig.

Die vorgenommenen Konfigurationen für Versuch 4:

numberOfInstances	1
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	random
scalingMechanism	true
filePath	Messungen/Versuch4/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

3.2.5 Versuch 5

Versuch 5 ist die Kombination der Versuche 4, 2.1 und 2.2. Hierbei werden sowohl Lastverteilungsalgorithmen als auch Skalierung verwendet. Die Skalierung ist konfiguriert wie in Versuch 4. Die Lastverteilungsalgorithmen RR und AWRR werden analog zu 2.1 und 2.2 verwendet. Hierbei wird wieder in Versuch 5.1 und Versuch 5.2 unterschieden. Diese Versuche sollen verdeutlichen, welche Einflüsse Lastverteilung und Skalierung auf das Gesamtsystem nehmen. An den Lastverteiler ist zu Beginn eine Instanz der UA angeschlossen. Die CSA erstellt 100 Client-Threads, welche jeweils eine Anfrage an den Lastverteiler senden.

Die vorgenommenen Konfigurationen für Versuch 5.1:

numberOfInstances	1
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	RR
scalingMechanism	true
filePath	Messungen/Versuch5_1/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

Die vorgenommenen Konfigurationen für Versuch 5.2:

numberOfInstances	1
numberOfClients	100
portLoadbalancer	7777
path	/calc
imageName	calc_pi
dockerPort	5000
loadbalancingAlgorithm	AWRR
scalingMechanism	true
filePath	Messungen/Versuch5_2/
lowerUtilizationLimit	0.15
upperUtilizationLimit	0.3
maxContainer	20
timeBetweenClientsMSMax	500
timeBetweenClientsMSMin	100

3.3 Erwartungen

Es wird erwartet, dass in den Versuchen ein klarer Unterschied zwischen einem optimierten System (durch Lastverteilung und Skalierung) und einem nicht hochverfügbaren System deutlich wird. Allgemein werden geringere Antwort- und Servicezeiten des optimierten Systems erwartet.

3.3.1 Versuch 1

In Versuch 1 wird erwartet, dass das System die Anfragen nacheinander abarbeitet und sich dadurch die Antwortzeit der einzelnen Anfragen aufaddiert. Somit sollte sich für jede Anfrage die Servicezeit (Zeit die für die Rechnung der UA) plus die Wartezeit, die die Anfrage in der Wartschlange des Semaphors verbringt, als gesamte Antwortzeit ergeben. Die Ergebnisse des Versuchs sollten ähnlich zu den aus den theoretischen Ergebnissen sein (siehe Unterunterabschnitt 2.7.4). In diesem Versuch wird ebenfalls davon ausgegangen, dass die totale Antwortzeit – durch die hoch konfigurierte Ankunftsrate – in einem vergleichbar hohen Bereich fallen wird. Allgemein sollte durch den Versuch verdeutlicht werden, dass ein solches System nicht in einem hochverfügbaren Kontext effizient verwendet werden kann.

3.3.2 Versuch 2

Im Vergleich zu Versuch 1 ist zu erwarten, dass sich in den Versuch 2.1 und 2.2 bereits deutliche Verbesserung ausmachen lassen. Da für diesen Versuch zehn Instanzen der UA konfiguriert sind, sollte die totale Antwortzeit stark sinken. Bei der Verwendung des AWRR sollte sich – im Vergleich zum RR – eine weitere Verbesserung der totalen Antwortzeit abzeichnen. Im Gegensatz zum RR wird für den AWRR die Last der jeweiligen Instanzen berücksichtigt und somit kann davon ausgegangen werden, dass die Wartzeiten reduziert werden und somit auch die Antwortzeiten der einzelnen Anfragen abnehmen werden. Für Versuch 2.1 und 2.2 wird gleichermaßen erwartet, dass durch die Aktivität von zehn Instanzen der UA die Effizienz negativ beeinflusst wird, da Container sich im Leerlauf befinden und Ressource blockieren.

3.3.3 Versuch 3

Für Versuch 3 wird erwartet, dass die theoretischen Vermutungen aus Unterunterabschnitt 2.7.4 bestätigt werden. Durch die Verwendung der gleichen Anzahl an Instanzen wie Anfragen soll die Wartezeit sich an null annähern und die Anwortzeit jeder Anfrage ungefähr der Servicezeit entsprechen. Im Vergleich zum AWRR sollte hier der RR bessere Ergebnisse erzielen. Die Abfrage der Auslastung der UA durch den AWRR führt zu Wartezeiten in der Bestimmung der nächsten Instanz. Die Wahl der Instanz des RR ist dagegen schneller. Dennoch sollten sich die Antwortzeiten aller Anfragen für RR und alle Anfragen für AWRR in einem gleichen Bereich befinden – Servicezeit plus die Zeit für die Wahl einer Instanz. Diese Versuche sollen verdeutlichen, dass viele Instanzen zu einer starken Beanspruchung der Ressourcen des Systems führen. Auch wenn die total Antwortzeit sinkt, wird in diesen Versuchen erwartet, dass die Implementierung kein effizientes System widerspiegelt.

3.3.4 Versuch 4

Die totale Antwort des Versuchs 4 sollte im Vergleich zu den Versuchen 2.1 und 2.2 weiter sinken. Die dynamische Anpassung der Anzahl der Instanzen der UA hat direkten Einfluss auf die Antwort- und Wartezeit einer jeden Anfrage. Sobald Skalierungsentscheidungen getroffen werden, wird erwartet, dass sich das System „entspannt“, da neue Ressourcen verfügbar sind. Auch wenn die Lastverteilung durch eine pseudo-zufällige Wahl gesteuert wird, wird erwartet, dass die Skalierung bessere Ergebnisse in der Auslastung der Ressourcen und Antwortzeiten erreicht. Die theoretischen Vermutungen aus Unterunterabschnitt 2.7.4, dass Skalierung die Effizienz des Systems verbessert, sollen auch hier bestätigt werden.

3.3.5 Versuch 5

Durch die Kombination der Versuche 2 und 4 wird in Versuch 5 eine optimale Lösung erwartet. Im Vergleich zu Versuch 4 soll sich durch die Verwendung von Lastverteilungsalgorithmen eine Verbesserung abzeichnen. Da hier sowohl die Last auf die angeschlossenen Ressourcen verteilt wird als auch während der Laufzeit neue Instanzen gestartet werden, sollte die totale Antwortzeit sich der Antwortzeit aus Versuch 3 annähern. In Versuch 3 gehen wir von einer minimalen Antwortzeit durch Sättigung des Systems aus. In Versuch

5 soll dies angenähert werden durch Skalierung und Lastverteilung. Diametral zu Versuch 3 ist jedoch die Auslastung des Gesamtsystem wesentlich geringer, da die Instanzen nur nach Bedarf gestartet werden und im Leerlauf befindliche Instanzen gestoppt werden. Dadurch sollte das System effizienter agieren als das System aus Versuch 3, aber trotzdem bessere Antwortzeiten als in den Versuchen 2 oder 4 erzielen.

3.4 Auswertung der Versuche

Die aufgezeichneten Metriken der Versuche wurden mit der Python Bibliothek „pandas“ eingelesen. Die Graphen wurden durch „pyplot“ erstellt. In allen Graphen wird Englisch verwendet.

3.4.1 Versuch 1

Abbildung 3.3 zeigt den Zeitaufwand, der pro Anfrage benötig wird, bis der oder die Nutzer:in eine Antwort erhalten hat. Hier ist deutlich zu erkennen, dass jede weitere Anfrage, die auf eine Anfrage folgt zu einer Summierung der Antwortzeiten führt. Abbildung 3.4 bestätigt, dass die Anfragen nacheinander abgearbeitet werden, da die Servicezeiten sich stark ähneln.

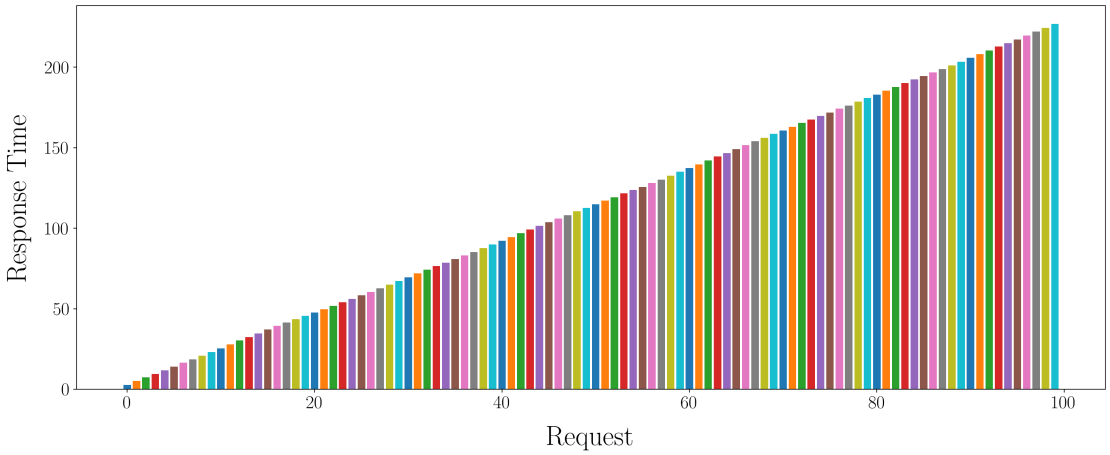


Abbildung 3.3: Versuch 1. Antwortzeiten in Sekunden - jeder Balken entspricht einer Anfrage

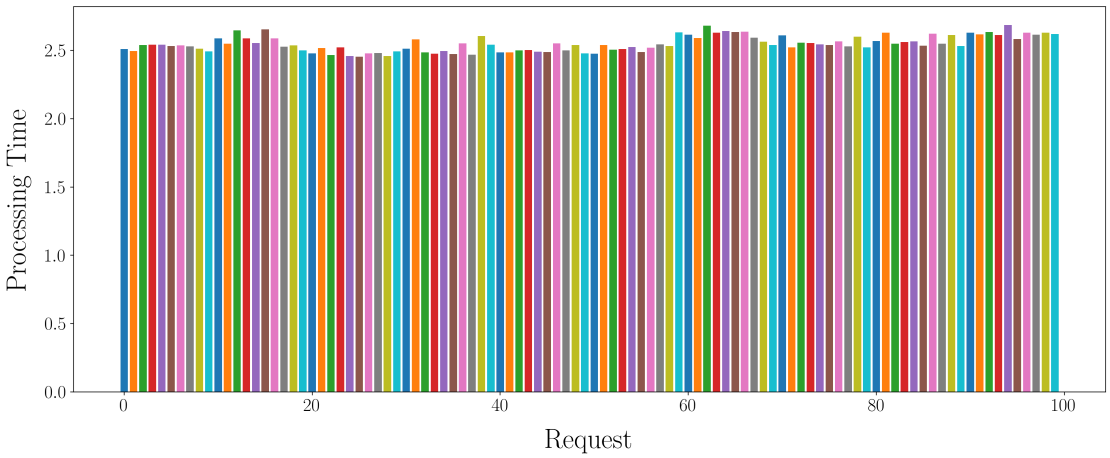


Abbildung 3.4: Versuch 1. Servicezeit in Sekunden - jeder Balken entspricht einer Anfrage

In Abbildung 3.5 ist die Auslastung der Container während der gesamten Laufzeit zu sehen. Es wird deutlich, dass der Container unter konstanter Last steht und die CPU-Auslastung erst sinkt nachdem keine Anfragen mehr eintreffen. Die verstrichene Zeit ab dem Versenden der ersten Anfrage bis zur Antwort auf die letzte Anfrage (totale Antwortzeit) beträgt 255,18 Sekunden.

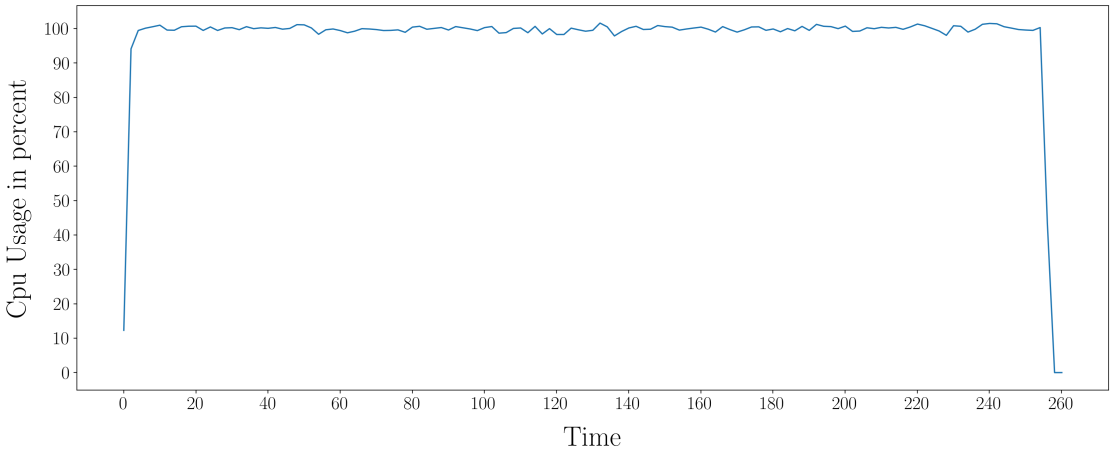


Abbildung 3.5: Versuch 1. CPU-Auslastung des Docker-Containers - Zeit in Sekunden

3.4.2 Versuch 2

In Abbildung 3.6 und Abbildung 3.7 wird die prozentuale CPU-Auslastung des Lastverteilers bei Verwendung von RR oder AWRR dargestellt. Der rechnerische Aufwand für AWRR ist zeitweise mehr als doppelt so hoch wie für den RR. Die totale Antwortzeit des RR beträgt 39,79 Sekunden. In Versuch 2.2 mit AWRR hingegen verstreichen 45,57 Sekunden.

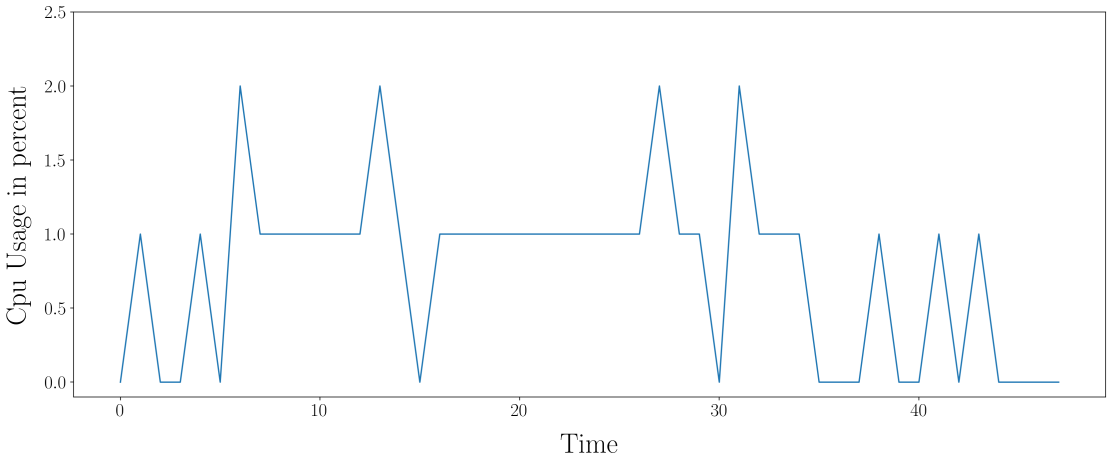


Abbildung 3.6: Versuch 2.1. CPU-Auslastung des Lastverteilers - Zeit in Sekunden

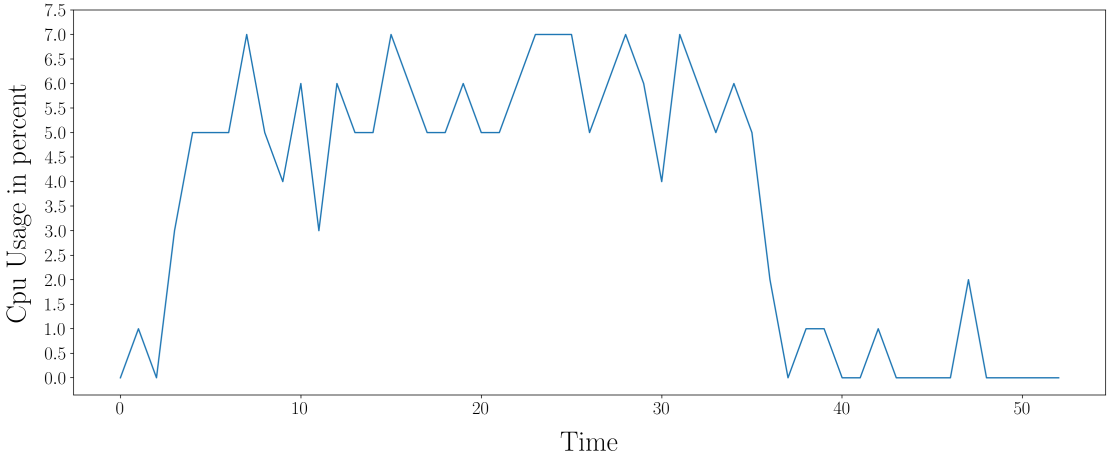


Abbildung 3.7: Versuch 2.2. CPU-Auslastung des Lastverteilers - Zeit in Sekunden

Abbildung 3.8 und Abbildung 3.9 stellen die Antwortzeit pro Anfrage dar. Erkennbar wird, dass die Antwortzeit in Versuch 2.1 mit RR im Verhältnis zum AWRR stetig

steigt. Die Antwortzeiten für AWRR verteilen sich wesentlich mehr, sind jedoch höher. Das lässt sich durch den größeren Aufwand der Wahl der Instanz begründen. Die längste Antwortzeit verzeichnet ebenfalls der AWRR.

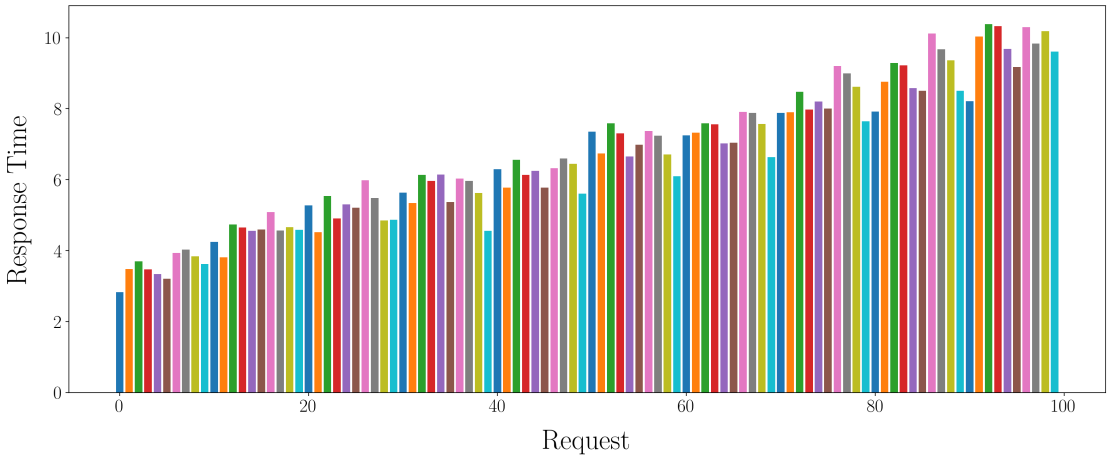


Abbildung 3.8: Versuch 2.1. Antwortzeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

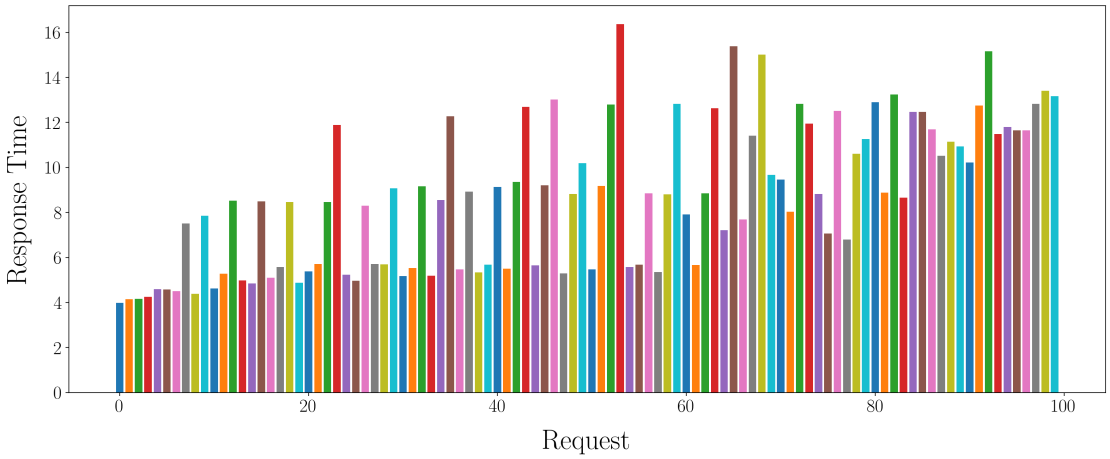


Abbildung 3.9: Versuch 2.2. Antwortzeiten mit erweiterten gewichteten Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

Durch Abbildung 3.10 Und Abbildung 3.11 wird jedoch deutlich, dass die Auslastung der Container durch AWRR wesentlich besser verteilt ist. Somit ist das System mit AWRR trotz schlechterer Antwortzei, effizienter als das System mit RR. Der Abfall der CPU-Auslastung in beiden Abbildung zeigt das Stoppen der einzelnen Container. Die Metriken

werden aufgezeichnet, bis alle Container gestoppt sind und der Lastverteiler beendet ist (plus fünf Sekunden Leerlauf).

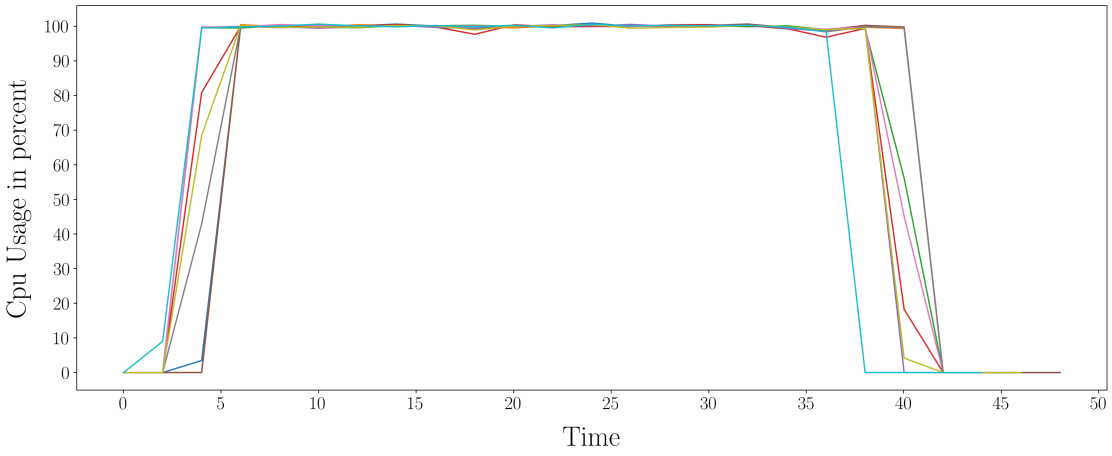


Abbildung 3.10: Versuch 2.1. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

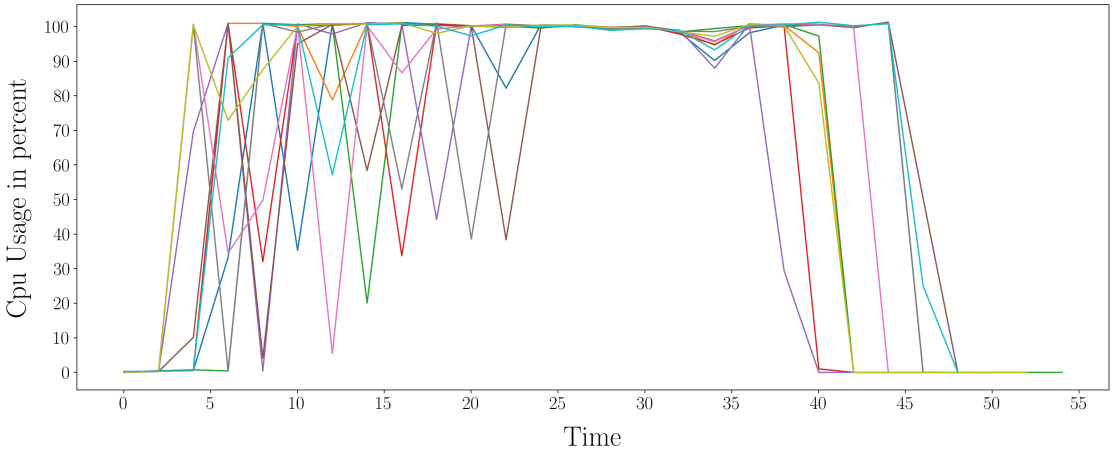


Abbildung 3.11: Versuch 2.2. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

3.4.3 Versuch 3

Abbildung 3.12 Und Abbildung 3.13 veranschaulichen die CPU-Auslastung der Container in der die UA ausgeführt wird. Bei RR ist zu erkennen, dass im Zeitraum von Sekunde 20 bis 35 die Auslastung deutlich sinkt. AWRR ähnelt anfänglich RR, jedoch ist im weiteren

Verlauf der Ausführung zu erkennen, dass ein Container die Hauptlast erhält. Ebenso ist zu erkennen, dass viele Container keine Berücksichtigung finden und sich im Leerlauf befinden. Die Abflachung ab Sekunde 40 bei beiden Varianten entsteht durch die Zeit, die benötigt wird, um jeden einzelnen Container zu stoppen.

Die totale Antwortzeit von Versuch 3.1 beträgt 36,5 Sekunden. Das System in Versuch 3.2 benötigt 130,16 Sekunden um jede Anfrage zu beantworten.

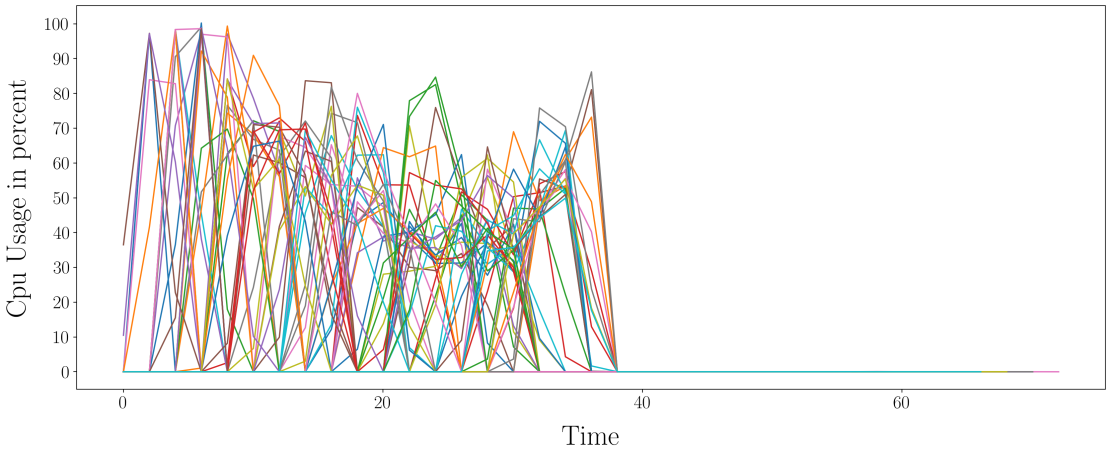


Abbildung 3.12: Versuch 3.1. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

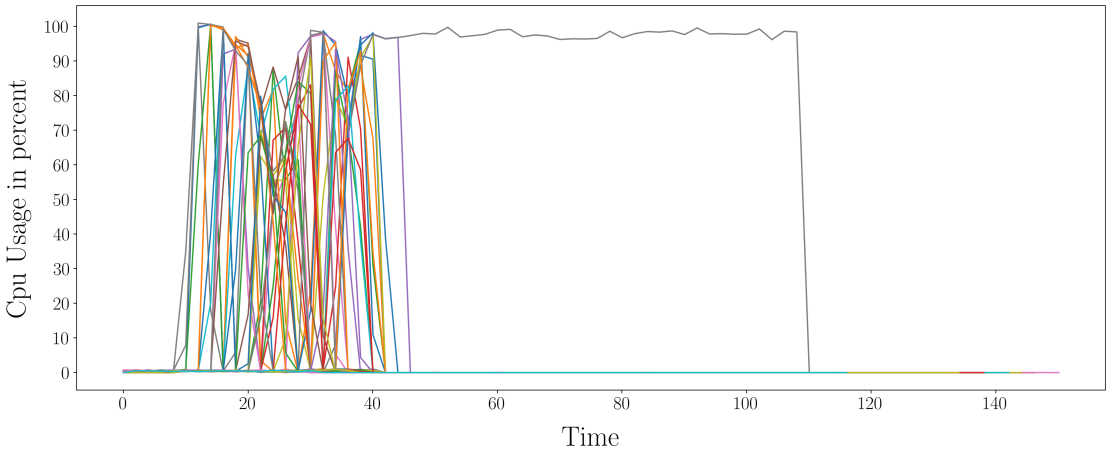


Abbildung 3.13: Versuch 3.2. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

Betrachten wir RR in Abbildung 3.14 Und Abbildung 3.15. Hier wird deutlich erkennbar, dass sich die Servicezeit zur Antwortzeit minimal unterscheidet. Die Steigung des Graphen lässt sich durch die Auslastung des Gesamtsystems begründen. Das System hat die Ressource zu 100% ausgeschöpft und die Berechnungen verlängern sich. Sobald die ersten Anfragen abgefertigt wurde, flacht die Kurve wieder ab.

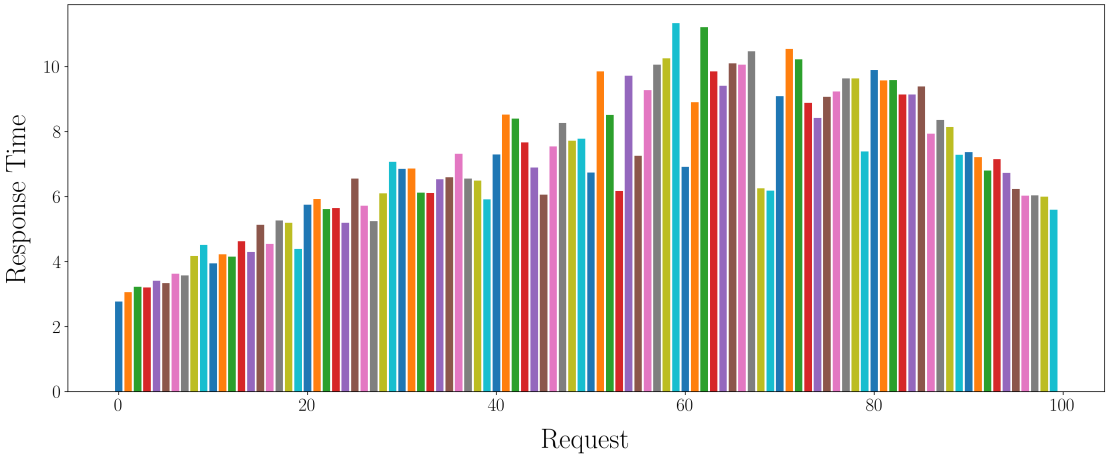


Abbildung 3.14: Versuch 3.1. Antwortzeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

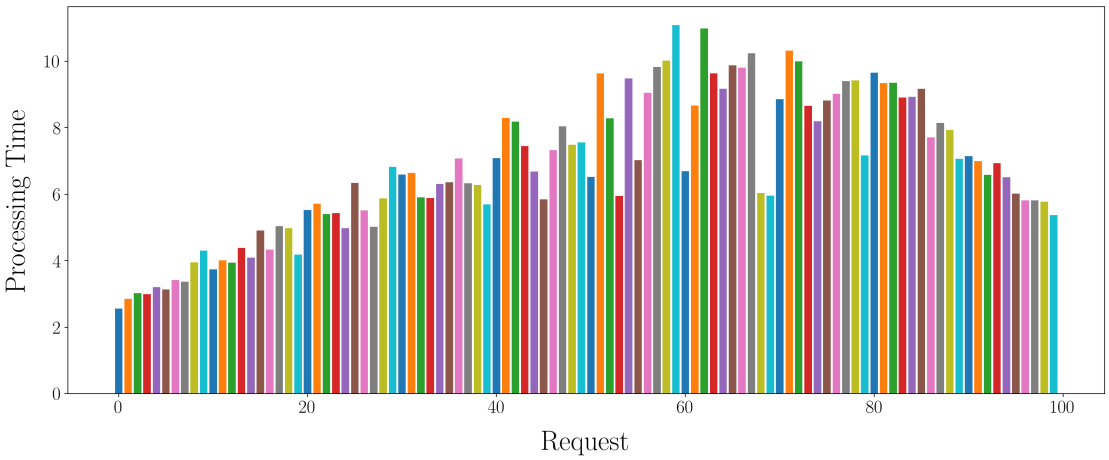


Abbildung 3.15: Versuch 3.1. Servicezeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

In Abbildung 3.16 Und Abbildung 3.17 von Versuch 3.2 ist der Unterschied der Servicezeit und Antwortzeit deutlich zu erkennen. Da viele Container nicht berücksichtigt werden

und gegen Ende der Laufzeit ein Container die Hauptlast erhält, entsteht ab 63 ein ähnliches Verhalten wie in Versuch 1. Dies gilt sowohl für die Servicezeit als auch für die Antwortzeit analog.

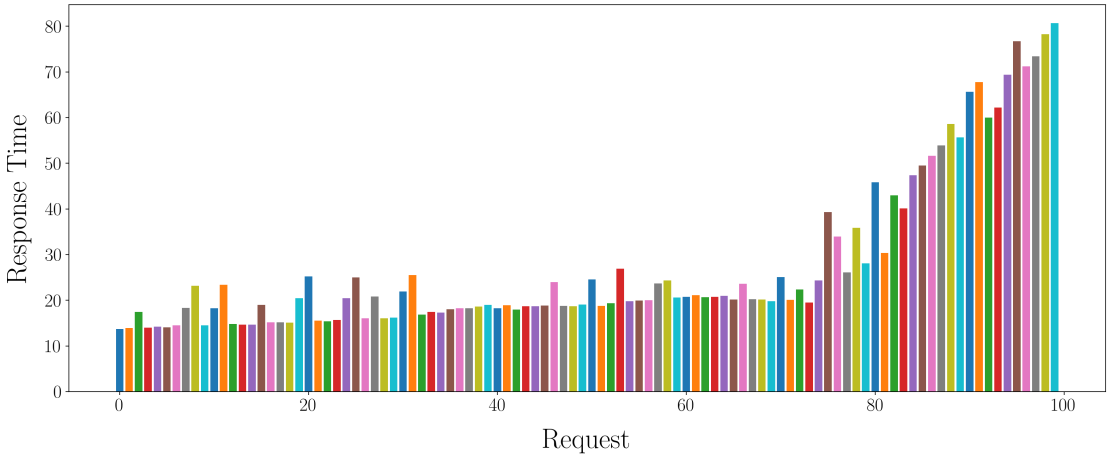


Abbildung 3.16: Versuch 3.2. Antwortzeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

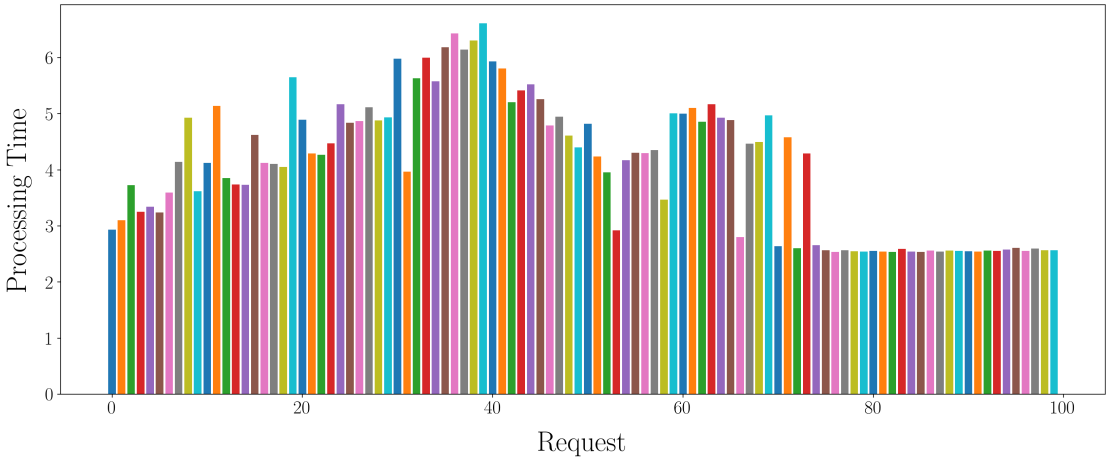


Abbildung 3.17: Versuch 3.2. Servicezeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

Aus Abbildung 3.18 lässt sich die CPU-Auslastung des Lastverteilers unter Verwendung des AWRR ablesen. Hier wird der enorm höhere Rechenaufwand (wie in Versuch 2.2) deutlich.

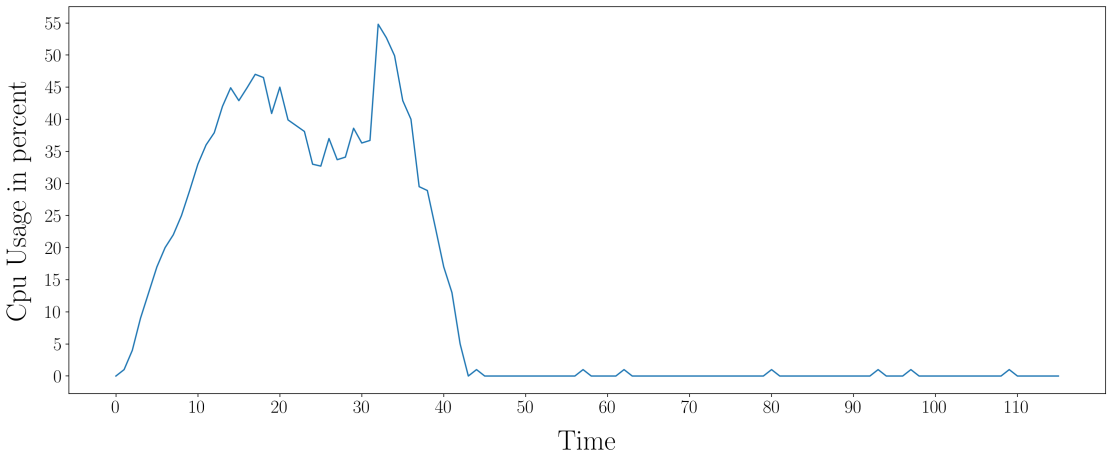


Abbildung 3.18: Versuch 3.2. CPU-Auslastung des Lastverteilers - Zeit in Sekunden

3.4.4 Versuch 4

In Abbildung 3.20 wird ein ähnliches Ergebnis wie in Versuch 1 erreicht. Die Servicezeiten sind höher als die aus Versuch 1, sind jedoch in einem fast konstanten Bereich. Betrachten wir Abbildung 3.19 kann die Entlastung des Systems bei jeder Skalierungsentscheidung abgelesen werden. Mehrere Kurven lassen sich erkennen. Jeder Einbruch steht für den Start einer neuen Instanz und die Steigung für die Ansammlung von Anfragen auf der jeweiligen Instanz.

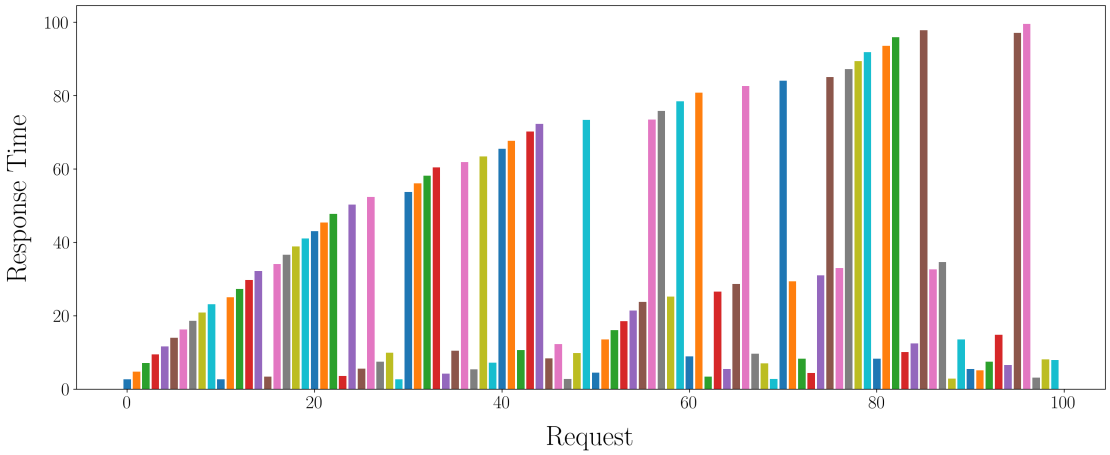


Abbildung 3.19: Versuch 4. Antwortzeiten in Sekunden - jeder Balken entspricht einer Anfrage

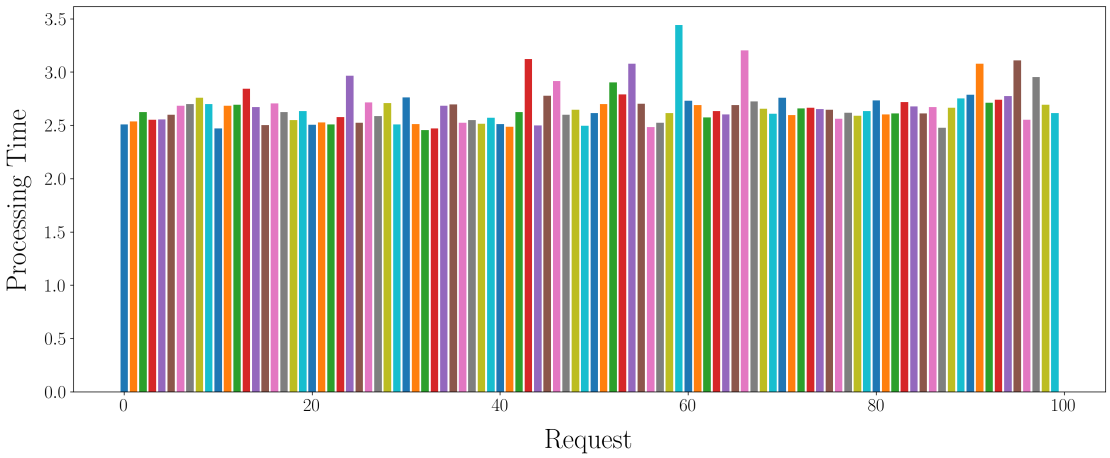


Abbildung 3.20: Versuch 4. Servicezeiten in Sekunden - jeder Balken entspricht einer Anfrage

In Abbildung 3.21 kann man die Ineffizienz der Wahl der Instanzen ablesen. Da die Instanzen pseudozufällig gewählt werden, sind hier keine klaren Strukturen erkennbar. Ein Teil der Container sind im Leerlauf, ein Teil wird permanent beansprucht.

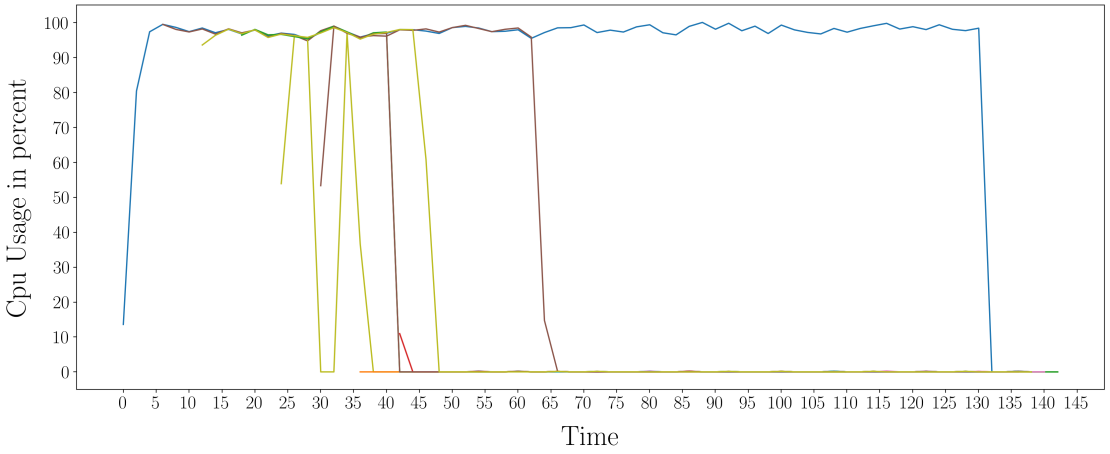


Abbildung 3.21: Versuch 4. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

Trotz der zyklischen Abfrage der Container-Auslastung und Skalierungsentscheidungen sind wenige Differenzen zwischen den CPU-Auslastungen aus Versuch 4 (Abbildung 3.22) und Versuch 2.1 zu erkennen. Die Auslastung der CPU ist insignifikant höher. Lediglich

die Schwankungen sind ein Indiz für den zyklischen Ablauf der Skalierungsentscheidungen.

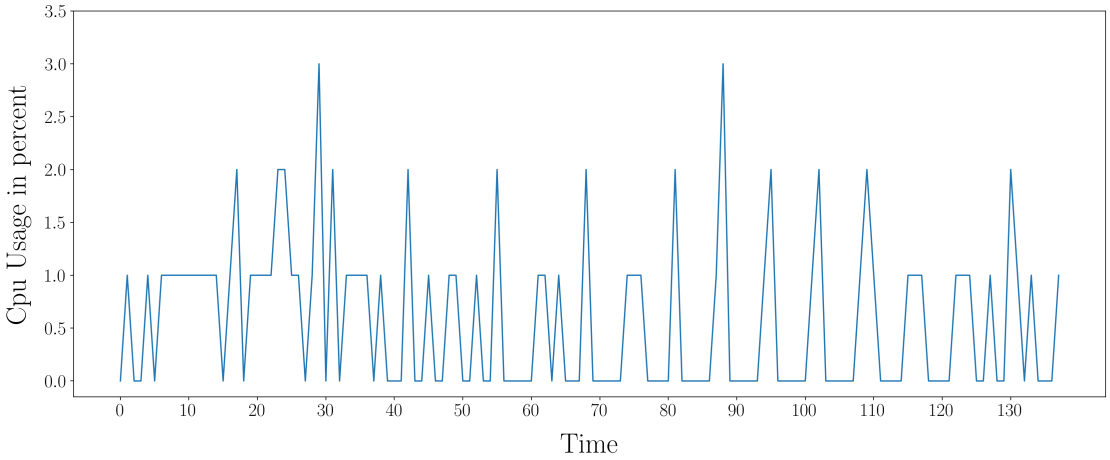


Abbildung 3.22: Versuch 4. CPU-Auslastung des Lastverteilers - Zeit in Sekunden

Die totale Antwortzeit von Versuch 4 beträgt 130,58 Sekunden.

3.4.5 Versuch 5

In Versuch 5.1 findet die Lastverteilung durch RR statt. In Abbildung 3.23 sind die CPU-Auslastungen der einzelnen Instanzen verbildlicht. Durch RR sehen wir eine klarere Verteilung der Auslastung. Die einzelnen Container werden regelmäßig ausgelastet und wieder entlastet.

In Abbildung 3.24 sind ähnlich wie in Versuch 4 einzelne Kurven in den Antwortzeiten der Anfragen erkennbar. Sie sind erneut erklärbar durch die einzelnen Skalierungsentscheidungen und den damit verbundenen Start einer weiteren Instanz. Wieder wird das System entlastet und eine neue Anfrage kann schneller beantwortet werden. Auch in Abbildung 3.25 ist ähnlich wie in Versuch 4 zu erkennen, dass die Servicezeiten sich in einem nur leicht schwankenden Bereich befinden.

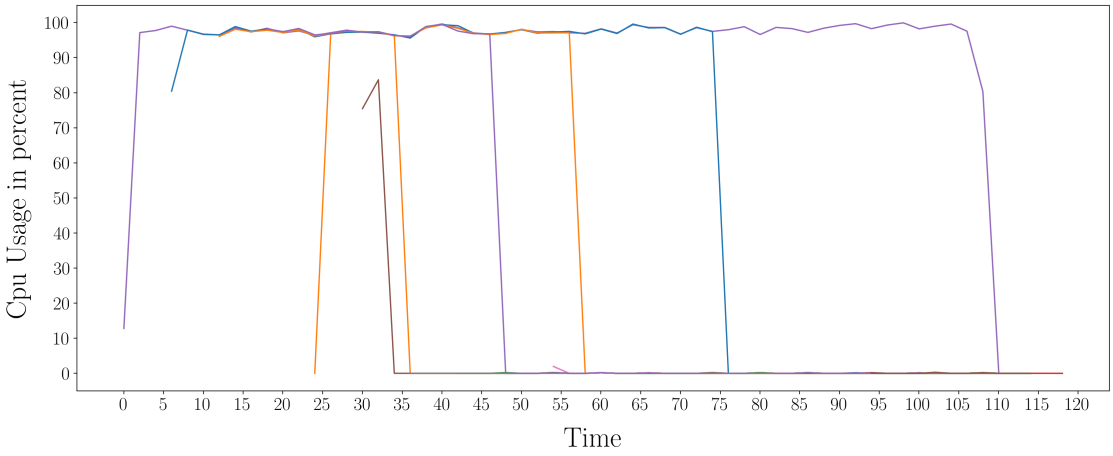


Abbildung 3.23: Versuch 5.1. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

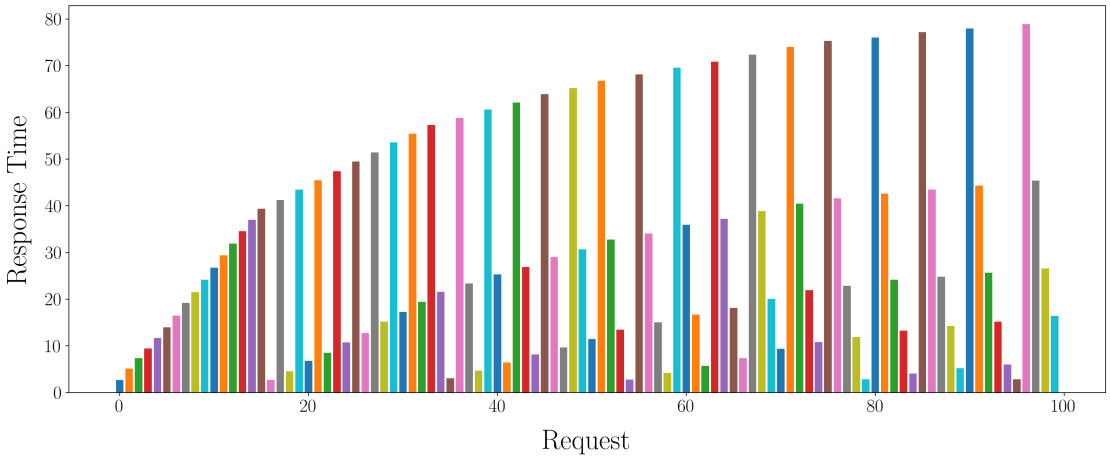


Abbildung 3.24: Versuch 5.1. Antwortzeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

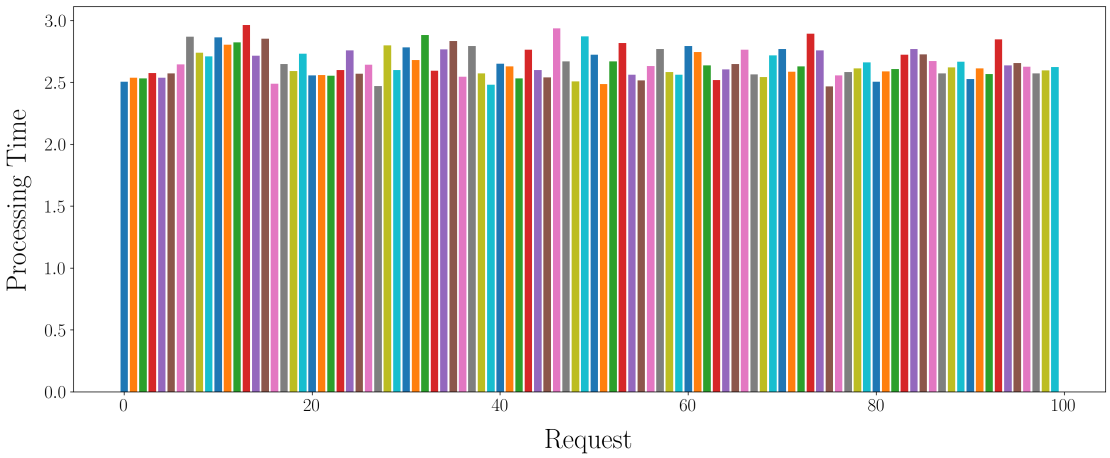


Abbildung 3.25: Versuch 5.1. Servicezeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

Betrachten wir Versuch 5.2 und AWRR, sehen wir in Abbildung 3.26 eine klare Veränderung in der Auslastung der einzelnen Instanzen. Sobald eine neue Instanz verfügbar ist, wird diese durch den AWRR direkt angesprochen und erreicht bereits nach kurzer Laufzeit hohe Auslastung. Allgemein müssen dadurch weniger Skalierungsentscheidungen getroffen werden, da die Container besser ausgenutzt werden können. Anomalien treten hier durch die Abfrage der Docker-API auf (siehe Sekunde fünf). Der Container erhält bereits Last noch bevor eine weiter Abfragezyklus der Docker-API vollendet ist. Bei Sekunde 35 wird deutlich, dass noch Verbesserungspotential für die Implementation des Algorithmus besteht, da der Container beendet wird bevor dieser Last erhalten hat.

Noch deutlicher wird dieser Zusammenhang durch Abbildung 3.27. Sobald eine neue Instanz gestartet wurde, verlagert sich die Last auf diese und die Anfragen sammeln sich an dieser Instanz an. Durch die konstante Belastung und Entlastung sehen wir in Abbildung 3.28 wieder Servicezeiten, die denen aus Versuch 1 ähneln.

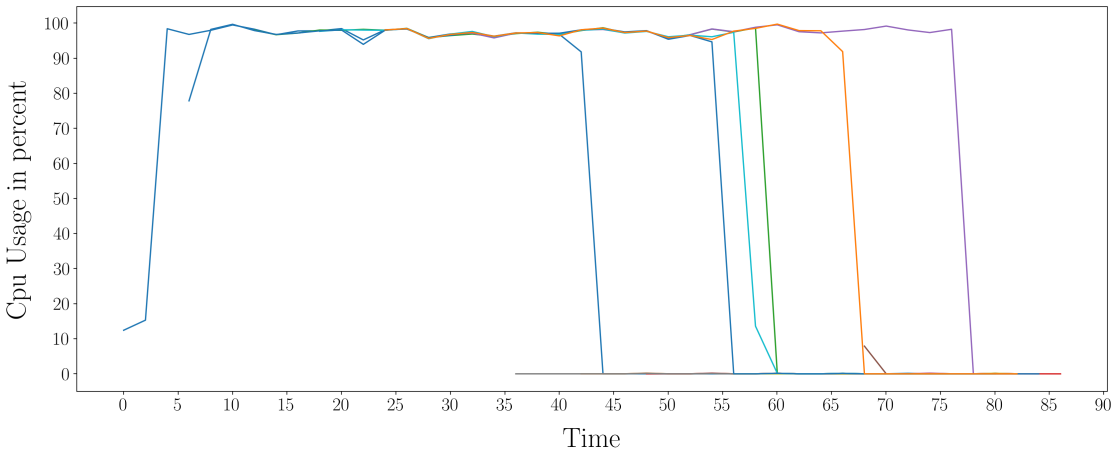


Abbildung 3.26: Versuch 5.2. CPU-Auslastung der einzelnen Docker-Container - Zeit in Sekunden

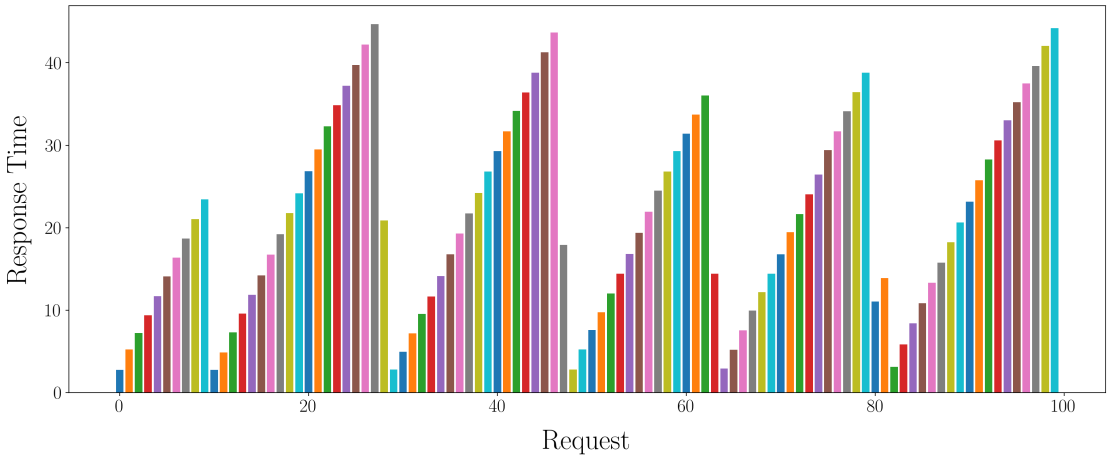


Abbildung 3.27: Versuch 5.2. Antwortzeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

Die totale Antwortzeit von Versuch 5.1 beträgt 107,66 Sekunden. Das System in Versuch 5.2 benötigt 76,17 Sekunden, um jede Anfrage zu beantworten.

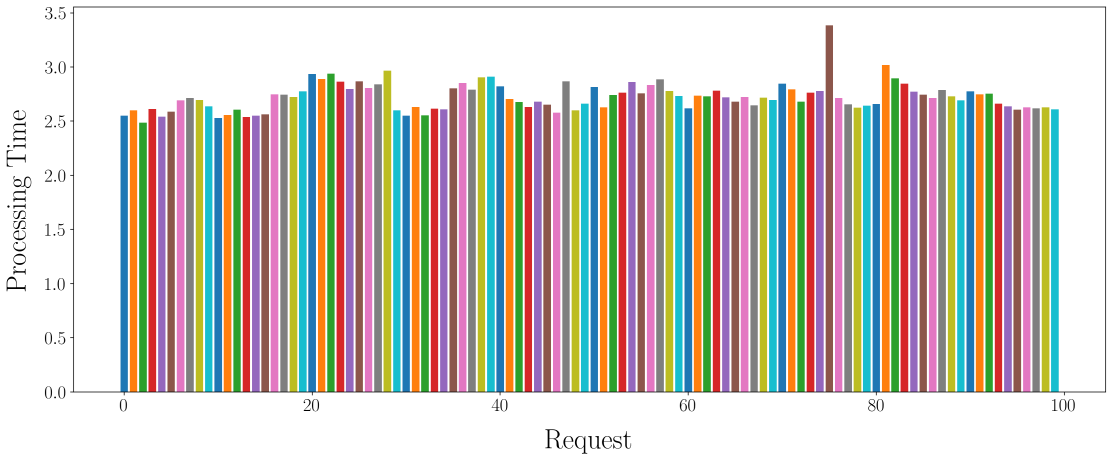


Abbildung 3.28: Versuch 5.2. Servicezeiten mit Round-Robin in Sekunden - jeder Balken entspricht einer Anfrage

3.5 Bewertung der Versuch/Analyse

Versuch 1 stellt ein Negativbeispiel dar. Zwar sind die Servicezeiten konstant, jedoch steigen die Antwortzeiten linear. Anhand dieser Erkenntnis lassen sich bereits eine erste Optimierungsmöglichkeit erkennen. Die Wartezeiten der Anfragen summieren sich auf, da lediglich eine Instanz gestartet wurde. Im Umkehrschluss bedeutet das, je mehr Instanzen gestartet werden, desto geringer wird die totale Antwortzeit.

Anhand Versuch 2 kann diese Erkenntnis genauer analysiert werden. In Versuch 2 wird deutlich, wie Lastverteilungsalgorithmen Last auf eine Sammlung an angeschlossenen Instanzen verteilt. Für Versuch 2.1 und 2.2 wurden jeweils 10 Instanzen zu Beginn der Laufzeit gestartet. Durch diese Erhöhung der Anzahl der Instanzen hat sich die totale Antwortzeit in Versuch 2.1 und 2.2 fast um den Faktor 10 verkleinert. Somit ist die erste Optimierungsmöglichkeit bestätigt.

Versuch 2 macht jedoch ebenso deutlich, dass Leerlaufzeiten zu vermeiden sind, da das System Ressourcen verbraucht, ohne diese nutzen zu können. Versuch 3 - als Grenzfall - bestätigt diese Vermutung. In Versuch 3 ist deutlich zu erkennen, dass das System die Anfragen schnell beantworten kann. Jedoch wird auch das Maximum der Ressourcen benötigt, um dies zu erreichen. Hierdurch ist gezeigt, dass für Effizienz im Gesamtsystem Instanzen nur auf Bedarf gestartet werden sollten. Somit zeigt dies eine weitere Optimierungsmöglichkeit auf.

Versuch 2 und 3 vergleichen zudem zwei verschiedene Lastverteilungsalgorithmen. Durch die Verwendung von RR zur Lastverteilung wurden die geringste totale Antwortzeit erreicht (siehe Abbildung 3.29).

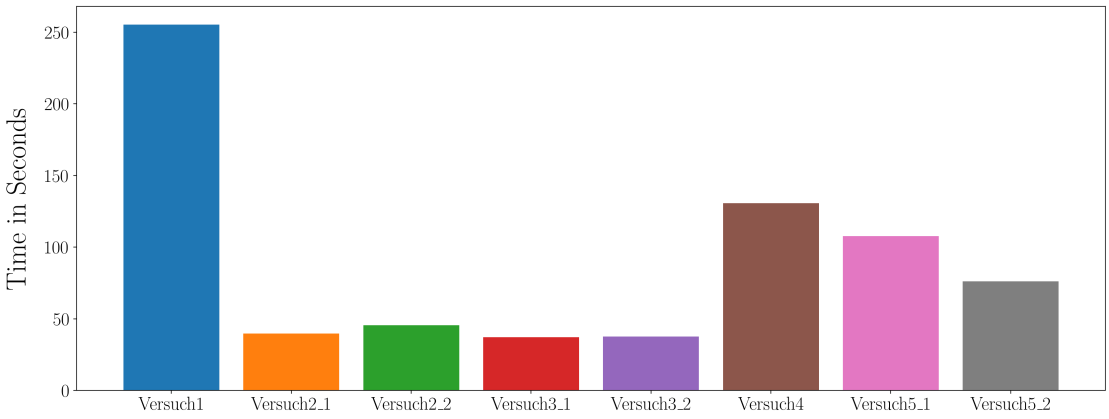


Abbildung 3.29: Total Antwortzeit aller Versuche

Der AWRR welcher laut der Erwartungen schnellere Antwortzeiten erzielen sollte, ist wider Erwarten langsamer. Das lässt sich in Versuch 3 dadurch erklären, dass RR in diesem System einen klaren Vorteil hat. AWRR errechnet bei jeder eintreffenden Anfrage die Wahrscheinlichkeit, dass eine Instanz gewählt wird und muss hierzu die Auslastung jeder Instanz abfragen. Dabei führt die Menge der Anfragen dazu, dass der AWRR die Auslastung nicht effizient interpretieren kann. RR erzielt hier durch sein weniger komplexeres Vorgehen bessere Ergebnisse.

Auch Versuch 2 wird das deutlich. Zwar kann hier ebenso das Potential von AWRR interpretiert werden. Die Verzögerung durch das Abfragen der Auslastung hat hier allerdings wieder eine höhere totale Antwortzeit zur Folge.

Dass AWRR jedoch auch eine hohe Effektivität ermöglichen kann, wird in Versuch 3 klar (siehe Abbildung 3.30).

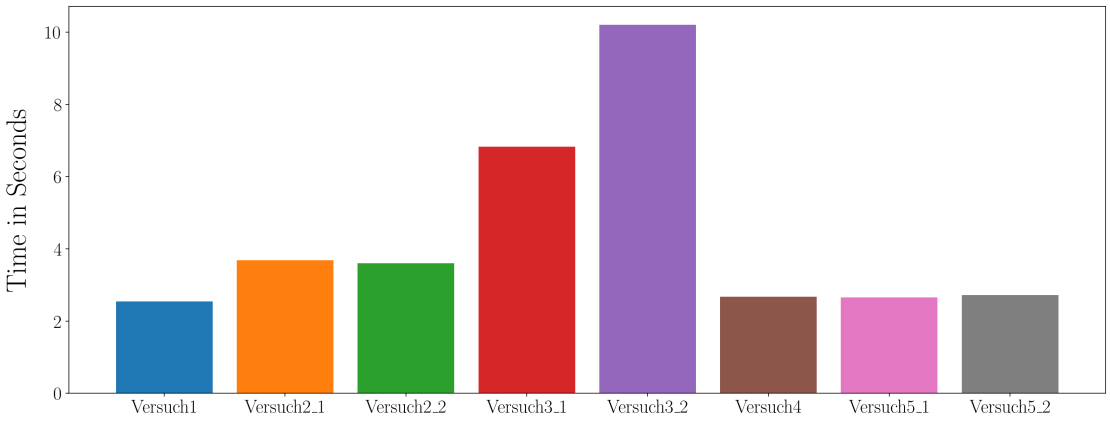


Abbildung 3.30: Durchschnittliche Servicezeiten aller Versuch

Die durchschnittliche Servicezeit der Instanzen ist für RR klar am höchsten. Diese Anomalie lässt durch die Gesamtauslastung des Systems begründen und die dadurch resultierende langsamere Berechnung. Jedoch ist hier erkennbar (siehe 3.11), dass dynamische Lastverteilung durch AWRR die Ressourcen des Systems schonen kann. Aus diesen Erkenntnissen lässt sich für Optimierung ableiten, dass Lastverteilung nah am Zielsystem realisiert werden sollte, um entsprechend der Auslastung der angeschlossenen Instanzen eine Wahl zu treffen.

In Versuch 4 wird ein Skalierungsmechanismus genauer betrachtet. Aus der Erkenntnis, dass mehr Instanzen eine geringere totale Antwortzeit zur Folge hat, wird deutlich, wie Skalierung die Effizienz und Leistung des Gesamtsystems steigern kann. Durch die Anpassung der Anzahl der Instanzen anhand der Auslastung aller Instanzen wird eine nahezu konstante Servicezeit erreicht.

Die totale Antwortzeit in Versuch 5 ist nicht wie erwartet die niedrigste. Das lässt sich dadurch erklären, dass bei Versuch 5 bereits einige Sekunden verstrichen sind, bis zehn Instanzen der UA gestartet wurden. In Versuch 2 hingegen sind von Beginn der Laufzeit an zehn Instanzen verfügbar. Diese Differenz im Versuchsaufbau kann die insgesamt höhere Antwortzeit erklären. Trotzdem lässt sich anhand von Versuch 5 erkennen, wie die Kombination aus Lastverteilungsalgorithmen (besonders durch AWRR) und Skalierungsmechanismen die Effizienz und Leistung des Systems (gleichermaßen) steigern kann.

3.6 Bezug auf Theoretischen Ansätze

Durch die theoretische Analyse eines vergleichbaren Systems wurden in Abschnitt 2.7 grundlegende Konzepte und Einflüsse der Ankunftsrate und Rechenleistung aufgezeigt. Diese sollte im praktischen Teil genauer geprüft und gegebenenfalls bestätigt werden.

Auf Grundlage der Ergebnisse aus Abschnitt 2.5 wurde angenommen, dass durch Skalierung der Energieverbrauch eines Systems kontrolliert werden kann. Im Vergleich zu einem System, in dem sich Nodes mit hoher Wahrscheinlichket im Leerlauf befinden, kann durch Skalierung die Effizienz und damit der Energieverbrauch verbessert werden. Anhand der Versuche aus Abschnitt 3.4 wurde deutlich aufgezeigt, wie das Skalierungsverhalten des Systems die Leistung und Auslastung eines Systems optimiert. Als Grundlage für diese Hypothese wurde angenommen, dass wenn die Serverzahl gleich der Anzahl an Anfragen/Nutzer:innen ist, die Antwortzeit gleich der Servicezeit ist. Durch Versuch 3.1 und 3.2 wurde diese Annahme bestätigt. Zwar wurde in Versuch 3.1 eine deutliche negative Beeinflussung der Antwortzeit durch die hohe Belastung des Gesamtsystem deutlich. Dennoch wurde durch diesen Versuch beschrieben, wie die Antwortzeit maximal optimiert werden kann durch die Sättigung der Anfragen mittels der Serverzahl. In diesem Versuch wurde ebenso deutlich, wie Leerläufe einen negativen Einfluss auf das System nehmen. Die aufgestellte Aussage aus Unterabschnitt 2.7.4, dass Leerläufe vermieden werden sollten wurde durch Versuch 3 verdeutlicht. Versuch 2.2 und 5.2 bestätigen diese Annahme. Diese Versuche erreichten durch den erweiterten gewichteten Round-Robin Lastverteilungsalgorithmus die Minimierung der Leerlaufzeiten der angeschlossenen Instanzen.

Die theoretischen Annahmen aus Unterabschnitt 2.7.4 wurden durch Implementierung des Modells und Simulationen in Kapitel 3 bestätigt.

3.7 Verbesserungen

Verbesserungsbedarf besteht in der Implementation des AWRR Lastverteilungsalgorithmuses. Um die Gewichte der Instanzen zu aktualisieren, wird bei jeder einzelnen Instanz die momentane Auslastung erfragt. Hierbei entstehen zwangsläufig Wartzeiten und der Algorithmus kann das gewünschte Ergebnis nicht zuverlässig erzielen. Deutlich wird dies in Versuch 4 und 5. In diesen Versuchen wird die Auslastung aller Instanzen in regelmäßigen Abständen und nicht für jede Anfrage erfragt (siehe A.1.1). Dadurch entstehen

weniger Anfragen und das System und der Algorithmus arbeiten vermeintlich effizienter. Jedoch sind dadurch nicht die aktuellen Metriken verfügbar. Der AWRR würde am effizientesten arbeiten, wenn die Aktualität der Metriken gewährleistet ist und hierdurch wenig bis keine Verzögerung in der Bearbeitung der Abfrage entstehen.

Des Weiteren wurde ein isoliertes System betrachtet. Weitere Forschung sollte das System in tatsächliche Nodes aufzuteilen, um somit auch die Netzwerk Latenzen berücksichtigen zu könne. So könnte ein Ergebnis erzielt werden, das realitätsnäher ist. In dem Kontext dieser Arbeit genügte es ein isoliertes System zu verwenden, um den Fokus auf Algorithmen zu lenken. Für weitere Arbeiten in diesem Themenbereich ist jedoch ein System, welches möglichst nah an einem produktiven System ist, von Relevanz.

In weiteren Versuchen könnte Eine Applikation betrachtet werden, welche regelmäßige Ausfälle implementiert. Hierdurch kann eine weitere Eigenschaft der nach Kapitel 2.3.6 definierten ungeeigneten Applikation analysiert werden. Unter anderem kann mithilfe weiterer Versuche die Elastizität des Systems durch verschiedene Startzeiten der Container vertiefend aufgezeigt werden.

Weiterführend können die Warteschlangen-Modelle in der Theorie detaillierter analysiert werden. Besonderer Augenmerk sollte auf die Analyse weiterer Modelle, wie G/G/1, G/G/m und M/G/m gelegt werden. Hier besteht großes Potential, da diese Modelle bis jetzt nur angenähert werden konnten.

4 Fazit

4.1 Zusammenfassung

In dieser Arbeit wurde gezeigt, wie Antwortzeiten, Servicezeiten und die Leistung einer Cloud-Applikation optimiert werden können. Die Lastverteilungsalgorithmen Round-Robin und erweiterter gewichteter Round-Robin wurden erörtert und in den Versuchen aus Kapitel 3 auf die Probe gestellt. Die Resultate, die durch Lastverteilung erzielt werden konnten, bestätigen die theoretischen Annahmen aus Unterabschnitt 2.7.4 und zeigen, wie Lastverteilung die Effizienz des Systems steigern kann. Durch weitere Versuche mit einem Skalierungsmechanismus wurde deutlich, wie die dynamische Zahl von Instanzen - angepasst an die Last des Systems - für die Optimierung von Antwortzeiten sorgen kann. Die Kombination dieser beiden Techniken erzielt die besten Ergebnisse und zeigt, wie Cloud-Applikationen bereitgestellt werden können. Auch Applikationen, die für diesen Einsatz nicht ausgelegt sind, können von diesen Techniken profitieren.

4.2 Beantwortung der Forschungsfrage

Ziel der Arbeit war es, zu veranschaulichen, welche Techniken und Algorithmen verwendet werden können, um Skalierung und Lastverteilung zu optimieren und Applikationen für den hochverfügbaren Einsatz zu realisieren. Anhand der theoretischen Konzepte und der Bestätigung dieser durch die Versuche aus Kapitel 3 wird deutlich, dass eine Applikation, welche nach den Kriterien einer ungeeigneten Applikation aus Unterabschnitt 2.3.6 definiert wurde, optimiert werden kann. Eine Applikation, welche nur wenige oder eine:n Nutzer:in zur gleichen Zeit abarbeiten kann, kann durch Skalierung verbessert werden. Dadurch können mehrere Nutzer:innen gleichzeitig abgearbeitet werden. Durch die Isolierung der Nodes ist auch gewährleistet, dass die Servicezeiten nicht durch Wartezeiten in den Node steigen. Die Elastizität des Systems ist hierbei ein großer Faktor. Wenn der

Aufwand, eine neue Instanz zu starten dem Nutzen überwiegt ist auch Skalierung nicht zwingend von Vorteil. Auch durch Lastverteilung kann nach den Ergebnissen dieser Arbeit Optimierung erzielt werden. Wenn die Last auf die verfügbaren Nodes verteilt wird, kann die Servicezeit weiter verbessert werden, da die Wahrscheinlichkeit, dass eine Node gewählt wird anhand der Last und Nutzerzahl angepasst wird.

Durch die Versuche sei ein Vorschlag aufgezeigt, wie die Bereitstellung von Cloud-Applikationen im besten Fall realisiert werden sollte. Intelligente Lastverteilung (dynamisch bezogen auf die Last oder andere Faktoren) und Skalierung dieser Applikation sind notwendig, um hohe Verfügbarkeit zu garantieren und nicht die User-Experience durch lange Wartezeiten negativ zu beeinflussen. Allgemein ist durch diese Techniken ein höherer Quality-of-Service erreichbar.

Literaturverzeichnis

[1] Bashari Rad, Babak ; Bhatti, Harrison ; Ahmadi, Mohammad: An Introduction to Docker and Analysis of its Performance. In: *IJCSNS International Journal of Computer Science and Network Security* 173 (2017), 03, S. 8

[2] Boettiger, Carl: An Introduction to Docker for Reproducible Research. In: *SIGOPS Oper. Syst. Rev.* 49 (2015), jan, Nr. 1, S. 71–79. – URL https://doi.org/10.1145/2723872.2723882. – ISSN 0163-5980

[3] Chen, Hong ; Yao, David D.: *Fundamentals of Queuing Networks: Performance, asymptotics, and Optimization*. Springer, 2011

[4] Das, Rajarshi ; Tesauro, Gerald ; Walsh, William: Model-Based and Model-Free Approaches to Autonomic Resource Allocation. (2005), 01

[5] Divers, a10networks.com: *Layer 4 vs layer 7 load balancing: Glossary*. Sep 2022. – URL https://www.a10networks.com/glossary/how-do-layer-4-and-layer-7-load-balancing-differ/. – Besucht: 23.10.2022

[6] Divers, aws.amazon.com. – URL https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-scheduled-scaling.html. – Besucht: 23.10.2022

[7] Divers, cloudflare.com: *ReverseProxyFlow*. – URL https://www.cloudflare.com/de-de/learning/cdn/glossary/reverse-proxy/. – Besucht: 25.10.2022

[8] Divers, cloudzero.com: *6 actionable ways to improve your cloud efficiency*. – URL https://www.cloudzero.com/blog/cloud-efficiency. – Besucht: 24.10.2022

[9] Divers, cloudzero.com: *Horizontal Vs. Vertical Scaling: How Do They Compare?*. – URL https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling. – Besucht: 03.11.2022

[10] Divers, eventhelix.com. – URL https://www.eventhelix.com/congestion-control/m-m-1/#Poisson%20Arrivals. – Besucht: 23.10.2022

[11] Divers, google.com: *Anhand von Vorhersagen Skalieren. Compute Engine-Dokumentation.* – URL https://cloud.google.com/compute/docs/autoscaler/predictive-autoscaling?hl=de. – Besucht: 23.10.2022

[12] Divers, nginx.com: *Nginx HTTP Server.* Packt Publishing Limited, 2015

[13] Divers, nginx.com: *Using round robin for simple load balancing.* Jan 2022. – URL https://www.nginx.com/resources/glossary/round-robin-load-balancing/. – Besucht: 23.10.2022

[14] Divers, solarwindssoftware.com: *What is load balancing? types, configurations, and best tools.* Jul 2020. – URL https://www.dnsstuff.com/what-is-server-load-balancing. – Besucht: 23.10.2022

[15] DIVERS, wikipedia.org: *Jackson Netzwerk.* – URL https://en.wikipedia.org/wiki/Jackson_network. – Besucht: 25.10.2022

[16] Divers, wikipedia.org: *Poisson-Verteilung.* Aug 2022. – URL https://de.wikipedia.org/wiki/Poisson-Verteilung. – Besucht: 23.10.2022

[17] Divers, wikipedia.org: *State–action–reward–state–action.* Jul 2022. – URL https://en.wikipedia.org/wiki/State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action. – Besucht: 23.10.2022

[18] Doherty, Rina A. ; Sorenson, Paul: Keeping Users in the Flow: Mapping System Responsiveness with User Experience. In: *Procedia Manufacturing* 3 (2015), S. 4384–4391. – URL https://www.sciencedirect.com/science/article/pii/S2351978915004370. – 6th International Conference on Applied Human Factors and Ergonomics (AHFE 2015) and the Affiliated Conferences, AHFE 2015. – ISSN 2351-9789

[19] Fielding, R. ; Nottingham, M. ; Reschke, J.: HTTP Semantics / RFC Editor. RFC Editor, June 2022 (97). – STD. https://www.rfc-editor.org/rfc/rfc9110.html. – ISSN 2070-1721

[20] Fielding, Roy T. ; Gettys, James ; Mogul, Jeffrey C. ; Nielsen, Henrik F. ; Masinter, Larry ; Leach, Paul J. ; Berners-Lee, Tim: Hypertext Transfer

Protocol – HTTP/1.1 / RFC Editor. RFC Editor, June 1999 (2616). – RFC. https://www.rfc-editor.org/rfc/rfc2616. – ISSN 2070-1721

[21] Freed, Ned ; Borenstein, Nathaniel S.: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies / RFC Editor. RFC Editor, November 1996 (2045). – RFC. – URL http://www.rfc-editor.org/rfc/rfc2045.txt. http://www.rfc-editor.org/rfc/rfc2045.txt. – ISSN 2070-1721

[22] Gandhi, Anshul ; Harchol-Balter, Mor ; Raghunathan, Ram ; Kozuch, Michael A.: Distributed, Robust Auto-Scaling Policies for Power Management in Compute Intensive Server Farms. In: *2011 Sixth Open Cirrus Summit*, 2011, S. 1–5

[23] Gandhi, Rohan ; Liu, Hongqiang H. ; Hu, Y. C. ; Lu, Guohan ; Padhye, Jitendra ; Yuan, Lihua ; Zhang, Ming: Duet: Cloud Scale Load Balancing with Hardware and Software. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. New York, NY, USA : Association for Computing Machinery, 2014 (SIGCOMM '14), S. 27–38. – URL https://doi.org/10.1145/2619239.2626317. – ISBN 9781450328364

[24] Gesvindr, David ; Buhnova, Barbora: Performance Challenges, Current Bad Practices, and Hints in PaaS Cloud Application Design. In: *SIGMETRICS Perform. Eval. Rev.* 43 (2016), feb, Nr. 4, S. 3–12. – URL https://doi.org/10.1145/2897356.2897358. – ISSN 0163-5999

[25] Gorton, Ian: *Essential Software Architecture (2. ed.)*. 01 2011. – ISBN 978-3-642-19175-6

[26] Heinze, Thomas ; Pappalardo, Valerio ; Jerzak, Zbigniew ; Fetzer, Christof: Auto-scaling techniques for elastic data stream processing. In: *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014, S. 296–302

[27] Huang, Dr. Y.: *Lecture CS - Basic Queueing Theory*. Jan 2003

[28] Jabbari, Ramtin ; Ali, Nauman bin ; Petersen, Kai ; Tanveer, Binish: What is DevOps? A Systematic Mapping Study on Definitions and Practices. In: *Proceedings of the Scientific Workshop Proceedings of XP2016*. New York, NY, USA : Association for Computing Machinery, 2016 (XP '16 Workshops), S. 0. – URL https://doi.org/10.1145/2962695.2962707. – ISBN 9781450341349

[29] Phulare, Arvind: *DockerSystem.* – URL https://www.edureka.co/blog/docker-architecture/. – Besucht: 25.10.2022

[30] Pishro-Nik, Hossein: *Introduction to probability, statistics, and Random Processes.* Kappa Research, 2014

[31] Rana, Omer: The Costs of Cloud Migration. In: *IEEE Cloud Computing* 1 (2014), Nr. 1, S. 62–65

[32] Save Energy, Alliance to ; Incorporated, ICF ; Incorporated, ERG ; Agency, U.S. Environmental P. ; Brown, Richard E. ; Brown, Richard ; Masanet, Eric ; Nordman, Bruce ; Tschudi, Bill ; Shehabi, Arman ; Stanley, John ; Koomey, Jonathan ; Sartor, Dale ; Chan, Peter ; Loper, Joe ; Capana, Steve ; Hedman, Bruce ; Duff, Rebecca ; Haines, Evan ; Sass, Danielle ; Fanara, Andrew: Report to Congress on Server and Data Center Energy Efficiency: Public Law 109-431. (2007), 8. – URL https://www.osti.gov/biblio/929723

[33] Shang, Zhihao ; Chen, Wenbo ; Ma, Qiang ; Wu, Bin: Design and implementation of server cluster dynamic load balancing based on OpenFlow. In: *2013 International Joint Conference on Awareness Science and Technology & Ubi-Media Computing (iCAST 2013 & UMEDIA 2013)*, 2013, S. 691–697

[34] Sohaib Ajmal, Muhammad ; Iqbal, Zeshan ; Zeeshan Khan, Farrukh ; Bilal, Muhammad ; Majid Mehmood, Raja: Cost-based Energy Efficient Scheduling Technique for Dynamic Voltage and Frequency Scaling System in cloud computing. In: *Sustainable Energy Technologies and Assessments* 45 (2021), S. 101210. – URL https://www.sciencedirect.com/science/article/pii/S2213138821002204. – ISSN 2213-1388

[35] Standardization, Switzerland. International Organization for: ISO/IEC 7498-1:1994, Information technology— Open Systems Interconnection— Basic Reference Model: The Basic Model — Part 1 / International Organization for Standardization. International Organization for Standardization, November 1994. – ISO Standard. – URL https://www.iso.org/obp/ui/en/#iso:std:iso-iec:7498:-1:ed-1:v2:en. https://www.iso.org/obp/ui/en/#iso:std:iso-iec:7498:-1:ed-1:v2:en

[36] Stewart, William J.: *Chapter 11. Elementary Queueing Theory.* S. 385–443. In: *Probability, Markov Chains, Queues, and Simulation.* Princeton : Princeton Univer-

sity Press, 2009. – URL https://doi.org/10.1515/9781400832811-012. – ISBN 9781400832811

[37] Tychalas, Dimitrios ; Karatza, Helen: An Advanced Weighted Round Robin Scheduling Algorithm. In: *24th Pan-Hellenic Conference on Informatics*. New York, NY, USA : Association for Computing Machinery, 2020 (PCI 2020), S. 188–191. – URL https://doi.org/10.1145/3437120.3437304. – ISBN 9781450388979

[38] Valeur, Fredrik ; Vigna, Giovanni ; Kruegel, Christopher ; Kirda, Engin: An Anomaly-Driven Reverse Proxy for Web Applications. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. New York, NY, USA : Association for Computing Machinery, 2006 (SAC '06), S. 361–368. – URL https://doi.org/10.1145/1141277.1141361. – ISBN 1595931082

[39] Vilaplana, Jordi ; Solsona, Francesc ; Teixidó, Ivan ; Mateo, Jordi ; Abella, Francesc ; Rius, Josep: A queuing theory model for cloud computing. In: *The Journal of Supercomputing* 69 (2014), Jul, Nr. 1, S. 492–507. – URL https://doi.org/10.1007/s11227-014-1177-y. – ISSN 1573-0484

A Anhang

A.1 Implementierungen

Für das gesamte Projekt siehe: GitHub Repository

A.1.1 Lastverteiler

```
import atexit
import json
import os
import socket
import subprocess
import threading
from operator import attrgetter
from time import time, sleep, strftime
import docker
import psutil
import random
import requests
import yaml
from flask import Flask, request

# Flask App Definition
loadbalancer = Flask(__name__)

# Backend class definition
class Backend:
    def __init__(self, port, weight):
```

```
        self.port = port
        self.weight = weight
        self.userCount = 0
        self.usageCount = 0
        self.serviceTime = 1
        self.selectionProbability = 0
        self.utilization = 0
        self.isStopping = False


# Class to save all backend and
# and keeping track of the index of the
# next backend
class BackendList:
    Backends = []
    next = 0


#GLOBALS
stoppingThreads = False
BackendList = BackendList()
ContainerList = []
ContainerBackendDict = {}
client = docker.from_env()
totalRequests = 0
servedRequests = 0

# Configuration Variables
with open("config.yml") as config_file:
    config = yaml.load(config_file, Loader=yaml.FullLoader)
    path = config["path"]
    numberOfInstances = config["numberOfInstances"]
    portLoadbalancer = config["portLoadbalancer"]
    imageName = config["imageName"]
    dockerPort = config["dockerPort"]
    loadbalancingAlgorithm = config["loadbalancingAlgorithm"]
```

```
    scalingMechanism = config["scalingMechanism"]
    upperUtilizationLimit = config["upperUtilizationLimit"]
    lowerUtilizationLimit = config["lowerUtilizationLimit"]
    maxContainer = config["maxContainer"]
    numberOfClients = config["numberOfClients"]
    filePath = config["filePath"]

# function to get next Backend
# depending on the configured algorithm
def getNextBackend() -> Backend:
    match loadbalancingAlgorithm:
        case "random":
            backend = None
            while backend is None:
                backend = random.choice(BackendList.Backends)
                if backend.isStopping:
                    backend = None
            backend.usageCount += 1
            return backend
        case "RR":
            index = BackendList.next
            backend = BackendList.Backends[index]
            BackendList.next = (index + 1) % len(BackendList.Backends)
            backend.usageCount += 1
            return backend
        case "AWRR":
            updateWeights()
            backend = max(BackendList.Backends, key=attrgetter("selectionProbability"))
            backend.usageCount += 1
            return backend
        case _:
            return BackendList.Backends[0]


# function returning the total
# weight of all Backends
```

```
def getTotalWeight():
    totalWeight = 0
    for backend in BackendList.Backends:
        totalWeight += 1 / backend.serviceTime
    return totalWeight

# function returning the utilization of
# all backend via their /getUtilization endpoint
def getUtilization():
    Utilization = 0
    for backend in BackendList.Backends:
        sleep(0.1)
        headers = {
            "Content-Type": "application/json;charset=UTF-8",
            "User-Agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)"
        }
        response = requests.get("http://localhost:{}".format(backend.port) + "/getUtilization"
                                , headers=headers)
        Utilization += json.loads(response.content)["utilization"]  # backend.utilization
        backend.utilization = min(0.99, Utilization + (backend.userCount / 100) * 2)
    return Utilization / len(BackendList.Backends)

# function updating the weights of all backends
# if scaling on, via their /getUtilization endpoint
def updateWeights():
    totalWeight = getTotalWeight()
    dynamicTotal = 0
    for backend in BackendList.Backends:
        if scalingMechanism == "none":
            sleep(0.1)
            headers = {
                "Content-Type": "application/json;charset=UTF-8",
                "User-Agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)"
            }
            response = requests.get("http://localhost:{}".format(backend.port) + "/getUtilization"
                                    , headers=headers)
```

```
            Utilization = json.loads(response.content)["utilization"]
            backend.utilization = min(0.99, Utilization + (backend.userCount / 100) * 2)
        backend.weight = (1 / backend.serviceTime) / totalWeight
        dynamicTotal += (backend.weight * (1 - backend.utilization))
    for backend in BackendList.Backends:
        backend.selectionProbability = (backend.weight * 1 - backend.utilization) / dynamicTotal
        # if backend.userCount > 0:
        #     selectionProbability = 0
        # backend.selectionProbability = selectionProbability


# endpoint to terminate Loadbalancer runtime and metrics capturing
@loadbalancer.route("/shutdown", methods=['GET'])
def shutdown():
    signal_handler()
    return "Shutting down..."

# main application endpoint
@loadbalancer.route('/')
def router():
    LoadbalancerReceived = time()
    global totalRequests
    totalRequests += 1
    backend = getNextBackend()
    print(f"Serving from Port: {backend.port}")
    backend.userCount += 1
    data = json.loads(request.json)
    sleep(0.2)
    headers = {
        "Content-Type": "application/json;charset=UTF-8",
        "User-Agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)"
    }
    response = requests.get("http://localhost:{}".format(backend.port) + path, json=json.dumps(data),
                            headers=headers)
    print(response)
    data = response.json()
```

```
    data["LoadbalancerReceived"] = str(LoadbalancerReceived)
    data["LoadbalancerResponse"] = str(time())
    response = loadbalancer.response_class(
        response=json.dumps(data),
        status=200,
        mimetype='application/json'
    )
    global servedRequests
    servedRequests += 1
    backend.userCount -= 1
    return response

# function starting container
def startContainer():
    con = client.containers.run(imageName, detach=True, ports={dockerPort: get_free_port()})
    ContainerList.append(con)
    backend = Backend(list(con.attrs["HostConfig"]["PortBindings"].values())[0][0]["HostPort"], 0)
    BackendList.Backends.append(backend)
    ContainerBackendDict[backend] = con

# function stopping and removing container
def stopAndRemoveContainer(container):
    ContainerList.remove(container)
    backend = list(ContainerBackendDict.keys())[list(ContainerBackendDict.values()).index(container)]
    backend.isStopping = True
    ContainerBackendDict.pop(backend)
    BackendList.Backends.remove(backend)
    print("Backend removed, stopping Container")
    container.remove(force=True)

# function to stop an remove all containers
def stopAndRemoveAllContainer():
    for container in ContainerList:
        container.remove(force=True)

# function returning a free port
```

```
def get_free_port():
    sock = socket.socket()
    sock.bind(('', 0))
    port = sock.getsockname()[1]
    sock.close()
    return port

# function to handle outer termination
def signal_handler():
    print("Stopping Observers")
    global stoppingThreads
    stoppingThreads = True
    print("Observers stopped")
    print("Stopping Container")
    stopAndRemoveAllContainer()
    print("All Container stopped")
    dockerStatsCapture.terminate()


# function returning a low utilised or idle backend
def getIdleOrLowUtilisedBackend():
    backend = next(backend for backend in BackendList.Backends if not backend.userCount > 0)
    if backend is None:
        backend = min(BackendList.Backends, key=attrgetter("utilization"))
    if backend.userCount > 0:
        return None
    return backend

# function checking Utilization every 5 seconds
def checkUtilization():
    sleep(5)
    while not stoppingThreads:
        print(f"Pending Requests: {totalRequests - servedRequests}")
        totalUtilization = getUtilization()
        if totalUtilization >= upperUtilizationLimit and len(
                BackendList.Backends) < maxContainer:
```

```
            print("starting New Backend")
            startContainer()
        elif totalUtilization < lowerUtilizationLimit and len(BackendList.Backends) > 1:
            backend = getIdleOrLowUtilisedBackend()
            if backend is not None:
                print("Removing Backend")
                stopAndRemoveContainer(ContainerBackendDict[backend])
        else:
            print("Nothing todo")
        sleep(5)

# function capturing LB metrics every second
def captureMetrics():
    filename = f"{filePath}LoadbalancerStats_{strftime('%Y.%m.%d—%H:%M:%S')}" \
               f"_Sn={numberOfInstances}_LA={loadbalancingAlgorithm}_SM={scalingMechanism}.txt"
    f = open(filename, "a")
    f.write("Time:CPU:Memory\n")
    f.close()
    while not stoppingThreads:
        f = open(filename, "a")
        cpu = process.cpu_percent()
        mem = process.memory_info().rss / 1024 ** 2
        f.write(f"{time()}:{cpu} %:{mem} MiB\n")
        f.close()
        sleep(1)


# Main Execution
if __name__ == '__main__':
    process = psutil.Process(os.getpid())
    atexit.register(signal_handler)
    i = 0
    for i in range(numberOfInstances):
        startContainer()
    print(f"{i + 1} Container started")
    if scalingMechanism != "none":
```

```
        App_observerThread = threading.Thread(target=checkUtilization, name="App_observer")
        App_observerThread.daemon = True
        App_observerThread.start()
    LB_observerThread = threading.Thread(target=captureMetrics, name="LB_observer")
    LB_observerThread.daemon = True
    LB_observerThread.start()
    dockerStatsCapture = subprocess.Popen(["sh",
                                           "./dockerStatsCapture.sh",
                                           f"{filePath}ContainerStats_"
                                           f"{strftime('%Y.%m.%d_%H:%M:%S')}"
                                           f"_Sn={numberOfInstances}"
                                           f"_LA={loadbalancingAlgorithm}"
                                           f"_SM={scalingMechanism}"])
    loadbalancer.run(host='0.0.0.0', port=portLoadbalancer, threaded=True)
```

A.1.2 Ungeeignete Anwendung

```
import json
import os
import threading
from decimal import Decimal
from flask import Flask, request
import psutil
from time import time


sem = threading.Semaphore()


process = psutil.Process(os.getpid())



# http://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80
def calcPi(n):
    pi, numer = Decimal(0), Decimal(4)
    for i in range(n):
        denom = (2 * i + 1)
```

```
        term = numer / denom
        if i % 2:
            pi -= term
        else:
            pi += term
    return pi


app = Flask(__name__)

# endpoint calling calcPi() and
# responding to sender
# http://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80
@app.route("/calc")
def calculatePI():
    sem.acquire()
    data = json.loads(request.json)
    ApplicationReceived = time()
    calcPi(10000000)
    data['success'] = True
    data['ApplicationReceived'] = ApplicationReceived
    data['ApplicationResponse'] = time()
    response = app.response_class(
        response=json.dumps(data),
        status=200,
        mimetype='application/json'
    )
    sem.release()
    return response

# endpoint responding with the current utilization
@app.route("/getUtilization")
def getWeight():
    cpu = process.cpu_percent()
    mem = process.memory_percent()
    return json.dumps({'utilization': ((cpu + mem) / 2) / 100}),\
```

```
            200, {'ContentType': 'application/json'}


# Main Execution
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8080, threaded=False)
```

A.1.3 Dockerfile der ungeeigneten Anwendung

```
# Base Image
FROM python:3.8-slim-buster

# Working directory
WORKDIR /app

# Copy and install all required modules and libraries
COPY requirements.txt requirements.txt
RUN pip install -r requirements.txt

# Copy all in working directory
COPY . .

# Execution command
CMD [ "python3", "-m" , "flask", "run", "--host=0.0.0.0", "--port=5000"]
```

A.1.4 Client-Simulationsanwendung

```
import asyncio
import json
import random
import threading
import uuid
from time import time, sleep, strftime
import aiohttp
```

```
import requests
import yaml

# async function sending a request to the loadbalancer and awaiting a response
# after receiving a response the function writes all timestamps and information to
# the metrics output file
async def simulateClient(i):
    loop = asyncio.get_event_loop()
    client = aiohttp.ClientSession(loop=loop)
    clientID = str(uuid.uuid1())
    print(f"Starting Client Simulation. ClientId: {clientID}")
    RequestID = str(uuid.uuid1())
    payload = {"uuid": RequestID,
               "clientSend": str(time())
               }
    async with client.get(url, json=json.dumps(payload), headers={
        "Content-Type": "application/json;charset=UTF-8",
        "User-Agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)"
    }) as response:
        assert response.status == 200

        await response.read()
        data = await response.json()
        await client.close()
        clientReceived = time()
        f = open(MetricsFilePath, "a")
        f.write(f"{i}|{data['clientSend']}|RequestID:{RequestID}|Client:{clientID}"
                f"|Send new Request\n")
        f.write(f"{i}|{data['LoadbalancerReceived']}|RequestID: {RequestID}"
                f"|Loadbalancer|Received new Request\n")
        f.write(
            f"{i}|{data['ApplicationReceived']}|RequestID: {RequestID}"
            f"|Application|Received new Request. Starting Calculation\n")
        f.write(
            f"{i}|{data['ApplicationResponse']}|RequestID: {RequestID}"
            f"|Application|Calculation finished. Sending Response\n")
```

```
        f.write(f"{i}|{data['LoadbalancerResponse']}|RequestID: {RequestID}"
                f"|Loadbalancer|Request processed\n")
        f.write(f"{i}|{clientReceived}|{RequestID}|RequestID: Client: {clientID}"
                f"|Received Response. Exiting\n")
        f.close()

# Configuration Variables
with open("config.yml") as config_file:
    config = yaml.load(config_file, Loader=yaml.FullLoader)
    numberOfInstances = config["numberOfInstances"]
    portLoadbalancer = config["portLoadbalancer"]
    loadbalancingAlgorithm = config["loadbalancingAlgorithm"]
    scalingMechanism = config["scalingMechanism"]
    numberOfClients = config["numberOfClients"]
    filePath = config["filePath"]
    timeBetweenClientsMSMax = config["timeBetweenClientsMSMax"]
    timeBetweenClientsMSMin = config["timeBetweenClientsMSMin"]

MetricsFilePath = f"{filePath}Timestamps_{strftime('%Y.%m.%d-%H:%M:%S')}" \
                  f"_Sn={numberOfInstances}_LA={loadbalancingAlgorithm}" \
                  f"_SM={scalingMechanism}.txt"
url = f"http://localhost:{portLoadbalancer}"

# starting 1 Thread per Client
threads = []
for i in range(numberOfClients):
    t = threading.Thread(target=asyncio.run, args=(simulateClient(i),))
    t.daemon = True
    threads.append(t)
    t.start()
    sleep(random.randint(timeBetweenClientsMSMin, timeBetweenClientsMSMax) / 1000)


# waiting for all clients to finish
for x in threads:
    x.join()
```

```
# wait 5 seconds and send termination request to loadbalancer
sleep(5)
requests.get("http://localhost:{}".format(portLoadbalancer) + "/shutdown", headers={
    "Content-Type": "application/json;charset=UTF-8",
    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)"
})
```

A.1.5 Aufzeichnung der Docker Metriken

```
#bin/bash

# File capturing docker container stats

# Time interval
INTERVAL=2
#file output
OUTNAME=$1.txt

#first line with startin timestamp
echo $(date +'%s.%N') | tee --append $OUTNAME
# column names and delimiter
echo "Name:CPUPercent:MemUsage:Timestamp" | tee --append $OUTNAME

# function writing docker stats output to file with custom format
update_file() {
  docker stats --no-stream --format "{{.Name}}:{{.CPUPerc}}:{{.MemUsage}}:
  $(date +'%s.%N')" | tee --append $OUTNAME
}


while true; do
  update_file &
  sleep $INTERVAL
done
```

A.1.6 Erstellung der Graphen der Simulation

```
from datetime import datetime

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import os
import latex

Versuch = "Versuch3_2"
FilePath = f"Messungen/{Versuch}/"

for filename in os.listdir(FilePath):
    root, ext = os.path.splitext(filename)
    if root.startswith("Container"):
        fileNameContainerStats = filename
    elif root.startswith("Loadbalancer"):
        fileNameLoadbalancerStats = filename
    elif root.startswith("Timestamps"):
        fileNameTimestamps = filename



def PlotContainerStats():
    ContainerStatsStartTime = float(open(
        f"{FilePath}{fileNameContainerStats}", "r").readline())

    ContainerStatsDF = pd.read_csv(f"{FilePath}{fileNameContainerStats}",
                                   sep=":", header=1)
    ContainerStatsDF = ContainerStatsDF[ContainerStatsDF["Name"] != "——"]
    ContainerStatsEndTime = (max(ContainerStatsDF["Timestamp"]))
    ContainerStatsDF["CPUPercent"] = ContainerStatsDF["CPUPercent"].apply(
        lambda x: float(x.strip("%")))
    ContainerStatsDF["TimeDiff"] = ContainerStatsDF["Timestamp"].apply(
        lambda x: x — ContainerStatsStartTime)
```

```
setRcParams()
timeRange = np.arange(min(ContainerStatsDF["TimeDiff"]),
                      max(ContainerStatsDF["TimeDiff"]) + 5, 20)
plt.xticks(timeRange)
plt.yticks(np.arange(min(ContainerStatsDF["CPUPercent"]),
                     max(ContainerStatsDF["CPUPercent"]) + 10, 10))

test = ContainerStatsDF.groupby("Name")

for group in test.groups.keys():
    plt.plot(test.get_group(group)["TimeDiff"], test.get_group(group)["CPUPercent"])
    #, label=group

#   plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
plt.xlabel("Time", fontdict=fontLabel, labelpad=15)
plt.ylabel("Cpu Usage in percent", fontdict=fontLabel, labelpad=15)
plt.savefig(f'Plots/{Versuch}_ContainerStatsCPU.png', bbox_inches='tight', dpi=200)
plt.clf()

# 19.46MiB / 30.65GiB
ContainerStatsDF["MemUsage"] = ContainerStatsDF["MemUsage"]\
    .apply(lambda x: x.split("MiB")[0])
ContainerStatsDF["MemUsage"] = ContainerStatsDF["MemUsage"]\
    .apply(lambda x: x.split("B")[0])
ContainerStatsDF["MemUsage"] = ContainerStatsDF["MemUsage"]\
    .apply(lambda x: x.split("KiB")[0])
ContainerStatsDF["MemUsage"] = ContainerStatsDF["MemUsage"].\
    apply(lambda x: x.split("GiB")[0])
ContainerStatsDF["MemUsage"] = ContainerStatsDF["MemUsage"].\
    apply(lambda x: float(x))
ContainerStatsDF["TimeDiff"] = ContainerStatsDF["Timestamp"].\
    apply(lambda x: x - ContainerStatsStartTime)
setRcParams()
timeRange = np.arange(min(ContainerStatsDF["TimeDiff"]),
                      max(ContainerStatsDF["TimeDiff"]) + 5, 20)
plt.xticks(timeRange)
```

```
    plt.yticks(np.arange(min(ContainerStatsDF["MemUsage"]),
                         max(ContainerStatsDF["MemUsage"]) + 1, 5))
    test = ContainerStatsDF.groupby("Name")

    for group in test.groups.keys():
        plt.plot(test.get_group(group)["TimeDiff"], test.get_group(group)["MemUsage"])

#   plt.legend(bbox_to_anchor=(1, 0.5), loc='center left')
    plt.xlabel("Time", fontdict=fontLabel, labelpad=15)
    plt.ylabel("Memory Usage in MiB", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/{Versuch}_ContainerStatsRAM.png', bbox_inches='tight', dpi=200)
    plt.clf()


def PlotLoadbalancerStats():
    LoadbalancerStats = pd.read_csv(f"{FilePath}{fileNameLoadbalancerStats}"
                                    , sep=":", header=0)
    #CPU
    setRcParams()
    LoadbalancerStats["CPU"] = LoadbalancerStats["CPU"].apply(lambda x: float(x.strip(" %")))
    plt.plot(range(LoadbalancerStats["CPU"].count()), LoadbalancerStats["CPU"])
    plt.xticks(np.arange(min(range(LoadbalancerStats["Memory"].count())),
                         max(range(LoadbalancerStats["Memory"].count())) + 1, 10))
    plt.yticks(np.arange(min(LoadbalancerStats["CPU"]), max(LoadbalancerStats["CPU"]) + 1, 5))
    plt.xlabel("Time", fontdict=fontLabel, labelpad=15)
    plt.ylabel("Cpu Usage in percent", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/{Versuch}_LoadbalancerStatsCPU.png', bbox_inches='tight', dpi=200)
    plt.clf()

    #RAM
    setRcParams()
    LoadbalancerStats["Memory"] = LoadbalancerStats["Memory"]\
        .apply(lambda x: float(x.strip(" MiB")))
    plt.plot(range(LoadbalancerStats["Memory"].count()), LoadbalancerStats["Memory"])
    plt.xticks(np.arange(min(range(LoadbalancerStats["Memory"].count())),
                         max(range(LoadbalancerStats["Memory"].count())) + 1, 10))
```

```
    plt.yticks(np.arange(round(min(LoadbalancerStats["Memory"])),
                         round(max(LoadbalancerStats["Memory"])) + 1, 0.5))
    plt.xlabel("Time", fontdict=fontLabel, labelpad=15)
    plt.ylabel("Memory Usage in MiB", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/{Versuch}_LoadbalancerStatsRAM.png', bbox_inches='tight', dpi=200)
    plt.clf()


def PlotTimestamps():
    # 0|1666362474.1514375|RequestID:8ce4aa85-514c-11ed-b603-359305e241e9|
    # Client:8ce4aa84-514c-11ed-b603-359305e241e9|Send new Request
    colnames = ["Index", "Timestamp", "RequestID", "ClientID", "Message"]
    TimestampsDF = pd.read_csv(f"{FilePath}{fileNameTimestamps}", sep="|", names=colnames)

    # Response Time per Request
    setRcParams()
    perRequest = TimestampsDF.groupby("Index")

    for request in perRequest.groups.keys():
        start = datetime.fromtimestamp(min(perRequest.get_group(request)["Timestamp"]))
        end = datetime.fromtimestamp(max(perRequest.get_group(request)["Timestamp"]))
        tdelta = end - start
        seconds = tdelta.total_seconds()
        plt.bar(perRequest.get_group(request)["Index"], seconds)
    plt.xlabel("Request", fontdict=fontLabel, labelpad=15)
    plt.ylabel("Response Time", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/{Versuch}_TimeStamps_ResponseTime.png', bbox_inches='tight', dpi=200)
    plt.clf()

    # Response Time per Request
    setRcParams()
    perRequest = TimestampsDF.groupby("Index")
    # Processing Time per Request
    for request in perRequest.groups.keys():
        df = perRequest.get_group(request)
        start = datetime.fromtimestamp(
```

```
            df.loc[df["Message"] ==
                   "Received new Request. Starting Calculation"]["Timestamp"].values[0])
        end = datetime.fromtimestamp(
            df[df["Message"] == "Calculation finished. Sending Response"]["Timestamp"].values[0])
        tdelta = end - start
        seconds = tdelta.total_seconds()
        plt.bar(df["Index"], seconds)
    plt.xlabel("Request", fontdict=fontLabel, labelpad=15)
    plt.ylabel("Processing Time", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/{Versuch}_TimeStamps_ProcessingTime.png',
                bbox_inches='tight', dpi=200)
    plt.clf()


def plotTotalResponseTimeOfEachExperiment():
    setRcParams()
    FilePath_ = "Messungen/"
    for _, dir_, _ in os.walk(FilePath_):
        dir_.sort()
        for experiment in dir_:
            for filename_ in os.listdir(FilePath_ + experiment + "/"):
                root_, ext_ = os.path.splitext(filename_)
                if root_.startswith("Timestamps"):
                    colnames = ["Index", "Timestamp", "RequestID", "ClientID", "Message"]
                    TimestampsDF = pd.read_csv(f"{FilePath_}{experiment}/"
                                               f"{filename_}", sep="|", names=colnames)
                    start = datetime.fromtimestamp(min(TimestampsDF["Timestamp"]))
                    end = datetime.fromtimestamp(max(TimestampsDF["Timestamp"]))
                    tdelta = end - start
                    seconds = tdelta.total_seconds()
                    plt.bar(experiment, seconds)
    plt.ylabel("Time in Seconds", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/TotalResponseTimes.png', bbox_inches='tight', dpi=200)
    plt.clf()
```

```
def plotTotalMeanResponseTimeOfEachExperiment():
    setRcParams()
    FilePath_ = "Messungen/"
    for _, dir_, _ in os.walk(FilePath_):
        dir_.sort()
        for experiment in dir_:
            for filename_ in os.listdir(FilePath_ + experiment + "/"):
                root_, ext_ = os.path.splitext(filename_)
                if root_.startswith("Timestamps"):
                    colnames = ["Index", "Timestamp", "RequestID", "ClientID", "Message"]
                    TimestampsDF = pd.read_csv(f"{FilePath_}{experiment}/"
                                               f"{filename_}", sep="|", names=colnames)
                    perRequest = TimestampsDF.groupby("Index")
                    totalSeconds = 0
                    for request in perRequest.groups.keys():
                        start = datetime.fromtimestamp(
                            min(perRequest.get_group(request)["Timestamp"]))
                        end = datetime.fromtimestamp(
                            max(perRequest.get_group(request)["Timestamp"]))
                        tdelta = end - start
                        totalSeconds += tdelta.total_seconds()
                    mean = totalSeconds / len(perRequest)
                    plt.bar(experiment, mean)
    plt.ylabel("Time in Seconds", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/MeanResponseTimes.png', bbox_inches='tight', dpi=200)
    plt.clf()


def plotTotalMeanProcessingTimeOfEachExperiment():
    setRcParams()
    FilePath_ = "Messungen/"
    for _, dir_, _ in os.walk(FilePath_):
        dir_.sort()
        for experiment in dir_:
            for filename_ in os.listdir(FilePath_ + experiment + "/"):
                root_, ext_ = os.path.splitext(filename_)
```

```
                if root_.startswith("Timestamps"):
                    colnames = ["Index", "Timestamp", "RequestID", "ClientID", "Message"]
                    TimestampsDF = pd.read_csv(f"{FilePath_}{experiment}/"
                                               f"{filename_}", sep="|", names=colnames)
                    perRequest = TimestampsDF.groupby("Index")
                    totalSeconds = 0
                    for request in perRequest.groups.keys():
                        df = perRequest.get_group(request)
                        start = datetime.fromtimestamp(
                            df.loc[df["Message"] == "Received new Request. "
                                                    "Starting Calculation"]["Timestamp"].values[
                                0])
                        end = datetime.fromtimestamp(
                            df[df["Message"] == "Calculation finished. "
                                                "Sending Response"]["Timestamp"].values[0])
                        tdelta = end - start
                        totalSeconds += tdelta.total_seconds()
                    mean = totalSeconds / len(perRequest)
                    plt.bar(experiment, mean)
    plt.ylabel("Time in Seconds", fontdict=fontLabel, labelpad=15)
    plt.savefig(f'Plots/MeanProcessingTimes.png', bbox_inches='tight', dpi=200)
    plt.clf()


font = {
    "family": "serif",
    "size": 20,
}
fontLabel = {
    "family": "serif",
    "size": 30,
}


def setRcParams():
    plt.style.context('science')
```

```
plt.rcParams["figure.figsize"] = (20, 7.5)
plt.rcParams["text.usetex"] = True
matplotlib.rc("font", **font)
```

Erklärung zur selbstständigen Bearbeitung

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

Ort	Datum	Unterschrift im Original