Dezentrales Management von großen verteilten Systemen

Dissertation

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

(Dr. rer. nat.)

der Fakultät für Angewandte Informatik

der Universität Augsburg

eingereicht von
Dipl.-Inf. Michael Roth

Erstgutachter:	Prof. Dr. rer. nat. Theo Ungerer
Zweitgutachter:	Prof. Dr. rer. nat. Jörg Hähner

Tag der mündlichen Prüfung: 10. Februar 2015

Abstract

Distributed systems are becoming increasingly larger and more complex. Examples of such large distributed systems are cloud computing data centers, in which thousands of computers work together to simulate unlimited resources for the cloud computing user. By dynamically allocating the available resources to different users with different usage behavior, high utilization can be achieved. It is therefore more economical for customers to share cloud computing resources rather than build their own computing centers, which remain largely unused and are only utilized to their full capacity during peak periods. The customer can quickly add new resources to his application and remove them if they are no longer required. Most cloud computing systems of this kind are controlled centrally. This means that central monitoring downtime or errors can cripple the entire application.

Using the example of a web application, this paper presents a concept for the decentralized control of large cloud computing systems. The decentralized controller allows the application to react and heal itself, even if a large number of components are down. In the example, the requests of website visitors are divided between different servers of a cloud computing application and processed on those servers. In order to maintain a constant processing time, the number of virtual computers used at any one time must vary depending on the site traffic. The aim of the website operator is to minimize costs by renting as few virtual machines as possible without losing visitors due to long response times. A distributed controller must be able to meet these two conflicting requirements.

In order to implement an efficient controller for large distributed cloud systems, two aspects are examined. Firstly, an efficient means of disseminating the load data from individual machines to all other machines must be developed. Due to the fact that they are completely decentralized and highly tolerant of server downtimes, peer-to-peer-based distribution algorithms are developed and tested as a method of information dissemination. These algorithms guarantee the complete dissemination of information; however, not all information is up to date. Aggregating the collected information can reduce the burden on the network, but this information will not be as exact.

In the second step, this distributed load data is used to control the number of machines employed. The controller must deal with these imprecise data sets and manage the system. Here, two decentralized methods are examined and compared with a central implementation. Arithmetic and fuzzy logic controllers are investigated and discussed.

Both methods are implemented with and without aggregated information. Global knowledge makes it possible for the centralized control algorithms to maintain a more stable system with shorter reaction times.

The distribution and control algorithms presented in this paper allow fully decentralized control of the system. With the fuzzy logic controller presented here, the system can be successfully controlled with a response time deviation of 5.8 % as compared to 4.1 %. In order to increase reliability, the decentralized controller requires 3.5 % more resources.

Zusammenfassung

Verteilte Systeme werden immer komplexer und größer. Beispiele für solch große verteilte Systeme sind Cloud-Computing-Rechenzentren. Tausende von Computern arbeiten zusammen um dem Cloud-Computing-Benutzer unbegrenzte Ressourcen zu simulieren. Durch die dynamische Aufteilung der vorhanden Ressourcen auf verschiedene Benutzer mit unterschiedlichem Nutzungsverhalten kann eine hohe Auslastung erzielt werden. Deshalb ist es für Kunden wirtschaftlicher, geteilte Cloud-Computing-Ressourcen zu nutzen, anstatt eigene Kapazitäten aufzubauen, die meistens ungenutzt sind und nur zu Spitzenzeiten voll ausgelastet werden. Der Kunde kann schnell neue Ressourcen zu seiner Anwendung hinzufügen und wieder entfernen, sollten diese nicht mehr benötigt werden. Die Überwachung solcher Cloud-Computing-Systeme und die Berechnung des Ressourcenbedarfs geschieht zentral. Ein Ausfall oder Fehler der zentralen Überwachung kann die komplette Anwendung lahmlegen.

In dieser Arbeit wird ein Konzept präsentiert, wie große Cloud-Computing-Systeme komplett dezentral gesteuert werden können. Durch die dezentrale Steuerung kann die Anwendung selbst bei dem Ausfall einer großen Anzahl von Komponenten noch reagieren und sich selbst heilen. Als Beispiel wird eine Internetanwendung vorgestellt. Dabei werden die Internetseitenbesucher auf verschiedene Server einer Cloud-Computing-Anwendung verteilt und dort die Anfragen bearbeitet. Um die Bearbeitungszeit konstant zu halten werden je nach Besucherzahlen mehr oder weniger virtuelle Computer benutzt. Der Internetseitenbetreiber will die Kosten minimieren und daher möglichst wenige Ressourcen anmieten. Andererseits sollen keine Seitenbesucher durch lange Antwortzeiten abgeschreckt werden. Die Antwortzeit kann durch eine größere Anzahl von Servern verringert werden. Eine verteilte Steuerung muss in der Lage sein, diese beiden gegensätzlichen Ziele zu erfüllen.

Um die Steuerung solch großer verteilten Cloud-Computing-Systeme mit tausenden von Servern zu implementieren werden zwei Aspekte untersucht. Als erstes müssen Lastdaten einzelner Maschinen an alle anderen gesendet werden. Peer-to-Peer-Algorithmen sind häufig komplett dezentral und besitzen eine hohe Toleranz gegenüber Serverausfällen. Auf Grund dieser Eigenschaften wurden für die Informationsverteilung dezentrale, auf Peer-to-Peer-Netze

basierende Verteilungsalgorithmen entwickelt und untersucht. Diese Algorithmen garantieren die komplette Verteilung der Informationen, allerdings sind nicht alle Informationen aktuell. Durch Aggregation der gesammelten Informationen kann die Belastung des Netzwerks gesenkt werden, jedoch werden die Informationen ungenauer.

Im zweiten Schritt wird mit diesen Informationen die Anzahl der Maschinen gesteuert. Die Steuerung muss mit den ungenauen und teilweise veralteten Daten umgehen und das System verwalten. Dafür werden zwei dezentrale Methoden untersucht und jeweils mit einer zentralen Implementierung verglichen. Es werden arithmetische und Fuzzylogik-Steuerungen untersucht und diskutiert.

Beide Methoden werden jeweils mit und ohne aggregierten Informationen implementiert. Die zentralen Steuerungsalgorithmen konnten auf Grund des globalen Wissens das System mit kürzeren Reaktionszeiten stabiler halten.

Mit den in dieser Arbeit vorgestellten Verteilungs- und Steuerungsalgorithmen kann das System komplett dezentral gesteuert werden. Die vorgestellte Fuzzylogik-Steuerung kann das System mit einer Abweichungen der Antwortzeit von 5,8%, statt 4,1 % bei der zentralen Steuerung, erfolgreich steuern. Die dezentrale Steuerung benötigt 3,5% mehr Ressourcen, erspart dafür aber die aufwändige Absicherung einer zentralen Überwachungsinstanz. Durch die dezentrale Steuerung kann darauf verzichtet werden.

Vorwort

Diese Arbeit entstand in den Jahren 2009 bis 2014 während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Systemnahe Informatik und Kommunikationssysteme der Universität Augsburg. Im Rahmen des DFG-Schwerpunktprogramms 1883 *Organic Computing* wurde die Middleware OCμ weiterentwickelt. Diese war die Grundlage für die hier vorliegende Arbeit mit großen verteilten Systemen.

An dieser Stelle möchte ich mich für die ausgezeichnete Betreuung durch Prof. Theo Ungerer bedanken. Er ermöglichte es mir meine Forschung nach Ende des OCμ-Projektes noch als Lehrstuhlmitarbeiter abzuschließen.

Meinen Kollegen Julia Schmitt, Rolf Kiefhaber und Florian Kluge möchte ich für die gute Zusammenarbeit in der *Organic Computing*-Gruppe danken.

Zusätzlich danke ich noch meinen Kollegen am Lehrstuhl für die ausgiebigen Diskussionen während der Lehrstuhlseminare. Besonderer Dank gebührt Stefan Metzlaf und Sebastian Weis, denen ich häufiger vom Stand meiner Forschung berichten durfte. Öfters habe ich noch Abends auf der Logia die nächsten Schritte mit ihnen diskutiert.

Der Fast-Food-Gruppe (Christian, Mike, Julian und Sebi) danke ich für die abwechslungsreiche Mittagsgestaltung.

Auch möchte ich Prof. Jörg Hähner danken, dass er sich bereiterklärt hat als Zweitgutachter an meinem Promotionsverfahren mitzuwirken.

Vielen Dank gilt auch meinen Eltern für die Unterstützung während des Studiums und für das Korrekturlesen dieser Arbeit.

Besonders möchte ich mich bei meiner Frau Miriam für ihren Rückhalt und ihre Unterstützung bedanken. Besonders in der Endphase haben sie und unsere Tochter Isabell den notwendigen Ausgleich für die Arbeit geschaffen.

Augsburg, April 2015 *Michael Roth*

Inhaltsverzeichnis

1. Einleitung

Die von der Bitkom durchgeführte Studie *Cloud Monitor 2014*[1] zeigt, dass Cloud-Computing stetig an Bedeutung gewinnt. Für die kommenden Jahre werden steigende Umsätze in diesem Bereich erwartet. Im Cloud-Computing werden die vorhanden Ressourcen (z.B. Rechenleistung, Speicher) virtualisiert und können dann je nach Bedarf von den Benutzern verwendet werden. Dabei können die Ressourcen dynamisch bereitgestellt werden. Durch das unterschiedliche Nutzungsverhalten der Benutzer können Spitzenlasten abgefangen und trotzdem eine hohe Auslastung über den kompletten Tag erreicht werden. Cloud-Computing wird in zwei Bereiche unterteilt: *Public-* und *Private-Cloud*.

In einer *Private-Cloud* wird die komplette Cloud-Computing-Infrastruktur im eigenen Rechenzentrum betrieben. Die vollständige Steuerung der Cloud muss hier selbst umgesetzt werden. Dies bedeutet einen Mehraufwand für die Firma, dafür wird die Verwaltung anderer Systeme vereinfacht und die Ressourcen können flexibel von den Mitarbeitern genutzt werden. Es wird die Illusion von unbegrenzten Ressourcen für einzelne Anwender erzeugt[3].

Meistens werden solche Systeme zentral durch eine oder mehrere Überwachungsinstanzen gesteuert. Dies birgt das Risiko eines *Single Point of Failure*. Durch den Ausfall der Überwachungsinstanz ist die komplette Cloud nicht mehr benutzbar. Um dies zu verhindern, werden die wichtigen Teile redundant ausgelegt. Ein alternativer Ansatz ist die Steuerung der Cloud dezentral auf alle Computer zu verteilen. Dadurch ist selbst bei einem Ausfall mehrerer Komponenten eines Rechenzentrums der Rest noch als Cloud nutzbar.

In der *Public-Cloud* wird das Cloud-Computing-Rechenzentrum von einem großen Anbieter, wie z.B. Amazon, Micrososft oder Google, betrieben. Entwickler können Ressourcen dynamisch von den Anbietern mieten. Es werden CPU-Zeit, Speicherplatz und Netzwerkbandbreite angeboten. Abgerechnet werden nur die benötigen Ressourcen. Im Englischen wird das aussagekräftig

[1] `http://www.bitkom.org/de/publikationen/38338_79386.aspx`

als *pay-as-you-go* Modell bezeichnet. Entwickler können so neue Internetanwendungen entwickeln ohne sich um die Hardwarewartung zu kümmern oder große Beträge für die Hardwarebeschaffung finanzieren zu müssen. Zusätzliche Ressourcen können innerhalb von Minuten bereitgestellt werden, welche auch dann erst bezahlt werden müssen.

Die Überwachung und Steuerung des Bedarfs wird von den Cloud-Computing-Anbietern übernommen. Der Cloud-Computing-Kunde kann beispielsweise Schwellwerte für Bearbeitungszeiten oder CPU-Auslastungen einstellen. Sollten die Werte unter- oder überschritten werden, wird die Ressourcenanzahl angepasst. Diese Überwachung und Steuerung geschieht zur Zeit zentral und wird vom Anbieter übernommen. Die Überwachungssysteme verschiedener Cloud-Computing-Anbieter arbeiten nicht zusammen. Daher ist das gleichzeitige Überwachen von Anwendungen, die auf den Systemen mehrerer Cloud-Computing-Anbietern laufen, mit solchen proprietären Systemen nicht möglich. Das Verteilen der Anwendung auf mehrere Cloud-Anbieter ist für Internetseitenbetreiber, die eine sehr hohe Verfügbarkeit erhalten wollen, aber notwendig. Daher ist es oft nötig, die Überwachungsstruktur der Anwendung selbst zu implementieren und damit die Unabhängigkeit von einem bestimmten Cloud-Computing-Anbieter zu erhalten. Ein dezentraler Ansatz ist hier ebenfalls interessant, da beim Ausfall eines Anbieters die Internetanwendung weiterhin erreichbar sein sollte.

1.1. Ziele der Arbeit

Die Organic-Computing-Initiative untersucht die immer komplexer werdenden Computersysteme und will diese mit Techniken aus den Bereichen Biologie, Lernalgorithmen und Emergenz managen. Hauptaugenmerk liegt auf dezentralen selbst-verwaltenden Systemen.

In dieser Arbeit wird das Management eines sehr großen Cloud-Computing-Systemes, bestehend aus tausenden von Instanzen, mit Techniken aus dem Organic-Computing erweitert. Durch die komplette dezentrale Implementierung existiert kein *Single Point of Failure* und das System ist robuster. Zusätzlich wird die Skalierbarkeit verbessert, da keine Komponenten eine globale Sicht auf das System benötigen.

1.2. Forschungsbeitrag

Um die Verwaltung komplett dezentral zu realisieren, müssen zuerst die Daten der einzelnen Komponenten in alle anderen Komponenten verteilt werden. Daher werden in dieser Arbeit Informationsverteilungsalgorithmen vorgestellt und bewertet. Durch die Nutzung von Peer-to-Peer-Algorithmen wird eine komplett dezentral arbeitende Informationsverteilungsstruktur implementiert. Durch die vorgestellten Verteilungsalgorithmen kann eine komplette Verteilung im Netz garantiert werden, allerdings sind die Informationen nicht immer aktuell.

Anschließend werden Steuerungsalgorithmen vorgestellt, die mit diesen, teilweise ungenauen Informationen, das System verwalten können. Es wird gezeigt, dass dezentrale Steuerungen geeignet sind, um große verteilte Systeme zu verwalten.

Da Cloud-Computing-Systeme sehr große verteilte Systeme sind, können die in dieser Arbeit vorgestellten Methoden auch auf andere verteilte Systeme mit tausenden von Komponenten angewendet werden.

1.3. Aufbau der Arbeit

Im nächsten Kapitel werden die Grundlagen des Cloud-Computing erörtert. Zusätzlich werden die Autonomic- und Organic-Computing-Initiativen vorgestellt, welche sich die Vereinfachung des Administrationsaufwands zum Ziel gesetzt haben.

Überwachung in verteilten Systemen ist komplex, da kein globales Wissen vorhanden ist. Um ein verteiltes System zu verwalten, müssen Informationen über andere Teile des Systems vorhanden sein. In Kapitel 3 wird eine neue Art der Informationsverteilung auf Grundlage von Peer-to-Peer Overlay Netzwerken vorgestellt. Selbst organisierende verteilte Systeme benötigen eine effiziente Informationsverteilung um allen Komponenten eine möglichst akkurate Sicht auf das System zu geben. Die Informationsverteilung mit Peer-to-Peer Netzwerken stellt allen Knoten eine komplette Sicht auf das System zur Verfügung. Mit diesen Informationen kann jede Komponente das verteilte System selbstständig steuern.

Die auf Peer-to-Peer-Algorithmen basierenden Informationsverteilungsalgorithmen garantieren, dass Informationen an alle Komponenten innerhalb einer festen oberen Zeitschranke verteilt werden. Allerdings führt diese Verzögerung zu unterschiedlichen Informationen auf verschiedenen Komponenten. In Kapitel 4 wird untersucht, wie aus den durch die Peer-to-Peer-Verteilungsalgorithmen gewonnenen Informationen die Steuerung des Systems möglich ist. Dabei werden zwei dezentrale Steuerungsalgorithmen untersucht und mit zentralen Ansätzen verglichen. Zusätzlich wird der Einfluss von aggregierten Informationen auf die Steuerung untersucht.

Kapitel 5 fasst die vorliegende Arbeit zusammen.

Im Anhang A werden die Rohdaten der im Kapitel 4 durchgeführten Evaluation aufgelistet.

2. Grundlagen

In diesem Kapitel wird eine Einführung in Cloud-Computing als Anwendungsbeispiel für ein großes verteiltes System gegeben. Durch Cloud-Computing haben sich auch die Anforderungen an moderne Rechenzentren geändert. Die neuen Anforderungen und der daraus resultierende Aufbau der Rechenzentren wird danach gezeigt. Anschließend wird Autonomic und Organic Computing vorgestellt und die in der Arbeit verwendeten Konzepte diskutiert. Das Kapitel endet mit einem Überblick über verwandte Arbeiten.

2.1. Grundlagen des Cloud-Computing

Dr. Werner Vogels, CTO von Amazon, beschreibt in einer Vorlesung an der Stanford University[68], wie Amazon durch das schnelle Wachstum gezwungen war, auf eine neue service-orientierte Software umzustellen. Später mussten die verschiedenen Services unabhängig voneinander verwaltet werden. Der Ausfall eines Rechenzentrums sollte keine Beeinträchtigung der Performance eines Seitenbesuchers haben. Diese Anforderung konnte nur durch Virtualisierung eingehalten werden. Dabei werden die vorhanden Ressourcen (CPU, Festplattenplatz, Arbeitsspeicher, Netzwerkanbindung) auf virtuelle Rechner aufgeteilt. Sollte physikalisch Hardware ausfallen, können die virtuellen Maschinen auf andere Rechner verschoben werden. Um aus diesem Wissen und dem vorhandenen Rechenzentrum Gewinn zu erzielen, wurde der Dienst für Kunden geöffnet. Damit wurde *Amazon Web Services*, oder kurz *AWS*, der erste große Cloud-Computing-Anbieter.

Die US Standardisierungsbehörde National Institute of Standards and Technology veröffentlichte im September 2011 eine Definition von Cloud-Computing[42]. Darin werden drei verschiedene Virtualisierungsstufen beschrieben.

Infrastructure as a Service (IaaS) IaaS ist das Bereitstellen von virtuellen Ressourcen, z.B. CPU-Zeit, Speicher oder Netzwerkbandbreite. Der Kun-

de kann beliebige Software auf den Ressourcen ausführen und muss sich um die Verwaltung der virtuellen Ressourcen selbst kümmern. Es besteht kein Zugriff auf die echte Hardware des Rechenzentrums, sondern nur auf die virtuellen Ressourcen. Beispiele für IaaS sind *Amazons Elastic Computing Cloud*[1] oder *Google Compute Engine*[2].

Platform as a Service (PaaS) Der Kunde entwickelt die Software die auf dem bereitgestellten Software-Stack läuft. Er hat keinen Zugriff auf die Hardware und das installierte Betriebssystem. Beispiele für PaaS sind *Amazon Elastic Beanstalk*[3] und *Google App Engine*[4].

Software as a Service (SaaS) Der Kunde benutzt die Software, meistens eine Internetanwendung, die bereitgestellt wird. Es können keine Änderungen an der Soft- oder Hardware vorgenommen werden. Zu SaaS zählen *Content Management Systeme* und alle Arten von Online-Anwendungen. Viele Internetdienste sind SaaS, dazu zählen Soziale Netzwerke (z.B. Facebook) und Webmail-Anbieter. Auch viele bekannte Programme werden inzwischen als SaaS Version angeboten, dazu zählen Microsoft Office oder Adobe Photoshop.

Die drei Stufen bauen aufeinander auf. Es kann eine SaaS-Anwendung auf einem PaaS-System erstellt werden. Um eine PaaS zu implementieren, muss eine IaaS-Architektur vorhanden sein. Abbildung 2.1 zeigt die sogenannte Cloud-Computing-Pyramide, die oft benutzt wird um die Abhängigkeiten darzustellen.

Die Begriff *Cloud* wird heute benutzt um zu beschreiben, dass Ressourcen im Internet benutzt werden. Diese Ressource kann Speicherplatz (Cloud-Storage) oder Rechenleistung (Cloud-Computing) sein. Die Ressourcen können innerhalb von Minuten bereitgestellt werden. Dabei wird die Illusion von unbegrenzten Ressourcen erzeugt[3].

Es werden nur die benutzten Maschinen bezahlt. Meistens wird pro angefangener Stunde abgerechnet. Die Kosten, eine Berechnung auf einer Maschine 1000 Stunden laufen zu lassen, ist identisch mit dem Preis für 1000 Maschinen, die eine Stunde laufen. Daher sind die Grenzen für die Skalierbarkeit nur die Grenzen der Parallelität des Algorithmus. Es haben sich Verfahren, wie

[1] `https://aws.amazon.com/ec2/`
[2] `https://cloud.google.com/products/compute-engine/`
[3] `https://aws.amazon.com/elasticbeanstalk/`
[4] `https://developers.google.com/appengine/`

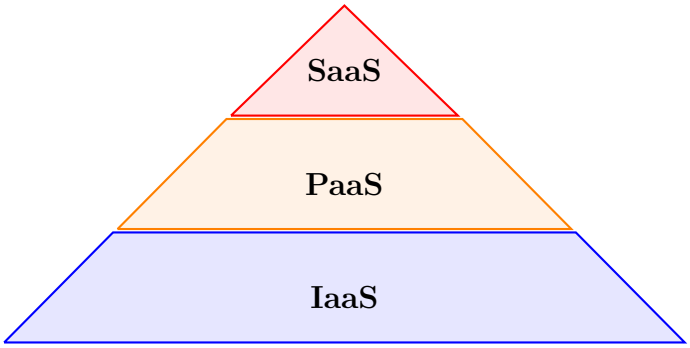


Abb. 2.1.: Cloud-Computing-Pyramide

z.B. Map-Reduce[16], entwickelt, die selbst große Datenmengen im Exabyte-Bereich effizient bearbeiten können.

Die Hardware eines Internetdienstes muss so ausgelegt sein, dass alle Benutzer, auch während den Spitzenzeiten, den Dienst ohne Verzögerung nutzen können. Falls diese Spitzenzeiten nur einige Tage im Monat auftreten, wird die Hardware die meiste Zeit nicht ausgenutzt. Cloud-Computing hilft durch die dynamische Ressourcenanpassung Kosten zu senken, da nur die gerade benötigten Ressourcen bezahlt werden müssen. Bei einem Anstieg des Bedarfs, können innerhalb von wenigen Minuten neue virtuelle Maschinen gestartet werden.

Cloud-Computing-Anbieter, wie beispielsweise Amazon, Google oder Mircosoft, besitzen große Rechenzentren und haben Wissen über die Verwaltung solcher Rechenzentren aufgebaut, da beides für das eigene Geschäftsmodell notwendig ist. Jetzt bieten sie Cloud-Computing-Dienste an, um von der vorhandenen Infrastruktur und dem angehäuften Wissen zu profitieren.

Kunden dieser Dienste sind meistens Entwickler oder Administratoren, die mithilfe von Cloud-Computing wiederum Dienste für Kunden oder Mitarbeiter bereitstellen. Dafür werden Rechenzeit, Speicher und Netzwerkbandbreite vom Entwickler beim Cloud-Computing-Anbieter angemietet.

In der vorliegenden Arbeit wird Cloud-Computing näher untersucht und der Cloud-Computing-Kunde näher betrachtet. Dabei wird eine Internetanwendung analysiert, die von Entwicklern bereitgestellt wird. Benutzer der Internetseite, welche Kunden der Entwickler sind, erwarten eine schnelle Reaktionszeit der Online-Anwendung. Die Entwickler müssen die Erwartungen der Kunden erfüllen, da diese ansonsten den Dienst nicht mehr nutzen. Zusätzlich sollen aber nicht zu viele Ressourcen gemietet und damit unnötige Kosten

verursacht werden. Mit Hilfe der dezentralen Steuerung soll der Ressourcenbedarf minimiert werden, ohne die Servicequalität des Internetseitenbesucher zu beeinflussen.

2.2. Warehouse-Scale-Computers

Durch Cloud-Computing hat sich die Anforderung an Rechenzentren verändert. Früher war ein Rechenzentrum eine Ansammlung von verschiedenen Computern, wobei jeder Rechner seine Aufgabe hatte und wenig Interaktion zwischen den Servern stattfand. Inzwischen werden Anwendungen nicht auf einzelne Server sondern auf einem Verbund aus mehreren Servern ausgeführt. Die Anwendung interagiert mit vielen Servern im Rechenzentrum. Zur Berechnung wird also ein Verbund aus allen Servern benutzt, daher müssen diese viel enger zusammenarbeiten als bisher üblich. Deshalb kann das komplette Rechenzentrum als ein Computer betrachtet werden. Im Englischen wird dies als *Warehouse-Scale-Computer* bezeichnet. Diese neue Verarbeitungsmethode führt zu veränderten Anforderungen an Rechenzentren.

Luiz Andre Barroso und Urs Hölzle beschreiben in [5] diese neuen Anforderungen an solche Rechenzentren. Die beiden Autoren helfen Google bei der Einrichtung von Rechenzentren. Da dieser Trend noch recht neu ist, wird in diesem Kapitel eine kurze Einführung in diese Warehouse-Scale-Computers auf Grundlage des Buches gegeben.

Durch die Größe der Rechenzentren können bei der Hardwarebeschaffung die Preise auf $\frac{1}{5}$ bis $\frac{1}{7}$ gesenkt werden[26, 25]. Zusätzlich sinkt die Anzahl der Administratoren pro Computer und die Entwicklungskosten für Software [5].

Ein wichtiger Aspekt sind die Energiekosten, da diese für den Betrieb der Computer und deren Kühlung benötigt werden. Daher werden große Rechenzentren häufig an Orten mit niedrigem Strompreis gebaut. Durch eine niedrige Außentemperatur können Kühlungskosten gesenkt werden.

Der Energieverbrauch von Servern ist nicht proportional zu ihrer Auslastung [6]. Abbildung 2.2 zeigt die durchschnittliche Auslastung von 5000 Servern über einen Zeitraum von 6 Monaten. Es ist deutlich erkennbar, dass die Auslastung häufig zwischen 30% und 40% liegt und sehr selten größer als 50% wird. Dieses Verhalten ist teilweise gewollt. Die Auslastung der Server ist so berechnet, dass

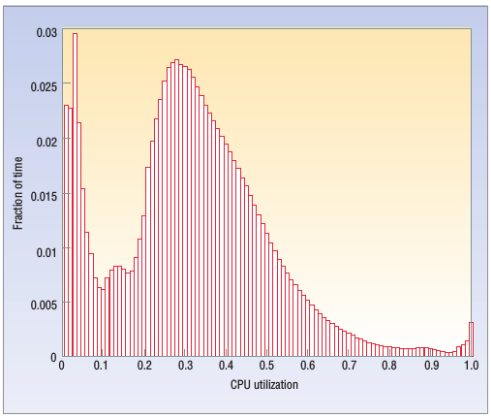


Abb. 2.2.: CPU Auslastung von 5000 Servern über 6 Monate aus [6]

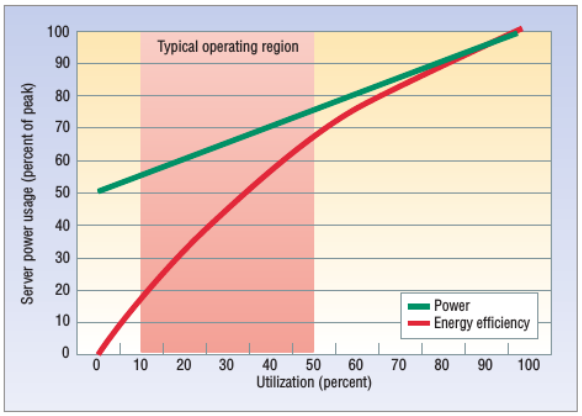


Abb. 2.3.: Energieeffizienz und Auslastung von Servern aus [6]

Lastspitzen nicht genau 100 % Auslastung erzeugen. Dies würde zu Problemen bei zusätzlichen Schwankungen oder Hardwareausfällen führen.

Abbildung 2.3 stellt die Serverauslastung und die Energieaufnahme gegenüber. Die grüne Kurve gibt die Energieaufnahme im Verhältnis zur maximalen Aufnahme an. Selbst bei einer sehr geringen Auslastung werden bereits 50% der Energie benötigt. Diese steigt linear an, bis bei voller Auslastung die maximale Energie aufgenommen wird. Die rote Linie zeigt die Energieeffizienz. Dabei wird die Auslastung durch die aufgenommene Energie geteilt. Wie vorher gezeigt, arbeiten viele Server mit einer Auslastung zwischen 15% und 45%. Der rot hinterlegte Bereich entspricht diesem Arbeitsbereich. Die Energieeffizienz liegt hierbei zwischen 20% und 70%.

Mit der CPU-Auslastung kann der Energieverbrauch eines Servers sehr gut abgeschätzt werden [21]. Durch große Rechenzentren mit vielen verschiedenen Anwendungen können die Server besser ausgelastet werden, was zu einer bessern Energiebilanz führt. Dadurch können Rechenzentrumsbetreiber durch das Anbieten von Cloud-Diensten ihre Ressourcen effizienter nutzen.

Abbildung 2.4 zeigt den Aufbau eines Netzwerks in einem *Warehouse-Scale Computers* Rechenzentrum wie er in [5] beschrieben ist. Viele Computereinschübe werden in einem Rack montiert. Die Rechner des Racks werden über einen schnellen Switch verbunden. Diese Racks werden zu Clustern zusammengefasst. Innerhalb des Clusters werden die Rack-Switches mit dem Cluster-

Abb. 2.4.: Netzwerkaufbau eines *Warehouse-Scale Computers* aus [5]

Switch verbunden. Da die Cluster-Switches mehr Ports benötigen als die Rack-Switches, sind diese teurer. Es ist nicht rentabel die gleiche Geschwindigkeit für beide Switches zu wählen. Statt dessen werden die Cluster mit einer langsameren Bandbreite angeschlossen. In einem Rechenzentrum werden mehrere dieser Cluster aufgebaut, so dass mehrere zehntausend Server pro Rechenzentrum verbunden sind.

Im *Open Compute Project*[5] stellen Ingenieure von Facebook seit 2011 die Spezifikationen ihrer Rechenzentren für Andere zur Verfügung. Die Idee ist von *Open Source Software* übernommen. Durch die offenen Standards sollen andere Unternehmen in der Lage sein, selbst Rechenzentren nach dem energiesparenden Modell zu bauen. Sollten diese Unternehmen weiter Verbesserungen finden, so werden diese in die Spezifikation übernommen und stehen anderen Rechenzentrenbetreibern kostenlos zur Verfügung.

Die in dieser Arbeit verwendeten Evaluationsumgebungen benutzen die im *Open Compute Project* und von Luiz Andre Barroso und Urs Hölzle beschriebene *Warehouse-Scale Computer*-Netzwerkstruktur.

[5] `http://www.opencompute.org`

2.3. Autonomic Computing

2001 beschrieb Horn [31] wie durch Automatisierung die immer komplexer werdenden Computersysteme verwaltet werden können. Seine Vision von zukünftigen Computersystemen basiert auf dem autonomen Nervensystem von Säugetieren. Wenn ein Mensch beispielsweise zu rennen beginnt, werden Herzschlag und Atmung erhöht. Diese Anpassungen finden automatisch und ohne bewusstem Zutun des Menschen statt. Dabei prägte Horn den Begriff *Autonomic Computing* (AC) und führte die Selbst-X-Eigenschaften ein. Die Idee von Horn war, Verwaltungsaufgaben von Computern ebenfalls zu automatisieren, damit diese ohne Benutzerinteraktion ablaufen. Autonomic Computing Systeme sollen sich selbst verwalten.

2003 schrieb Kephart [34] über die konkretere Umsetzung. Er führte eine Überwachungsstruktur ein und konkretisierte die Selbst-X-Eigenschaften. Die von ihm genannten Eigenschaften sind:

Selbst-Konfiguration Installieren und Konfigurieren eines großen Systems kann heute mehrere Tage oder Wochen dauern. Die Selbst-Konfiguration soll diese Aufgaben nach Kepharts Idee selbstständig ausführen und wird dabei von High-Level Regeln geleitet.

Selbst-Optimierung Große Systeme haben viele Parameter, die für die Performance entscheidend sind. Diese korrekt einzustellen, benötigt viel Zeit und Fachwissen. Müssen verschiedene Programme zusammenspielen, steigt die Komplexität. Ein AC-System erkennt, welche Parameter für die Performance entscheidend sind und optimiert diese automatisch.

Selbst-Heilung Fehlersuche und Reparatur in großen Systemen ist ebenfalls sehr aufwändig. Das Zusammenspiel vieler Komponenten kann zu sporadischen Fehlern führen. Diese sind schwer zu erkennen, da diese erst reproduziert werden müssen um sie zu beheben. Oft verschwinden solche Fehler auch wieder, ohne dass ein Grund gefunden wurde. Mit Autonomic Computing werden solche Fehler beim ersten Auftreten erkannt und automatisch nach der Ursache gesucht. Sobald diese gefunden wurde, kann der Fehler entweder selbstständig behoben oder einem Programmierer eine genaue Beschreibung gesendet werden.

Selbst-Schutz Die Anzahl der Angriffe auf Computersysteme steigt ständig. Trotz Firewalls und Intrusion-Detection-Systemen gelingt es immer wie-

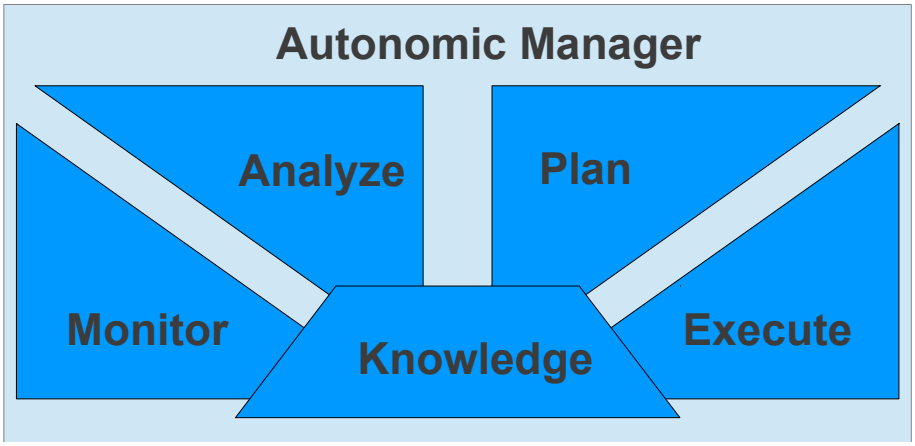


Abb. 2.5.: Der Autonomic Manager mit MAPE-Zyklus nach Kephart [34]

der, Angreifern wichtige Daten zu entwenden oder Dienste zu stören. Bei komplexen Computersystemen ist es schwer, alle bekannten Angriffsszenarien zu testen. Zusätzlich werden oft unbekannte Lücken mit neuen Angriffen ausgenutzt. AC-Systeme werden Angriffe abwehren. Sie werden Angriffe erkennen und dann das komplette System davor schützen. Zusätzlich werden Probleme anhand von Fehlermeldungen erkannt und behoben.

In seiner Arbeit wählt Kephart einen hierarchischen Ansatz. Jedes zu steuernde Element, von Kephart als *Autonomic Element* bezeichnet, besitzt einen *Autonomic Manager*, der für die Steuerung verantwortlich ist. Die Umsetzung der Selbst-X-Eigenschaften wird laut Kephart von einem *Autonomic Manager* übernommen. Dieser Manager überwacht das System. Bild 2.5 zeigt den Aufbau des *Autonomic Managers*. Sollte das System nicht optimal laufen, so wird dies erkannt und das Systemverhalten geändert. Um dies zu erreichen ist der *Autonomic Manager* in vier Bereiche unterteilt:

Monitor Diese Phase ist verantwortlich für die Überwachung des Systems.

Analyze Die gesammelten Daten werden analysiert, um Muster zu erkennen und zukünftiges Verhalten vorher zusagen.

Plan Die analysierten Daten werden benutzt, um zu entscheiden wie das System beeinflusst werden muss um ein bessere Leistung zu erzielen.

Execute Die letzte Phase führt die Veränderungen am System durch.

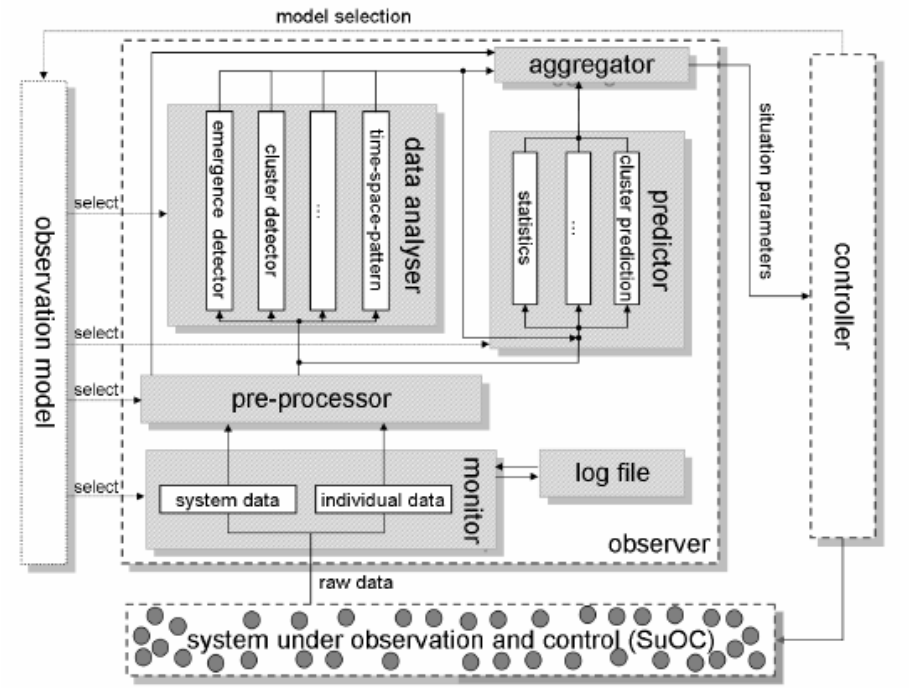


Abb. 2.6.: DerObserver/Controller nach Richter et al. [54]

2.4. Organic Computing

Aus der Idee des Autonomic Computing entstand die Idee des *Organic Computing*. In einem Positionspapier [1] präsentieren die Gesellschaft für Informatik (GI) und die Informationstechnische Gesellschaft im VDE (ITG) ihre Vision von *Organic Computing*. Die *Organic Computing Initiative* hat sich zum Ziel gesetzt, die immer steigende Komplexität in Computersystemen beherrschbar zu machen, in dem diese Systeme nach dem Vorbild der Natur angepasst werden. Es sollte nicht mehr notwendig sein, alle möglichen Probleme während der Entwicklungszeit zu kennen und Reaktionen darauf festzulegen. Statt dessen sollen *Organic-Computing-Systeme* selbstständig während des Betriebs auf Probleme reagieren können und entscheiden, wie das Problem gelöst wird. Damit zukünftige Computer Systeme weiterhin beherrschbar bleiben, müssen diese Systeme intelligenter werden und sich selbst überwachen.

Abbildung 2.6 zeigt die *Observer/Controller-Struktur*, die bei der Überwachung und Steuerung von *Organic-Computing-System* benutzt werden soll.

Diese ähnelt dem MAPE-Zyklus aus dem Autonomic-Computing. Allerdings wird beim Observer/Controller die Umsetzung detaillierter beschrieben. So sollen hier durch verschiedene Analysemethoden Daten aufbereitet und durch einen *Predictor* zukünftiges Verhalten vorausgesagt werden.

Organic Computing befasst sich mit dem Auftreten von Emergenz [44]. Dabei wird untersucht, wie ein großes System durch kleine Änderungen und ohne globales Wissen zielführend beeinflusst werden kann. Ein Beispiel für solch ein emergentes Verhalten stellt der Ameisenalgorithmus [19] dar. Ameisen hinterlassen auf ihrem Weg vom Bau zur Futterstelle Duftstoffe. Der Duft dieser Pheromone lässt nach einiger Zeit nach. Sollten noch keine Duftstoffe vorhanden sein, suchen die Ameisen zufällig einen Weg. Die Konzentration erhöht sich auf dem kürzeren Weg schneller, da die Ameisen häufiger den Weg passieren. Durch die höhere Konzentration wählen nach einiger Zeit alle Ameisen den kürzeren Weg. Dadurch konnte nur durch Pheromon ein kürzerer Weg gefunden werden, ohne dass die Ameisen explizit danach gesucht haben.

2.5. Verwandte Ansätze

Es gibt viele Arbeiten die sich mit dem verteilten Steuern von Systemen befassen. In diesem Abschnitt wird ein Überblick über relevante Arbeiten gegeben. Als Erstes werden Arbeiten aus dem Organic Computing vorgestellt, die sich mit der dezentralen Steuerung von verteilten Systemen befassen.

Julia Schmitt [61] untersuchte den Einsatz eines automatischen Planers zur Steuerung eines Organic Computing Systems. Schmitt erweiterte OCμ (kurz für Organic Computing Middleware for Ubiquitous Environments)[55] um einen automatischen Planer. Mit dem Planer wurden Ziele für Selbstheilung, -optimierung und -konfiguration untersucht. Das untersuchte OC-System waren mehrere mit OCμ betriebene PCs. Die PCs führen verschiedene Services aus. Hierbei wurden Services vom Planer gestartet, gestoppt oder auf andere Knoten verschoben. Aufgabe des automatischen Planers war es, die Last auf den PCs gleichmäßig zu verteilen und beim Ausfall eines PCs die dort laufenden Services auf den anderen Knoten erneut zu starten. Als Planer wurde der JavaFF-Planer benutzt. Hierbei handelt es sich um die Implementierung des Fast Forward Algorithmus [29]. Der Fast-Forward-Algorithmus war der schnellste Planungsalgorithmus auf der AIPS-2000 Planning System Competi-

tion [4]. Ein automatischer Planer besitzt eine hohe Laufzeit, liefert dafür aber immer korrekte Ergebnisse. Um das System schnell in einen stabilen Zustand zu überführen, implementierte Schmitt einen Reflex Manager [62] als einen Cache für Pläne. Damit konnte die Geschwindigkeit enorm gesteigert werden. Bei steigender Systemgröße verlängert sich auch die Planungszeit exponentiell und die vielen verschiedenen Pläne erhöhen die Anzahl der Cache Misses. Daher ist dieser Ansatz nicht für Systeme mit mehr als hundert Knoten geeignet.

Das *Artificial Hormon System* (AHS) [10] platziert Tasks auf Knoten mit Hilfe künstlicher Hormone. Diese werden als Nachrichten realisiert. AHS wurde wie OCμ im Organic Computing Schwerpunktprogramm entwickelt. Bei AHS liegt der Fokus allerdings auf der Verwendung in eingebetteten Systemen. Für jede Task gibt es Unterdrücker (suppressors) und Beschleuniger (accelerators). Ein Knoten bewertet, wie gut dieser geeignet ist, den Task auszuführen. Sollte der Knoten gut geeignet sein, werden Unterdrücker gesendet um eine ungewollte doppelte Platzierung zu vermeiden. Falls der Knoten die Task nicht übernehmen will, werden Beschleuniger übermittelt, damit andere Knoten die Task übernehmen. Hierbei senden alle Komponenten Hormone an alle anderen Teilnehmer. Die Hormone werden als Nachrichten an andere Knoten implementiert. Aufgrund der vielen Nachrichten ist diese Methode für die in dieser Arbeit betrachteten Systeme mit tausenden Knoten nicht geeignet.

Im Projekt *Self-Organizing Smart Camera System* [30, 33] werden Kameras selbständig ausgerichtet, um einen möglichst großen Bereich zu überwachen. Die Smart-Kameras können das Objektiv schwenken, die Neigung ändern und zoomen. Zusätzlich verfügen Sie über einen Prozessor, Speicher und können miteinander kommunizieren. Jede Kamera kennt ihre Position, z.B. über GPS, sowie die Position von Hindernissen in ihrem Sichtfeld. Benachbarte Kameras tauschen Informationen über ihre Position und ihr Sichtfeld aus. Mit den Smart-Kameras kann ein großer Bereich überwacht werden, wobei die Sichtfelder so wenig wie möglich überlappen um eine große Abdeckung zu erhalten. Sollte ein auffälliges Ereignis beobachtet werden, können die Sichtfelder angepasst werden um das Vorkommnis aus möglichst vielen Blickwinkeln zu sehen. Es wird angenommen, dass eine möglichst kleine Überlappung einer Kamera mit ihren Nachbarn zu einem großen Überwachungsbereich führt. Daher versucht jede Kamera eine möglichst geringe Überlappung mit den Sichtbereichen benachbarter Kameras zu erreichen. Die Evaluierung mit 350 Smart-Kameras zeigt, dass durch dieses lokale Verhalten eine Lösung gefunden wird, die sehr

nahe an der optimalen liegt. Auf den Ausfall einzelner Kameras passt sich das System schnell an. Durch lokale Optimierung wird versucht ein globales Optimum zu erzielen. Bei der Problemstellung der Smart-Kameras ist dieses Vorgehen erfolgreich. Auf das in dieser Arbeit untersuchte Problem der Anpassung einer Cloud-Computing-Infrastruktur an die aktuellen Nutzung kann keine lokale Optimierung benutzt werden. Es ist notwendig, die Ausnutzung der vorhanden Ressourcen über die komplette Infrastrukur zu ermittelt. Mit diesem Wert kann die Auslastung des Systems bestimmt und der Ressourcenbedarf ermittelt werden.

Mit *Organic Computing Traffic Control* [67, 48], kurz OCTC, werden Verkehrsampeln mit Hilfe von Organic-Computing-Techniken gesteuert. Hierbei werden dezentral von jeder Ampel unter Verwendung von Learning-Classifier-Systemen mit genetischen Algorithmen die Schaltzeiten optimiert. Das System simuliert die Auswirkungen neuer Regeln auf den Verkehr. Wenn die Regeln in der Simulation zu einer Verbesserung führen, werden die neuen Regeln in das echte System übernommen. Durch diesen zweistufigen Ansatz wird verhindert, dass Regeln angewendet werden, die zu einer Verschlechterung führen. Jede Ampel ermittelt einen Schaltzyklus, der für den beobachteten Verkehrsfluss optimal ist. Da keine Koordination zwischen verschiedenen Ampeln statt findet, können alle Ampeln eine optimal Schaltzeit besitzen, aber eine *Grüne Welle* kann nicht erstellt werden. In [66] wird der dezentrale Ansatz erweitert um die Steuerung mehrere Ampeln zu koordinieren. Durch diese Koordination können *Grüne Wellen* erzielt und der Verkehrsfluss weiter optimiert werden. Bei der verteilten Cloud-Computing-Steuerung sind die Lastinformationen aller Knoten notwendig um den Ressourcenbedarf zu ermitteln. Der bei OTC benutzte Ansatz kann daher nicht auf die Cloud-Computing-Steuerung übernommen werden. Hierbei kann sich die Anzahl der zu überwachenden Instanzen häufig ändern und die Koordination aller Teilnehmer wird bei Systemen mit über tausend Teilnehmern zu langsam, da der OTC Ansatz nicht skaliert.

Die *Organic Robot Control Architecture* (ORCA) [32, 43, 11] steuert mit einer hierarchischen Observer/Controller-Architekur autonome Roboter. Ein Roboter besteht aus zwei Arten von Überwachungs-/Steuerungs-Einheiten. Die *Organic Computing Units* (OCUs) überwachen den Roboter, so wird beispielsweise überprüft, ob ein Servomotor die Stellung des Beins korrekt ändert. Die *Basic Computing Units* (BCUs) steuern die einzelnen Komponenten des Roboters. Jede BCU und OCU kennt die Kompetenten, die gesteuert bzw.

überwacht werden. Diese Informationen werden an die anderen Einheiten weitergeleitet. Die Einheiten bilden Kommunikationsverbindungen untereinander. Dabei werden Verbindungen hergestellt, wenn Daten für die Erfüllung ihrer Aufgabe notwendig sind. Zusätzlich wird eine hierarchische Überwachung aufgebaut. So kann beispielsweise eine OCU die Überwachung aller sechs Beine übernehmen und baut daher sechs Verbindungen zu den entsprechenden OCUs der Beine auf. Beim Ausfall einer Komponente kann eine Neugruppierung eingeleitet werden. Sollte eine OCU ein Fehlverhalten feststellen, so wird versucht dieses mit den bekannten BCUs zu kompensieren. Wenn dies nicht gelingt, wird der Fehler an eine Einheit mit mehr Verbindungen weitergeleitet. Die ORCA kann nicht für die Cloud-Computing-Steuerung übernommen werden, da die Anzahl der Cloud-Computing-Instanzen sich häufig ändert und für die Steuerung immer die Informationen aller Komponenten benötigt werden. Zusätzlich ist die vorgestellte Architektur nicht auf tausende Komponenten ausgelegt, was den Kommunikationsaufwand enorm vergrößern würde.

Auch außerhalb der Organic Computing Forschergemeinschaft findet das Thema der dezentralen Steuerung von großen verteilten Systemen Beachtung.

Die *Tuples On The Air* (TOTA)-Middleware [40, 74] benutzt *spatial computing*. Bei *spatial computing* wird jeder Knoten in einem n-dimensionalen Raum positioniert. Ein Knoten ist zuständig für den umliegenden Raum und kennt die Knoten, die für den angrenzenden Raum zuständig sind. Informationen werden in Tupel gesendet. Jeder Knoten empfängt Tupel von seinen Nachbarn, ändert die Informationen und leitet diese Tupel weiter. Auf diese Weise werden Informationen an alle Knoten verteilt.

CARISMA [46] ist eine service-orientierte Middleware für eingebettete Echtzeitsysteme. Es werden Dienste auf verschiedene Knoten verteilt, um eine gleichmäßige Auslastung zu erreichen. Jeder Knoten benutzt lokale Informationen um zu bestimmen, wie gut Dienste auf ihnen ausgeführt werden können. Basierend auf diesen Berechnungen werden die Dienste an Knoten in einem Auktionssystem verteilt. Die Middleware stellt die Grundlage für die Kommunikation zur Verfügung. Jeder Dienst optimiert sich selbst, was zu einem optimalen Gesamtsystem führt. Die Dienste verarbeiten die Knotenbewertungen und wählen die Knoten aus, auf denen sie platziert werden.

CarSOC [36, 35] benutzt ein zweistufiges Verfahren um eingebettete Kontrolleinheiten eines Autos (Embedded Control Unit oder ECU) zu steuern. Auftretende Fehler sollen möglichst schnell und nah an der Ursache erkannt

und behoben werden. Dazu besitzt jede Komponente einen MAPE-Zyklus zur Steuerung von Teilaufgaben. Dieser kann auch unvollständig sein und nur bestimmte Teile enthalten. Es wurde eine hierarchische Überwachungsstruktur implementiert.

2009 stellten Marinos und Briscoe das *Community Cloud-Computing*[41] vor. Hierbei handelt es sich um eine Vereinigung der Konzepte aus dem verteilten Rechnen (engl. *distributed computing*) mit dem Cloud-Computing. Ziel ist es eine verteilte Cloud zu erstellen. Es wird nur das Konzept, aber keine konkrete Implementierung vorgestellt, da noch viele Fragen offen sind. Die in dieser Arbeit vorgestellten Methoden zur dezentralen Steuerung können in das Konzept des Community Cloud-Computing integriert werden.

Openstack [63], eine offene Verwaltungssoftware für Cloud-Computing-Rechenzentren, benutzt ein Message-Queue-System, wie z.B. rabbitMQ[6], zur Steuerung und Überwachung der Cloud. Einzelne Instanzen können Meldungen in die Message-Queue schreiben. Steuerungsrechner lesen die Queue und bearbeiten die Anfragen. Die Queue-Server müssen redundant vorhanden sein, um einen Ausfall zu verhindern.

Es existieren mehrere dezentrale Algorithmen, die Probleme des Cloud-Computing lösen. Im folgenden werden verteilte Algorithmen für die Platzierung von Services auf Rechner vorgestellt.

Bubble-Flux [72] optimiert die Antwortzeiten von Cloud-Anwendungen durch die Platzierung auf möglichst wenige virtuellen Maschinen. Dabei wird das Verhalten der Anwendungen untersucht und ermittelt, welche Anwendungen zusammen auf einer virtuellen Maschine laufen können, ohne sich gegenseitig stark zu beeinflussen. Die Anforderungen der Anwendungen werden während des Betriebs gemessen. Die Messwerte werden von einem Scheduler benutzt um die Platzierung der Anwendungen zu bestimmen. Der Scheduler läuft nicht im Netz verteilt und stellt daher einen Single-Point-of-Failure da.

Randles et al.[51, 45] untersuchten die bei der Futtersuche von Honigbienen verwendete Tanzsprache (engl. honey-bee foraging). Je nach Wetterlage und Standort kann die Menge an Blütenstaub, die von einer Blume produziert wird, stark schwanken. Beim Futtersuche-Problem müssen die Arbeiterbienen die Blumen mit viel Blütenstaub ausfindig machen und diesen effizient sammeln. Wenn eine Biene von einer Blüte zu einer andern fliegt, vergeht Zeit, die

[6] `www.rabbitmq.com`

nicht zum Sammeln benutzt werden kann. Arbeiter sollten also nur die Blüten wechseln, wenn der Zeitausfall durch mehr Ernte ausgeglichen wird. Wenn Arbeiter zum Bienenstock zurück kehren, melden sie dort den Standort und die Rentabilität der Blüte mit dem sogenannten *Schwänzeltanz*[7] an andere anwesende Arbeiter. Diese Bienen entscheiden nun, wohin sie als nächstes fliegen. Arbeiterbienen erfahren nur von Blüten, wenn ein Arbeiter gerade am Bienenstock eintrifft und seine Informationen mitteilt. Dadurch besitzen sie kein globales Wissen. Dieser Algorithmus kann auf Computersysteme übernommen werden. Dabei entsprechen die Arbeitsbienen Servern, die Anfragen einer bestimmten Anwendung abarbeiten. Die Zeit, die zum Bearbeiten einer Anfrage benötigt wird, entspricht der Flugzeit zur Blüte. Jede Anwendung hat einen Eingangsspeicher. Je nach Anzahl der Anfragen und Bearbeitungszeit werden mehr oder weniger Server für eine Anwendung benötigt. Mit dem Futtersuche-Algorithmus werden Informationen nach der Abarbeitung einer Anfrage über ein *Advertboard* geteilt und daraus entschieden, ob Server umverteilt werden müssen. In [45] wurde untersucht, wie sich drei dezentrale Algorithmen bei der Verteilung von 2, 3 und 4 Anwendungen verhalten. Es wurde gezeigt, dass der Futtersuche-Algorithmus den höchsten Durchsatz erzielt.

Di Nitto et al. untersucht in [18, 17] einen in [60] vorgestellten Graphenalgorithmus auf die Eignung zur Lastverteilung in verteilten System. Als Last werden wieder Aufgaben gesehen, die an verschiedene Arbeiter verteilt werden müssen. Der Algorithmus basiert auf einem Peer-to-Peer-Netzwerk. Dabei wird bei jeder erfolgreichen Beendigung einer Aufgabe eine Kante zwischen dem Arbeiter und dem Initiator erzeugt. Durch eine Jobzuweisung wird eine Kante des Arbeiters gelöscht. Um einen Job zu verteilen, wird der Graph vom Auftraggeber aus durchlaufen. Es wird zufällig eine Kante gewählt. Beim Zielknoten wird eine weitere Kante zufällig ausgewählt. Nach einer bestimmten Anzahl an Schritten wird der aktuelle Knoten als Arbeiter ausgewählt. Durch erfolgreiche Jobs werden neue Kanten erzeugt, dadurch existieren nur Verbindungen zu Knoten, die bereits erfolgreich einen Job bearbeitet haben. Da bei Vermittlung eines Auftrags eine Kante entfernt wird, kann durch die Anzahl der Kanten die Zahl der maximal zu bearbeiten Aufträge festgelegt werden.

In [50] werden die beiden oben genannten Algorithmen verglichen. Ziel ist die Maximierung der Jobabarbeitung. Es werden heterogene Netzwerke mit unterschiedlicher Anzahl an Teilnehmern und Teilnehmerklassen untersucht.

[7] `https://de.wikipedia.org/wiki/Tanzsprache`

Die Autoren stellen fest, dass der Futtersuche-Algorithmus gleichmäßig Ergebnisse erzielt, welche unabhängig von der Teilnehmeranzahl oder der Heterogenität sind. Wogegen die Leistung des graphenbasierten Algorithmus stark von der Netzgröße und der Heterogenität abhängt. Bei kleiner Netzgröße oder großer Heterogenität erzielt der graphenbasierte Algorithmus einen besseren Datendurchsatz. Für das verteilte Steuern der in dieser Arbeit betrachteten Cloud-Computing-Anwendung eignen sich die beiden untersuchten Algorithmen nicht. Die Bearbeitung von Serveranfragen erfordert eine schnelle Zuweisung auf die vorhanden Instanzen und eine Entscheidung, ob die Anzahl der Instanzen verändert werden soll. Dies kann mit den beiden Algorithmen nicht erzielt werden.

Dies war ein Überblick über Arbeiten, die sich mit dem verteilten Steuern von großen Systemen befassen. Andere Arbeiten beschäftigen sich mit der Lastverteilung von Anfragen auf Webserver, bieten allerdings keinen dezentralen Ansatz.

Chieu et al. untersuchen die Skalierung einer Cloud-Computing-Anwendung in [15]. Internetseitenbesucher werden über Lastverteiler auf einzelne Webserver geleitet. Soll eine festgelegte Zahl an gleichzeitigen Verbindungen pro Server überschritten werden, müssen neue Server gestartet werden. Die Überwachungsinstanz erhält regelmäßig von allen Servern die Anzahl der aktiven Verbindungen und berechnet daraus, ob die Serveranzahl angepasst werden muss.

Ghanbar et al. schlagen in [23] einen Regelkreis für Cloud-Computing-Anwendungen vor. Dabei werden die Antwortzeiten gemessen und nach verschiedenen Regeln entschieden, wie das System beeinflusst werden soll. Es werden drei Beispielregeln mit verschiedenen Werten verglichen. Keine allgemeingültigen Regeln wurden gefunden, sondern müssen je nach Besucher- und Anwendungsverhalten angepasst werden.

In beiden Arbeiten besitzt die Überwachungsinstanz globales Wissen und bietet einen Single-Point-of-Failure, der abgesichert werden muss. Zusätzlich skalieren die Ansätze nicht, da jeder Server eine Nachricht an die Überwachungsinstanz sendet.

Im folgenden Kapitel werden zusätzlich noch verwandte Arbeiten aus dem Bereich der Informationsverteilung vorgestellt.

2.6. Fazit

In diesem Kapitel wurde eine Einführung in Cloud-Computing gegeben. Ein Cloud-Computing-Szenario wird zur Evaluation des verteilten Managements benutzt. Die Struktur des Netzes wird aus dem *Warehouse-Scale-Computer* übernommen.

Es wurden mehrere verwandte Arbeiten untersucht. Keine der vorgestellten Arbeiten bietet eine Lösung für das Problem der Lastverteilung von Webserveranfragen und der Steuerung der Webserverpoolgröße durch Cloud-Computing.

Zur Verwaltung der Cloud sollen Konzepte aus dem Organic- und Autonomic-Computing verwendet werden. Daher wird ein komplett dezentraler Ansatz untersucht. Der Benutzer soll nur ein Ziel vorgeben und das Managementsystem kümmert sich um die konkrete Umsetzung.

3. Informationsverteilung

Um ein verteiltes System zu steuern, müssen möglichst viele Informationen über die Teilnehmer des Netzes bekannt sein. Idealerweise sollte jede Komponente aktuelle Informationen über alle anderen besitzen. Daher ist die effiziente Verteilung der Knoteninformationen ein wichtiger Aspekt der verteilten Steuerung.

Große verteilte Systeme können aus unterschiedlichen Geräten bestehen, die mit sehr unterschiedlichen Techniken kommunizieren können. Daher kann nicht davon ausgegangen werden, dass Broadcasts in allen Kommunikationsprotokollen möglich sind oder dass Broadcasts über Protokolldomains gesendet werden können. Im Beispiel der Public Cloud werden verschiedene Maschinen in verschiedenen Subnetzen oder bei verschiedenen Cloud-Computing-Anbietern angemietet. Hierbei existiert kein einheitlicher Adressraum um einen Broadcast an alle virtuellen Maschinen zu senden.

Eine Möglichkeit ist, dass jeder Knoten seine Knoteninformationen direkt an alle anderen Knoten sendet und damit einen Broadcast nachbildet. Dies würde das Netzwerk zu sehr belasten und bei einer Vergrößerung des Netzes zu exponentiell steigenden Nachrichtenzahlen führen. Zusätzlich existiert keine Liste aller Netzwerkteilnehmer, da diese sich ständig ändern können und von keiner zentralen Instanz überwacht werden. Daher wird hier eine Lösung vorgestellt, bei der jeder Knoten Nachrichten nur an wenige andere Knoten sendet, aber trotzdem die Verteilung der Informationen garantiert werden kann. Die Empfänger können alle empfangenen Informationen kombinieren und in einer großen Nachricht an andere Knoten weiter leiten. Um alle Knoten zu erreichen und doppelte Nachrichten zu vermeiden, wird ein strukturiertes Netzwerk benötigt. Darum werden strukturierte Overlaynetzwerke über das beliebig aufgebaute physikalische Netz gelegt.

Distributed Hash Tables (DHT) sind Peer-to-Peer Netzwerke, die auf das effiziente Speichern und Finden von Key-Value-Paaren optimiert sind. Dabei wird

über das physikalische Netz ein neues Kommunikationsnetz gelegt. Jeder Knoten in dem DHT-Netz besitzt eine eindeutige ID und kennt ein paar Nachbarn, die in der Routingtabelle gespeichert werden. Die gespeicherten Nachbarn werden nach einem bestimmten Schema ausgewählt. Jeder DHT-Algorithmus benutzt eine andere Methode um Nachbarn zu finden. Auch der Aufbau der Routingtabelle unterscheidet sich je nach verwendetem Algorithmus. Durch die Routing-Algorithmen der DHT-Netze können gesuchte Knoten-IDs innerhalb einer begrenzten Anzahl von Hops gefunden werden. Diese Beschränkung der Hops und die kleine Routingtabelle sind der Grund für die hohe Verbreitung von DHT. DHT-Netzwerke sind selbstorganisierend, dezentralisiert und skalierbar. Daher sind diese gut geeignet, um in einem verteiltem System eingesetzt zu werden. Im Folgenden wird die Eignung von DHT-Netzwerken für die strukturierte Informationsverteilung betrachtet.

Es folgt eine kurze Einführung in Peer-to-Peer Algorithmen. Anschließend werden verwandte Arbeiten präsentiert. Danach werden die Informationsverteilungsalgorithmen vorgestellt und evaluiert. Teile dieses Abschnitts sind unter [56] bereits veröffentlicht. Zusätzlich werden Optimierungen vorgestellt und evaluiert. Diese Optimierungen sind unter [57] veröffentlicht.

3.1. Peer-to-Peer Netzwerke und Verteilte Hashtabellen

In Peer-to-Peer (kurz P2P) Netzwerken sind alle teilnehmenden Rechner gleichberechtigt, d.h. dass alle Rechner Dienste anbieten oder nutzen können. Häufig werden P2P-Netzwerke mit Filesharing in Beziehung gebracht. Das ist auch der Bereich, in dem sie am häufigsten benutzt werden. Jeder teilnehmende Rechner bietet Dateien an und kann auch Dateien herunterladen.

P2P-Netzwerke unterteilen sich in strukturierte und unstrukturierte Netzwerke. In unstrukturierten P2P-Netzwerken ist nicht klar, welcher Rechner welche Information bereitstellt. Die Arten, wie nach Informationen gesucht wird, teilen die unstrukturierten P2P-Netze wieder in drei Kategorien:

Zentralisierte P2P-Systeme: Das System besitzt zentrale Instanzen. Jeder Teilnehmer meldet dort die Informationen, die er besitzt. Wird nach einer

Information gesucht, kann diese schnell gefunden werden. Das System ist auf die zentrale Instanz angewiesen, diese darf daher nicht ausfallen.

Reine P2P-Systeme: Systeme ohne zentrale Instanz. Hier sind alle Teilnehmer gleichberechtigt. Informationen werden mit Broadcasts bei alle Teilnehmer gesucht. Dieses Verfahren ist aufwändiger und es kann nicht garantiert werden, dass alle passenden Informationen gefunden werden. Dafür können bei diesem System beliebige Komponenten ausfallen ohne die Funktionsfähigkeit zu beeinflussen.

Hybride P2P-Systeme: Eine Mischung aus zentralisierten und reinen P2P-Systemen.

Zentralisierte P2P-Netzwerke werden als Systeme der ersten Generation bezeichnet. Beispiele dafür sind Napster und gnutella. Napster benötigte einen zentralen Server, der speicherte, welche Dateien von welchem Teilnehmer zur Verfügung gestellt wurden.

Dezentrale Algorithmen sind Systeme der zweiten Generation. P2P-Netzwerke, die keine direkte Verbindung zwischen dem Besitzer und dem Suchenden einer Datei herstellen, werden als Systeme der dritten Generation bezeichnet. Für die Informationsverteilung in dem Cloud-Computing-Szenario werden hier P2P-Systeme der zweiten Generation betrachtet, da eine zentrale Instanz dem dezentralen Prinzip von Organic-Computing widerspricht. Auch Protokolle der dritten Generation können für die Informationsverteilung benutzt werden. Die Anforderung, dass keine direkte Verbindung zwischen den Kommunikationspartnern besteht, erzeugt allerdings zusätzlichen Traffic. Da für die Informationsverteilung diese Anforderung nicht notwendig ist und unnötiger Traffic vermieden werden soll, werden die Protokolle der dritten Generation hier nicht betrachtet.

Strukturierte P2P-Netzwerke werden auch als *Verteilte Hashtabelle* (engl *Distributed Hash Table* oder kurz *DHT*) bezeichnet. Hier ist bekannt, welcher Rechner für welche Information verantwortlich ist. In diesen Netzwerken erhält jeder Knoten eine eindeutige ID. Für Inhalte wird über eine Hash-Funktion aus dem Key eine ID generiert. Jeder Knoten ist für Inhalte mit bestimmten IDs verantwortlich. Daher ist das Finden von Inhalten in strukturierten P2P-Netzwerken identisch mit dem Finden von Knoten mit einer bestimmten ID. Für die Informationsverteilung werden nur strukturierte P2P-Netze betrach-

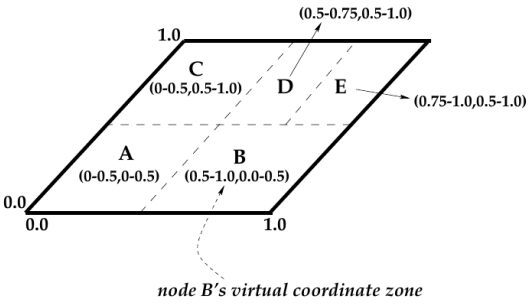


Abb. 3.1.: Beispiel für ein zweidimensionales CAN mit 5 Knoten aus [52]

tet, da diese die benötigte Struktur besitzen, um die Information effizient zu verteilen.

3.1.1. CAN

Bei CAN [52] repräsentiert das Netzwerk die Oberfläche eines n-dimensionalen Torus. Jeder Knoten erhält als ID eine Position auf dem Torus und einen Bereich auf der Oberfläche, für die der Knoten verantwortlich ist. Der Knoten speichert Informationen, wenn deren ID in seinem zuständigen Bereich liegt. Jeder Knoten kennt die Knoten-IDs der zuständigen Knoten von angrenzenden Bereichen. Dadurch kann eine Nachricht in jede Richtung weitergeleitet werden. Abbildung 3.1 zeigt ein Beispiel für ein zweidimensionales CAN mit 5 Knoten. Zu jedem Knoten ist dort der zugehörige Bereich gestrichelt eingezeichnet.

Meistens wird die Kantenlänge m für jede Dimension gleich gewählt. Dies führt zu m^n möglichen Knoten. Eine Nachricht braucht maximal $n\frac{m}{2}$ Hops. Die Routingtabelle eines Knoten speichert mindestens einen Nachbarn für jede Richtung, also zwei pro Dimension. Die maximale Größe ist schwieriger abzuschätzen, da ein Knoten beliebig viele Nachbarn haben kann. Dadurch ergibt sich eine Mindestgröße der Routingtabelle von $2n$.

3.1.2. CHORD

In CHORD [64] werden die Knoten in einem Ring mit aufsteigender ID angeordnet. Jeder Knoten ist mit dem Knoten, der die nächst größere ID besitzt, verbunden. Die Länge der ID bestimmt die Größe des maximalen Rings. Nach-

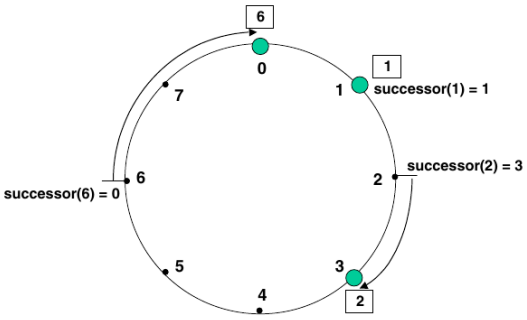


Abb. 3.2.: Beispiel für ein Chord Netz mit drei Knoten aus [64]

richten werden nur in eine Richtung in diesem Ring weitergeleitet. Um den Nachrichtenversand zu beschleunigen, bildet jeder Knoten noch Verbindungen zu Knoten mit einem ID-Abstand von 2^n. Falls kein Knoten mit der gesuchten ID vorhanden ist, wird einer mit der nächste höheren ID benutzt.

Abbildung 3.2 zeigt ein Beispiel für ein Chord Netzwerk mit drei Knoten. Die Knoten-IDs sind 0, 1 und 3. Wie im Bild zu sehen ist, werden die Informationen mit Hash 2 auf Knoten 3 und die Informationen mit Hash 6 auf dem Knoten 0 gespeichert.

In einem CHORD Netzwerk mit n Knoten kann ein Knoten mit maximal $\log_2 n$ Hops erreicht werden. Die Routingtabelle eines Knoten enthält mindestens einen Eintrag, das ist der direkte Nachfolger. Falls sich wenige Knoten im Netzwerk befinden, kann der unwahrscheinliche Fall eintreten, dass kein anderer Knoten in dem von der Routingtabelle abgedeckten Teil des Netzwerkes vorhanden ist. Maximal können aber $log_2 n$ Einträge in der Routingtabelle enthalten sein.

3.1.3. Plaxton

Pastry [58] und Tapestry [75] sind zwei ähnliche DHT-Algorithmen. Sie basieren beide auf dem von Plaxton [47] vorgestellten Routing-Algorithmus. Bei diesem Algorithmus wird jede Knoten-ID mit einer festen Länge l zur festgelegten Basis b dargestellt. Das führt zu maximal $n = b^l$ Knoten.

Die Routingtabelle wird bei Plaxton mit Knoten gefüllt, deren Knoten-ID der lokalen Knoten-ID in den ersten Stellen gleichen. In der ersten Spalte der Routingtabelle ist die Anzahl der identischen Stellen null. Mit jeder weiteren

0-xxxx	2-0-xxx		230-0-x	2301-0
1-xxxx	2-1-xxx	23-1-xx		2301-1
	2-2-xxx	23-2-xx	230-2-x	
3-xxxx		23-3-xx	230-3-x	2301-3

Tab. 3.1.: Plaxton Routingtabelle für den Knoten 23012

Spalte haben die Knoten-IDs eine Stelle mehr gemeinsam. In der letzten Spalte unterscheiden sich die gespeicherten Knoten-IDs nur noch in der letzten Stelle von der lokalen Knoten-ID. Damit hat die Routingtabelle $l - 1$ Spalten. Die erste Ziffer nach der übereinstimmenden Präfix ist auch vorgegeben. Es wird für jede Zahl zwischen 0 und b ein Eintrag in der Routingtabelle gespeichert. Ausnahme ist die Ziffer, die in der lokalen Knoten-ID dem gemeinsamen Präfix folgt. Da in der nächsten Spalte die Präfixlänge um eins erhöht wird, besitzen alle Einträge dort die ausgelassene Ziffer. Daraus ergibt sich eine Zeilenanzahl von $b - 1$.

Abbildung 3.1 zeigt die Routingtabelle für den Plaxton Knoten mit der ID 23012. Die leeren Felder werden nicht in der Routingtabelle des Knotens gespeichert, wurden hier aber eingefügt, um die Übersichtlichkeit zu erhöhen. Mit jeder Spalte nimmt die Länge des gemeinsamen Präfix zu. Die Stelle nach dem Präfix entspricht der Zeilennummer. Einträge mit einem X dienen als Platzhalter für beliebige Knoten, deren ID in den ersten Stellen mit den vorherigen Ziffern übereinstimmt. Da bei den meisten Einträgen mehrere Knoten-IDs gültige Einträge sind, benutzt Plaxton eine Abstandsmetrik um den besten Knoten zu finden. Als Abstandsmetrik kann beispielsweise der Hop Count oder die Ping Zeit benutzt werden.

Um eine Nachricht zum Zielknoten zu übermitteln, sendet jeder Knoten die Nachricht an den Nachbarknoten in der Routingtabelle, dessen Knoten-ID mit der ID des Zielknoten die längste Übereinstimmung besitzt. Dadurch kann garantiert werden, dass mit jedem Hop die Anzahl der gemeinsamen Stellen in den Knoten-IDs mindestens um eins steigt. Daraus ergibt sich eine maximale Hop Count von l oder $\log_b n$.

3.1.4. Pastry

In Pastry [58] wird der Plaxton-Routingalgorithmus benutzt. Zusätzlich zur Routingtabelle besitzt jeder Knoten noch eine Nachbarschafts- (engl. neighborhood set) und eine Blätterliste (engl. leaf set). In der Nachbarschaftsliste sind die Knoten mit der geringsten Abstandsmetrik gespeichert. Ein Knoten kann sowohl in der Routingtabelle, als auch in der Nachbarschaftsliste gespeichert sein. Diese Liste wird nicht zum Nachrichtenversand benutzt. Sie wird nur benutzt um gute Nachbarn für die Routingtabelle zu finden.

Die Blätterliste enthält die Knoten-IDs, die numerisch kleiner oder größer als die lokale ID ist. Diese Liste wird benutzt, um eine Nachricht möglichst nahe an die gesuchte ID zu leiten. Da der Zahlenraum der IDs recht groß ist, kommt es häufig vor, dass die gesuchte ID keiner vorhanden Knoten-ID entspricht. In diesem Fall werden die Daten von dem Knoten mit der geringsten Distanz zu der gesuchten ID gespeichert.

3.1.5. Tapestry

Tapestry [75] befasst sich mit dem Auffinden von Daten. Die Daten für einen Schlüssel der DHT werden nicht direkt vom zuständigen Tapestry-Knoten gespeichert. Statt dessen besitzt der Knoten einen Verweis auf den Speicherort. Wenn nun ein Eintrag gesucht wird, antwortet der für den Eintrag zuständige Knoten mit diesem Verweis, der an den suchenden Knoten weitergeleitet wird. Dabei speichern alle Knoten auf dem Weg den Verweis. Sollte nochmal nach diesem Eintrag gesucht werden, muss nicht mehr der zuständige Knoten gesucht, sondern die Position der Daten aus dem Verweis entnommen werden. Durch dieses Verhalten werden Einträge, die häufig abgefragt werden, mit wenigen Schritten gefunden.

3.2. Verwandte Arbeiten

Die simpelste Methode um Informationen in Netzwerken zu verteilen ist Flooding. Dabei sendet jeder Knoten seine Daten an alle anderen erreichbaren Knoten. Die Empfänger senden die Nachricht ebenfalls an alle erreichbaren Knoten weiter. Um zu verhindern, dass sich Kreise bilden, können die Nach-

richten kurz gespeichert werden; nur neue Nachrichten werden weitergeleitet. Dadurch kann eine Nachricht im kompletten Netzwerk verteilt werden; allerdings wird sehr viel Traffic erzeugt.

Gossiping [27] ist ein probabilistischer Ansatz. Dabei werden empfangene Nachrichten nur zu einer bestimmten Wahrscheinlichkeit p weitergeleitet. Wenn p korrekt gewählt wurde, ist die Wahrscheinlichkeit, dass alle Knoten die Nachricht empfangen haben, sehr groß. Da bei jedem Broadcast andere Knoten die Nachricht weiterleiten, verteilt sich die Last über alle Teilnehmer. Die Bestimmung von p ist nicht einfach und auch von der Größe des Netzwerkes abhängig. Verschiedene Arbeiten [38, 37, 24] befassen sich mit der Optimierung des Gossiping-Protokolls. Aber selbst durch diese Verbesserungen kann eine hundertprozentige Abdeckung nicht garantiert werden. Zusätzlich können Nachrichten mehrmals an einen Knoten geschickt werden, wobei ebenfalls unnötiger Traffic erzeugt wird.

Sameh El-Ansary et al. zeigen in [20], dass in P2P-Netzen schnell ein Broadcast-Baum erstellt werden kann. Als Beispiel wird Chord benutzt, aber auch Pastry und Tapestry sind für die Konstruktion geeignet. Dabei wird die Information an jeden Nachbarn direkt gesendet und zusätzlich angegeben, an welchen Bereich der Chord-IDs die Information weitergeleitet werden soll. Dadurch kann eine Nachricht in $\log n$ Schritten an alle Knoten gesendet werden. In der Arbeit wird nicht betrachtet, ob diese Methode auch für die Informationsverteilung von allen Knoten gleichzeitig genutzt werden kann.

Bei Broadcasts haben die Knoten in der Mitte des Broadcastbaumes eine hohe Last, da diese die Nachrichten empfangen und an mehrere Nachbarn weiterleiten. Dagegen haben die Knoten, welche die Blätter des Baumes darstellen, eine sehr geringe Last, da diese nur Nachrichten empfangen. Um die Last auf alle Knoten gleichmäßig zu verteilen, zerlegt SplitStream [13] die Nachricht in kleine Teile und sendet diese über verschiedene Broadcastbäume. Alle Knoten sind Teil mehrerer Bäume, wobei darauf geachtet wird, dass jeder Knoten äußerer Knoten in einem Baum ist. Bei der Informationsverteilung senden alle Knoten Nachrichten mit einem eigenen Broadcastbaum. Daher wird die Last bei den hier vorgestellten DHT-basierten Algorithmen automatisch balanciert.

Scribe [59, 14] ist ein auf Pastry basierender Multicast-Algorithmus. Von jedem Knoten kann ein Topic erzeugt werden. Aus einem Topic kann eine eindeutige TopicId berechnet werden. Der Topic wird auf dem Knoten, dessen KnotenId numerisch am nächsten an der TopicId liegt, verwaltet. Um einen

Topic zu abonnieren, muss dem Verwalter eine Nachricht geschickt werden. Jeder Knoten, der die Anmeldenachricht weiterleitet, speichert die KnotenId des Vorgängers als neuen Abonnenten des Topics. Sollte der Empfänger noch kein Abonnent sein, schickt dieser eine Abonnementnachricht an den nächsten Knoten auf dem Weg zum Verwalter. Wird nun eine Nachricht zu einem Topic veröffentlicht, so wird diese direkt an den Verwalter gesendet. Dieser leitet die Nachricht an die ihm bekannten Abonnenten weiter. Diese schicken die Nachricht wiederum an ihre lokal eingetragenen Abonnenten. Auf diese Weise wird die Nachricht an alle Abonnenten verteilt. Bei einem großen Problem, wie der hier diskutierten Informationsverteilung, ist der Overhead von scribe recht groß. Jeder Knoten muss seine Daten erst an den zuständigen Verwalter schicken und erst dann kann die Nachricht verteilt werden. Um dies zu verhindern, kann die TopicId so gewählt werden, dass der sendende Knoten der Verwalter ist. Da alle Knoten die Informationen benötigen, sind die zusätzlichen Abonnentenlisten unnötig. Mit dem später vorgestellten Verteilungsalgorithmus können direkt die Routingtabellen der Knoten benutzt werden.

Bayeux [76] benutzt Tapestry um Multimediainhalte zu verteilen. Im Paper wird von einer Streaming-Anwendung ausgegangen. Der Algorithmus ist für eine kleine Anzahl von Inhalteanbietern und eine große Zahl von Konsumenten konzipiert. Es wir ebenfalls ein Verwalter mit Hilfe der ID, die aus einem Streamnamen generiert wird, bestimmt. Jeder Knoten, der den Stream erhalten will, sendet eine Nachricht an den Verwalter. Bayeux ähnelt bis hier Scribe, unterscheidet sich aber im Weg der Daten vom Verwalter zum Abonnenten. Der Verwalter sendet nun eine Nachricht zurück an den Abonnenten. Alle Knoten, die auf dem Pfad dieser Rücknachricht liegen, merken sich den Pfad und leiten den Stream an den Empfänger weiter. Durch die Eigenschaften des Routingalgorithmuses können Hin- und Rückrichtung unterschiedlich sein. Auch hier muss eine zusätzliche Abonnentenliste verwaltet werden, welche bei den später präsentierten Verteilungsalgorithmen entfällt. Zusätzlich entspricht unsere Problemstellung mit vielen Sendern und ebenso vielen Konsumenten nicht dem Netzwerk des Bayeux-Szenarios.

Xu und Zhang [71] erweitern CAN um sogenannte Expressways. Diese sind vergleichbar mit den Routingtabelleneinträgen eines Chord Netzwerks. Mit Hilfe der Expressways können große Distanzen mit nur einem Hop zurück gelegt werden. Da es sich hierbei um eine Erweiterung von CAN handelt, behält das Netzwerk alle Eigenschaften von CAN. Die Expressways werden

dynamisch während des Betriebs gebildet und erzeugen nur geringen Overhead. Durch die Erweiterung verbessert sich die Routingeigenschaft von CAN auf den Faktor $\log n$. Die Expressways erlauben das schnelle Weiterleiten von Daten, zerstören aber durch die zusätzlichen Verbindungen die wohldefinierte Struktur von CAN. Daher werden bei dem Informationsverteilungsproblem zusätzliche Nachrichten versendet und damit unnötiger Traffic erzeugt.

3.3. Verteilungsalgorithmen

In diesem Abschnitt werden die auf den DHT-Algorithmen basierenden Verteilungsalgorithmen präsentiert. Die virtuellen Maschinen, die Teil des selbstverwaltenden verteilten Systems sind, werden hier als *Knoten* bezeichnet. Ziel der Algorithmen ist es, Knoteninformation (die Lastdaten der virtuellen Maschine) im Netz zu verteilen und dabei zu garantieren, dass alle Knoten diese Information genau einmal erhalten. Unter *Nachbar* wird ein Knoten verstanden, der in der Routingtabelle des lokalen Knotens eingetragen ist. Im folgenden bezeichnet eine *Information* die Daten über einen Knoten. Jede Information wird mit einem *Verteilungsbereich* versehen. Eine *Nachricht* wird von einem Knoten an einen anderen gesendet und enthält eine Information mit einem Verteilungsbereich. Jeder Knoten kombiniert verschiedene Nachrichten, die an den gleichen Knoten in der Routingtabelle gesendet werden, zu einem *Paket*. Der Empfänger eines Pakets betrachtet den Verteilungsbereich jeder Nachricht. Sollte die Nachricht noch weiter verteilt werden müssen, so teilt der Knoten den Verteilungsbereich auf und merkt die Nachrichten zur Sendung an seine Nachbarn vor. Wenn alle eingehenden Nachrichten bearbeitet wurden, verpackt der lokale Knoten alle vorgemerkten Nachrichten mit dem gleichen Empfänger in neue Pakete und sendet diese. Der Aufbau der Verteilungsbereiche variiert je nach verwendetem Verteilungsalgorithmus. Abbildung 3.3 veranschaulicht den Aufbau eines Paketes.

Algorithmus 3.1 zeigt den gemeinsamen Teil der Informationsverteilungsalgorithmen. Hier werden die empfangenen Nachrichten ausgewertet und für den Weiterversand aufgeteilt. Die Aufteilung der Nachrichten variiert je nach verwendetem Algorithmus und wird in den entsprechenden Abschnitten genauer beschrieben.

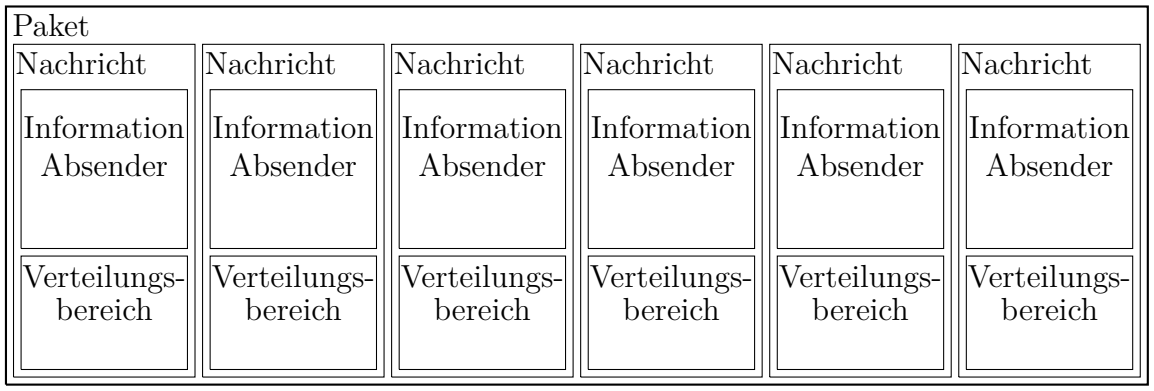


Abb. 3.3.: Aufbau eines Pakets

Algorithmus 3.1 : Informationsverteilung mit DHT

```
1  Netzwerk aufteilen;
2  Eigenen Knotenstatus mit erzeugten Empfängerintervallen in Sendeliste;
3  solange Pakete in Empfangsliste tue
4  |   Nachrichten aus Paket extrahieren;
5  |   wenn Verteilungsbereich größer als lokale Abdeckung dann
6  |   |   Verteilungsbereich anhand der Nachbarn aufteilen;
7  |   |   Nachricht mit passendem Verteilungsbereich in Sendeliste
   |   |   aufnehmen;
8  |   Ende
9  Ende
10 solange Sendeliste nicht leer tue
11 |   Nachrichten von Sendeliste an gleichen Nachbarn ermitteln;
12 |   Neues Paket mit Nachrichten erstellen;
13 |   Paket senden;
14 Ende
```

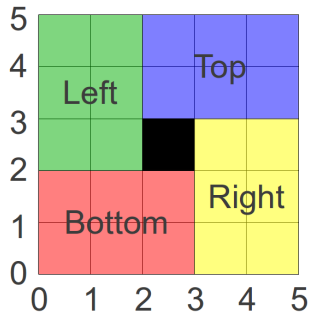


Abb. 3.4.: Verteilungsbereiche in einem aufgefaltetem zweidimensionalen CAN Torus

3.3.1. CAN

In CAN stellt die Knoten-ID eine Position auf der Oberfläche eines Torus dar; daher ist der Verteilungsbereich eine Fläche auf dem Torus. Der Erzeuger einer Information teilt die komplette Torusoberfläche in vier Bereiche mit dem erzeugenden Knoten in der Mitte. Abbildung 3.4 zeigt die Aufteilung eines zweidimensionalen Torus aus Sicht des Informationserzeugers. Anschließend werden die Informationen mit dem entsprechenden Verteilungsbereich an die Nachbarn gesendet. Jeder Knoten, der eine Nachricht empfängt, befindet sich daher an der Ecke eines Verteilungsbereichs. Daher muss die Nachricht nur noch in n Richtungen weitergeleitet werden.

3.3.2. Chord

In Chord werden zur Informationsverteilung Bäume, die aus den Nachbarschaftbeziehungen der Knoten erzeugt werden, benutzt. Der Verteilungsbereich entspricht hier einem Intervall mit Knoten-IDs. Der Chord Knoten, der seine Information verbreiten will, sendet Nachrichten an alle Knoten in seiner Routingtabelle und gibt als Verteilungsbereich das Intervall zwischen den Knoten-IDs an. Die Empfänger der Nachrichten senden die Informationen nur an Knoten in der Routingtabelle, die innerhalb des Verteilungsbereichs liegen. Abbildung 3.5 zeigt einen Verteilungsbaum in einem Chord Netzwerk mit acht Knoten.

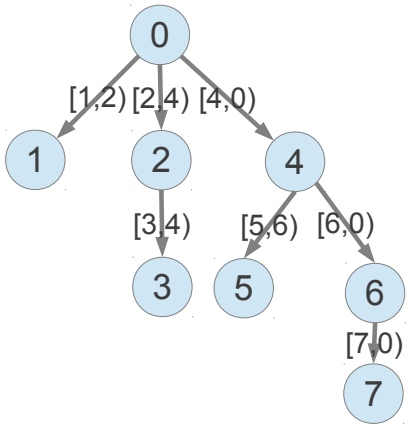


Abb. 3.5.: Verteilungsbaum in einem Chord Netzwerk mit 8 Knoten

3.3.3. Plaxton

Da in Plaxton die Routingtabelle auf der Ähnlichkeit der Knoten-ID mit der lokalen Knoten-ID basiert, wird hier die Länge der Übereinstimmung zur Bestimmung des Verteilungsbereichs benutzt. Der erzeugende Knoten schickt die Information wieder an alle Einträge in der Routingtabelle und sendet als Verteilungsbereich die Spaltennummer der Routingtabelle, in welcher der Nachbarknoten gespeichert ist. Der Empfänger leitet die Information darauf nur an Nachbarn, die in einer größeren Spalte in seiner Routingtabelle gespeichert sind, weiter.

3.4. Grenzen

Ermittlung eines Minimums an Nachrichten: Bei n Knoten muss jeder Knoten an alle anderen eine Nachricht schicken. Also folgt daraus, dass jeder Knoten $n-1$ Nachrichten sendet. Für die n Knoten ergibt sich daraus ein Minimum an $n(n-1)$ Nachrichten.

Ziel der hier vorgestellten Algorithmen ist es, nicht mehr als dieses Minimum an Nachrichten zu versenden. Nachrichten mit gleichem Empfänger sollen auch in Pakete kombiniert werden.

3.5. Evaluation

Im Folgenden wird kurz das Evalutionsszenario beschrieben. Die Ergebnisse der Evaluation werden präsentiert und anschließend diskutiert.

3.5.1. Evaluationsszenario

Es wurde eine Netzwerkgröße von 1024 gewählt, da 1024 eine gültige Größe für die 3 untersuchten DHT-Algorithmen ist. Diese führt zu einem Chord Exponenten von 10 ($2^{10} = 1024$) und für Plaxton zu dem Exponenten 5 zur Basis 4 ($5^4 = 1024$). Für CAN wurde ein zweidimensionales Netz mit einer Kantenlänge von 32 ($32^2 = 1024$) gewählt. Es wurden auch Evaluierungen mit größeren Netzen durchgeführt und diese führten zu vergleichbaren Ergebnissen.

In den Netzen wurden dann 50 bis 1000 Knoten in Schritten von 50 Knoten erzeugt. Die Knoten-IDs und damit ihre Position und Nachbarschaftsbeziehungen werden zufällig gewählt. Um den Einfluss von guten oder schlechten Platzierungen zu verringern, wurden für jede Kombination aus Knotenanzahl und Netzwerktyp 1000 Simulationen durchgeführt.

Pakete werden in festen Zeitabständen gesendet. Alle Pakete erreichen innerhalb eines dieser Zeitabstände ihr Ziel. Die Anzahl der verstrichenen Zeitintervalle wird benutzt um die Geschwindigkeit der Informationsverteilung zu messen. Auf Grund des Determinismuses der Algorithmen wird jede Information mit gleichem Sender und Empfänger auf dem gleichen Weg zum Ziel geroutet, unabhängig in welcher Runde die Information gesendet wurde. Daher stoppt die Evaluierung, wenn jeder Knoten Informationen über jeden anderen besitzt.

3.5.2. Ergebnisse

Abbildung 3.6 zeigt die durchschnittliche Rundenanzahl, die benötigt wird, um alle Informationen im Netz zu verteilen. Die untersuchte Netzgröße führt bei CAN zu einem rechnerischen Maximum von 32 Runden, bei Chord zu 10 Runden und 5 Runden bei Plaxton. Alle drei Algorithmen benötigen bei spärlich besetzten Netzen weniger Runden als bei voll besetzten Netzen, aber

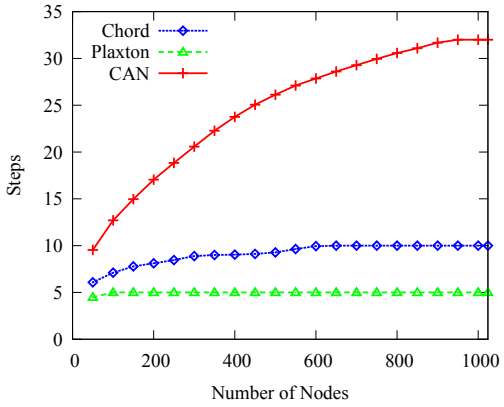


Abb. 3.6.: Durchschnittliche Anzahl der Hops bei verschieden Netzwerkgrößen

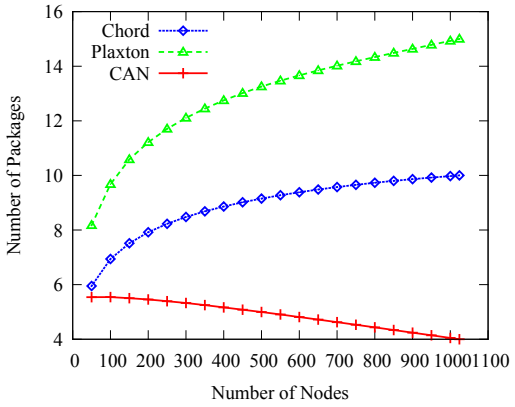
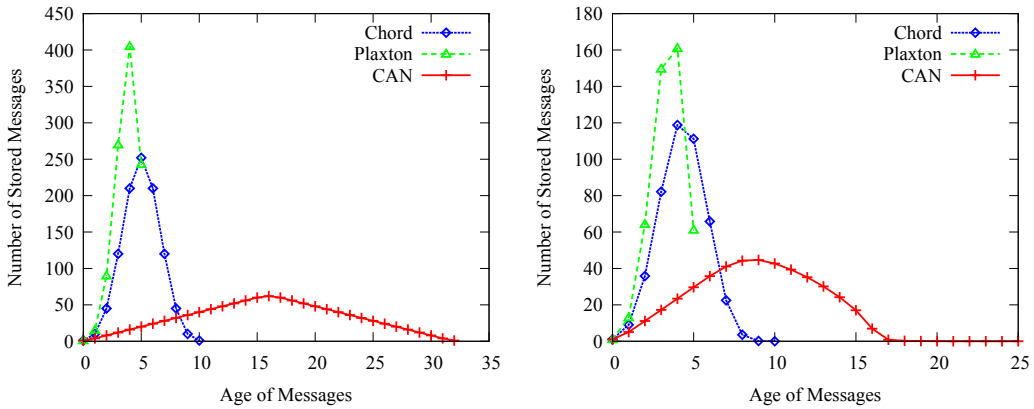


Abb. 3.7.: Durchschnittliche Anzahl der gesendeten Pakete bei verschieden Netzwerkgrößen

brauchen nie länger als das rechnerisch ermittelte Maximum. CAN hat das höchste Maximum, liegt aber auch bei größeren Netzen noch unter ihm.

Die Anzahl der gesendeten Pakete pro Knoten wird in Abbildung 3.7 gezeigt. Bei Chord und Plaxton nimmt die Anzahl zu, wenn mehr Knoten im Netzwerk vorhanden sind. CAN verhält sich entgegengesetzt. Da bei CAN Pakete an alle Nachbarn gesendet werden, führt ein unterbesetztes Netz zu mehr Pakete, da die Fläche für die ein Knoten verantwortlich ist größer wird und damit die Anzahl der Nachbarn steigt. In Chord und Plaxton nimmt die Anzahl der Pakete ab, falls kein passender Knoten gefunden wird. So kann in Chord die Abkürzung zum 2^n Nachbarn identisch mit dem 2^{n+1} Nachbarn sein, falls sich keine Knoten dazwischen befinden. Bei Plaxton kann es vorkommen, dass kein Knoten mit einem gesuchten Präfix im Netz existiert. Sowohl bei Chord als auch bei Plaxton wird in diesen Fällen der Eintrag in der Routingtabelle leer gelassen.

Im Folgenden wurde ein voll besetztes Netzwerk und eines mit 450 Knoten untersucht. Netzwerke mit 100 bis 900 Knoten verhalten sich alle vergleichbar. Nur bei mehr oder weniger Knoten kommt es zu Sonderfällen, die das Verhalten verändern. Alle folgenden Abbildungen für Netze mit 450 Knoten zeigen den Durchschnitt eines Knoten für 1000 Läufe. Daher kann es vereinzelt zu Zahlen im Bereich 10^{-6} kommen, falls das Ereignis nur einmal in 1000 Läufen für einen der 450 Knoten auftritt. Bei voll besetzten Netzen verhalten sich die Knoten fast gleich, daher wurde nur ein Lauf durchgeführt. Die Abbildungen zeigen



(a) Alter der Informationen in einem Netz mit 1024 Knoten

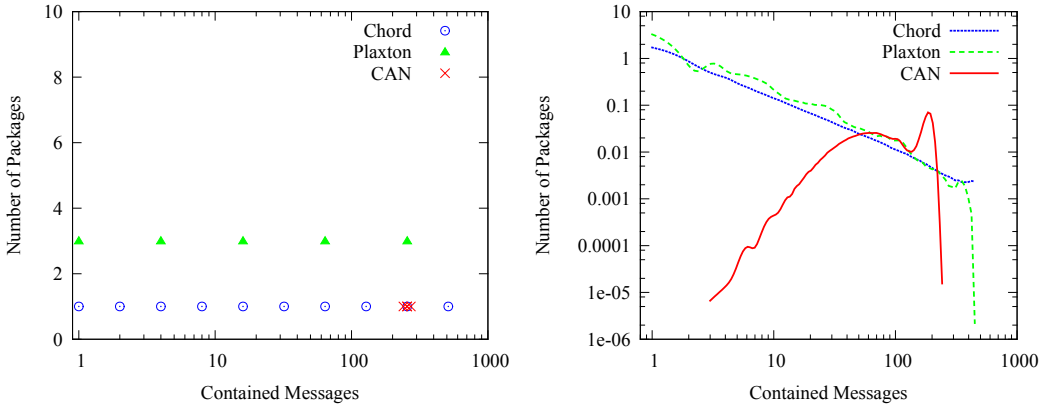
(b) Alter der Informationen in einem Netz mit 450 Knoten

Abb. 3.8.: Alter der Informationen in einem Knoten

den Durchschnitt für einen Knoten bei einem Lauf. Daher sind keine großen Abweichungen vorhanden.

Abbildungen 3.8 zeigen das Alter der Informationen, die in einem Knoten gespeichert sind. In einem voll besetzten Netz entspricht das genau einer Normalverteilung. Plaxton verteilt Informationen am schnellsten und liefert daher auch die aktuellsten Informationen, wogegen CAN die Informationen langsamer verteilt und deshalb in den Knoten auch mehr ältere Informationen gespeichert sind. In nicht voll besetzten Netzwerken ist das Verhalten ähnlich. Der größte Unterschied ist bei CAN-Netzen erkennbar. Hier enthält der Graph noch sehr kleine Einträge nach der Normalverteilung. Dies ist auf die großen Unterschiede in der Laufzeit bei CAN zurückzuführen. In CAN hat die Platzierung der Knoten eine große Auswirkung auf die Laufzeit, daher kann es bei den 1000 Läufen zu Ausreißern kommen. Bei diesen Durchläufen werden die Informationen viel langsamer verteilt als bei den anderen Durchläufen.

Um die Größe eines Pakets zu bestimmen, wurden die enthaltenen Nachrichten gezählt. Abbildung 3.9 zeigt die Häufigkeit von Paketen zu ihrer Größe. Bei einem vollständig besetzten Netzwerk (Abbildung 3.9a) sendet jeder Plaxton Knoten drei Pakete je Größe und bei Chord ein Paket je Größe. Der Grund dafür ist der Aufbau der Routingtabellen. In Plaxton enthält jede Spalte drei Einträge, daher ergeben sich drei Pakete. An alle Einträge werden die gleiche Anzahl von Informationen gesendet, da der Verteilungsalgorithmus in Plaxton



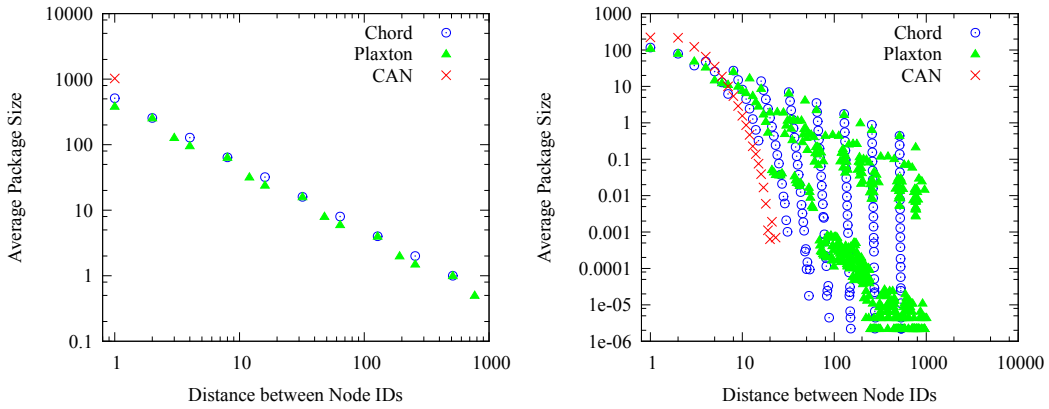
(a) Anzahl von Paketen und enthalten Nachrichten in einem Netz mit 1024 Knoten

(b) Anzahl von Paketen und enthalten Nachrichten in einem Netz mit 450 Knoten

Abb. 3.9.: Anzahl und Größe von Nachrichten

nur die Spaltennummer für die Verteilung benutzt. In Chord wird zu jedem Knoten in der Routingtabelle eine Nachricht gesendet. In CAN sendet jeder Knoten genau vier Nachrichten. Wie oben beschrieben, teilt der Sender das verbleibende Netzwerk in vier Flächen und benutzt diese als Verteilungsbereich. Bei einer Kantenlänge von 32 müssen abzüglich des Senders die Pakete an 31 Knoten pro Kante verteilt werden. Da diese durch zwei geteilt werden, ergeben sich 16 Knoten in die eine Richtung und 15 Knoten in die andere. Dies führt zu vier Verteilungsbereichen mit leicht unterschiedlicher Größe. Aus diesem Grund liegen in der Abbildung die vier roten Kreuze recht nahe beieinander, aber sind nicht identisch.

Als nächstes wurde die Beziehung zwischen der Paketgröße und der Distanz zwischen Sender und Empfänger im DHT-Netzwerk untersucht. Um die Distanz der beiden Knoten zu bestimmen, wird die numerische Distanz der beiden IDs benutzt. Sei s die Sender ID und r die Empfänger ID. Für Plaxton wird einfach die Distanz der beiden IDs, also $d_{Plaxton} = |r - s|$, benutzt. In Chord Netzen werden Nachrichten nur in Richtung aufsteigender IDs gesendet. Daher wird in Chord die Distanz mit folgender Formel $d_{Chord} = (r - s) \mod 2^m$ gezählt. In CAN wird eine Nachricht immer in die Richtung weitergeleitet, die näher am Empfänger liegt. Daher benutzt CAN die Summe



(a) Durchschnittliche Distanz per Hop in einem Netz mit 1024 Knoten

(b) Durchschnittliche Distanz per Hop in einem Netz mit 450 Knoten

Abb. 3.10.: Durchschnittliche Distanz per Hop im Verhältnis zur Paketgröße

der kürzesten Distanzen für jede Dimension. Das ergibt die Formel $d_{CAN} = \sum_{i=1}^{n} \min((r_i - s_i), m - (r_i - s_i))$.

Abbildung 3.10 zeigt die Nachrichtengrößen und den Abstand zwischen Sender und Empfänger nach den vorgestellten Metriken. Bei den vollständig besetzten Netzen (Abbildung 3.10a) ist deutlich zu erkennen, dass die Nachrichtengröße mit der Distanz der Knoten abnimmt. Ein ähnliches Verhalten ist bei den nicht voll besetzten Netzen in Abbildung 3.10b zu erkennen. Da die Knoten-ID bei den Evaluationsläufen zufällig vergeben werden, gibt es bei jedem Lauf unterschiedliche Distanzen. Das führt zu den vielen Einträgen in der Grafik.

3.5.3. Diskussion

Chord und Plaxton verhalten sich ähnlich, da die Algorithmen analog aufgebaut sind. Allerdings ist Chord nicht einfach ein Plaxton-Algorithmus mit der Basis zwei. Chord wählt Knoten mit einer festen Distanz für die Routingtabelle. Plaxton betrachtet nur das gemeinsame Präfix und wählt den Knoten für die Routingtabelle dann anhand einer Abstandsmetrik aus allen Kandidaten aus. Dieses Verhalten führt zu unregelmäßigen Einträgen in der Routingtabelle.

Plaxton sendet die meisten Pakete mit den wenigsten Nachrichten pro Schritt, da dessen Routingtabelle größer ist. Dafür benötigt Plaxton aber die wenigsten Schritte um Informationen im Netz zu verteilen und liefert daher auch aktu-

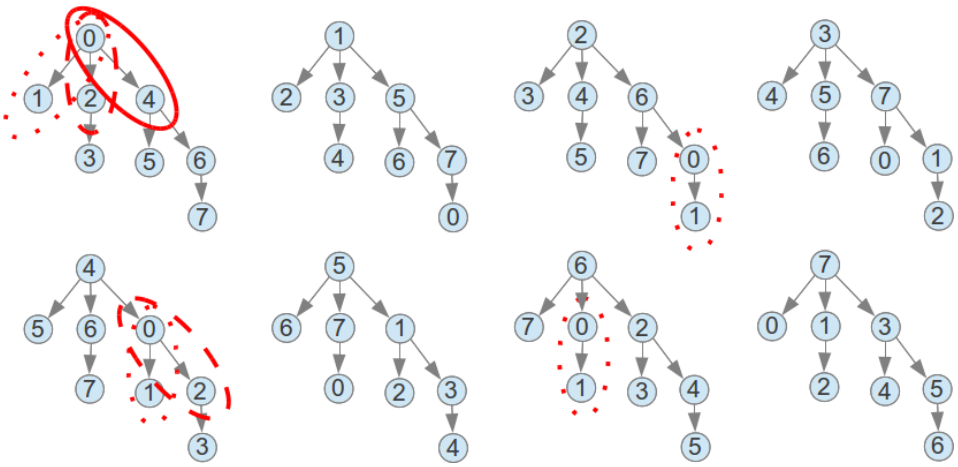


Abb. 3.11.: Verteilungsbäume in einem Chord Netzwerk mit 8 Knoten

ellere Informationen. CAN sendet nur wenige Pakete mit vielen Nachrichten pro Schritt, im Idealfall eine pro Richtung, benötigt dafür aber am längsten, um die Informationen zu verteilen.

Das Verhalten der Algorithmen ändert sich nicht signifikant falls das Netzwerk nicht voll besetzt ist. Üblicherweise wird das Peer-to-Peer-Netzwerk sehr viel größer gewählt als die tatsächlich beteiligten Knoten. Damit wird sichergestellt, dass Knoten beim Netzbeitritt keine belegte Knoten-ID wählen. Es wurde gezeigt, dass auch für die Informationsverteilung unterbesetzte Netzwerke verwendet werden können.

Die Nachrichtengröße nimmt mit steigender Distanz zwischen Sender- und Empfänger-Knoten-ID ab. Dieses Verhalten kann bei Betrachtung der Algorithmen auch erklärt werden. In Chord-Netzwerken sendet der erzeugende Knoten eine Nachricht an alle Knoten innerhalb der Routingtabelle. Weitergeleitet werden Nachrichten aber nur noch an Knoten innerhalb des Verteilungsintervalls. Da der entfernteste Eintrag in der Routingtabelle das halbe Netz umspannt, wird dieser Eintrag nur vom Erzeuger zur Verteilung genutzt. Der erste Eintrag ist der direkte Nachbar; dieser wird immer benutzt, wenn das Verteilungsintervall noch eine weitere Verteilung vorschreibt.

Plaxton Knoten benutzen die Spalte der Routingtabelle als Verteilungsintervall. Hier wird die Nachricht vom Erzeuger an alle Knoten in der Routingtabelle gesendet. Die Empfänger leiten die Nachricht dann nur an Knoten mit größer Spaltennummer in der Routingtabelle weiter. Daher werden Nachrichten von

fast allen Knoten an Einträge in der letzten Spalte gesendet, aber nur vom Erzeuger an Einträge in der ersten Spalte. Die Einträge in der letzten Spalte haben ein längeres gemeinsames Präfix mit der lokalen Knoten-ID, sind daher also näher im Sinne der vorgestellten Distanzmetriken.

Chord sendet Nachrichten immer nur an direkte Nachbarn. Ist die Distanz zwischen Nachbarn groß, so ist auch der Bereich, der von einem Knoten überwacht wird, groß. Daher enden mehr Verteilungsintervalle innerhalb dieses Bereichs.

Plaxton wählt die Einträge in der Routingtabelle anhand einer externen Metrik, wie beispielsweise der Ping Zeit oder dem Hop Count. Die Einträge in der ersten Spalte, an welche die kleinsten Nachrichten gesendet werden, haben eine geringe Übereinstimmung mit der lokalen Knoten-ID und können daher recht frei mit Hilfe der externen Metrik gewählt werden. Einträge in der letzten Spalte haben eine fest vorgegeben Knoten-ID. Falls kein Knoten mit dieser ID existiert, bleibt der Eintrag frei. Hierbei wird die externe Metrik nicht benutzt. Bei der Informationsverteilung werden allerdings die größten Pakete an Einträge in der letzten Spalte gesendet. Eine Analyse hat gezeigt, dass dieses Verhalten keinen großen Einfluss auf die Netzwerkauslastung hat.

3.6. Optimierung der ID-Vergabe

Die Evaluierung des DHT-Informationsverteilungsalgorithmus hat gezeigt, dass große Pakete an Knoten mit einer nahen Knoten-ID gesendet werden. Je größer die Distanz der Knoten-ID ist, desto geringer ist die Paketgröße. Um die Auslastung des Netzwerkes zu optimieren, wird hier ein Algorithmus vorgestellt, der Knoten mit guter Verbindung auch nahe Knoten-IDs zuteilt. Gute Verbindung heißt in diesem Fall, Knoten mit hoher Bandbreite, geringem Hop Count oder wenig Latenz.

3.6.1. Verwandte Arbeiten

Es wurden bereits einige Verbesserungen für Peer-to-Peer-Netzwerke betrachtet. Nachfolgend werden Ansätze für die in dieser Arbeit verwendeten P2P-Algorithmen vorgestellt.

Castro et al. [12] stellt eine Verbesserung für Pastry vor. Dabei wird die Blätterliste von nahen Knoten benutzt, um beim Netzbeitritt eine bessere Routing-

tabelle zu erhalten. Diese Erweiterung ist nur für Pastry geeignet und benötigt eine Blätterliste, die auch aktuell gehalten werden muss. In den hier vorgestellten Informationsverteilungsalgorithmen werden keine Blätterlisten geführt, daher kann die Optimierung nicht übernommen werden.

Topology-Aware CAN [53] benötigt wenige Landmark Server, die allen Knoten bekannt sein müssen. Jeder Knoten bestimmt seine Verbindungsqualität zu allen dieser Landmark Server. Knoten werden anhand der ermittelten Verbindungsqualität in verschiedene Eimer (engl. Bins) einsortiert. Nachbarn von Peers werden aus dem gleichen oder einem ähnlichen Eimer gewählt. Der Artikel zeigt, dass bei CAN eine ungleichmäßige Verteilung der Knoten zu einer Abnahme der durchschnittlichen Laufzeit von Nachrichten führt. Dieser Ansatz benötigt kein globales Wissen; ist aber auf Grund der Landmark Server nicht komplett dezentral.

Xu et al. [70] benutzen einen kombinierten Ansatz mit Landmark-Servern und *Round-Trip-Time* (RTT) Berechnung um das CAN-Overlaynetzwerk an die topologische Netzwerkstruktur anzupassen. Die RTT zu den Landmark-Servern wird benutzt, um die Knoten vorzusortieren. Anschließend wird die RTT von einem Knoten zu den Top Treffern bestimmt. Wieder werden Landmark-Server benötigt, die einen Single Point of Failure darstellen.

Xiong et al. [69] präsentieren eine Erweiterung von Chord für IPv6. Die Chord ID wird anhand der IPv6 Adresse erzeugt. Dazu wird der obere Teil der Chord ID durch den Hash der ersten Hälfte der IPv6 Adresse erzeugt und der untere Teil durch den Hash der letzten Stellen. Da IPv6 Adressen mit gleichem Prefix auch geografisch nah liegen, liegen auch die Knoten nahe beieinander. Da OCµ verschiedene Netzwerke unterstützt, kann nicht davon ausgegangen werden, dass alle Netzwerke eine IP-ähnliche Adresse aufweisen.

In PChord [22] wird zusätzlich zur Routingtabelle noch eine Proximity List eingeführt. Diese enthält Knoten mit geringer RTT. Die Liste ist anfangs leer und wird dann gefüllt, wenn Knoten mit geringer RTT eine Nachricht senden. Hierfür werden die normalen Nachrichten und diejenigen zur Netzinstandhaltung untersucht. Nachrichten werden dann nicht immer an den Knoten gesendet, dessen ID möglichst nah am Zielknoten liegt, sondern auch an Knoten in der Proximity-Liste, die eine geringe Latenz aufweisen und nahe am Zielknoten liegen. Bei der Benutzung von DHT-Algorithmen zur Informationsverteilung werden die Routing Algorithmen der DHT-Netze benutzt. Da die Proximity Liste die wohldefinierte Struktur durchbricht, kann damit zwar der Hop Count

zur Nachrichtenübermittlung minimiert werden, aber nicht für die Informationsverteilung. Es kann im Gegenteil sogar zu doppelten Nachrichten führen.

Bounded Gossip [8, 9] ist ein Gossiping Protokoll für große Rechenzentren. Es werden IP-Adressen benutzt, um Knoten im gleichen Subnetz zu finden. Mit dieser Information werden verschiedene Gruppen von Knoten anhand der Routeranzahl, die eine Nachricht passieren muss, gebildet. Nachrichten werden mit einer größeren Wahrscheinlichkeit an Knoten mit geringer Distanz übermittelt. Es wird mindestens eine Nachricht an die größeren Gruppen gesendet. Dadurch werden Nachrichten im kompletten Netzwerk verteilt, ohne unnötigen Traffic zu verursachen. Da es sich um ein Gossiping-Protokoll handelt, kann im Gegensatz zu den hier verwendeten Peer-to-Peer-Protokollen nicht garantiert werden, dass alle Nachrichten ihr Ziel erreichen.

Obwohl bereits viele Optimierungen für Peer-to-Peer-Netzwerke vorhanden sind, kann keine direkt auf das Anwendungsgebiet der Informationsverteilung übernommen werden. Es werden zwar Peer-to-Peer-Algorithmen für die Verteilung benutzt, allerdings sind die Unterschiede zwischen den Anwendungsbereichen zu groß. Daher ist ein angepasster Algorithmus für die Optimierung der Informationsverteilung notwendig.

3.6.2. Algorithmen

In DHT-Netzen wird die Knoten-ID generiert, bevor der Knoten dem Netz beitritt. Da die möglichen IDs viel größer gewählt sind als die tatsächlich teilnehmenden Knoten, kommt es nur in seltenen Fällen zu ID Kollisionen. Sollte die gewählte ID trotzdem bereits vergeben sein, so stellt dies der neue Knoten beim Suchen seiner Nachbarn fest. Es wird dann einfach eine neue Knoten-ID generiert und der Vorgang wiederholt. Algorithmus 3.2 zeigt diesen Algorithmus in Pseudocode.

Um Knoten mit guter Verbindung benachbarte IDs zu geben, wird die Knoten-ID beim neuen Ansatz erst nach Beitritt des Knotens berechnet. Der beitretende Knoten sendet eine Nachricht an andere, die bereits im DHT-Netzwerk teilnehmen. Diese Knoten senden eine Antwort mit ihrer Knoten-ID. Im Gegensatz zum originalen Protokoll wartet der Knoten nun eine bestimmte Zeit auf Antworten. Es werden alle eingehenden Antworten gespeichert. Aus der Antwort wird die Güte des Knotens mit einer Metrik, wie z.B. die Latenz, be-

Algorithmus 3.2 : Beitrittsalgorithmus

```
1 Erzeuge zufällige Knoten-ID;
2 wenn Kein Knoten in der DHT bekannt dann
3 |   Suche Knoten;
4 Ende
5 Direkte Nachbarn finden;
6 wenn ID ist schon vergeben dann
7 |   Gehe zu Zeile 1;
8 Ende
9 DHT-Netzwerk beitreten;
```

stimmt. Die Nachrichten werden anschließend anhand der Güte sortiert. Der beitretende Knoten sucht sich eine Knoten-ID, die nahe des besten Knotens liegt. Da alle Knoten mit guten Verbindungsqualitäten nahe gelegene IDs haben, ist die Wahrscheinlichkeit groß, dass von allen Knoten mit naher vergebener ID, eine Antwort erhalten wurde. Daher wählt sich der beitretende Knoten eine ID, die nahe des Knotens mit der besten Verbindungsqualität liegt, aber nicht in den Antworten enthalten ist. Sollte die ID bereits im Netzwerk vorhanden sein, wird solange eine weiter entfernte ID gewählt, bis eine freie ID gefunden wurde. Algorithmus 3.3 zeigt den verbesserten Algorithmus.

Algorithmus 3.3 : Verbesserter Beitrittsalgorithmus

```
1  Suche Knoten;
2  solange Wartezeit noch nicht abgelaufen tue
3  |   Antworten sammeln;
4  |   Verbindungsqualität aus Antwortnachricht auslesen;
5  Ende
6  Ordne Antworten nach Verbindungsqualität;
7  Finde unbenutzte ID in der Nähe guter Knoten;
8  Knoten-ID auswählen;
9  wenn ID ist schon vergeben dann
10 |   gehe zu Zeile 7;
11 Ende
12 DHT-Netzwerk beitreten;
```

Um beim Start des Systems gute von schlechten Verbindungen unterscheiden zu können, sollte dem Algorithmus eine Abschätzung der maximalen und minimalen Abstandsmetrik übergeben werden. Ansonsten würden die ersten beiden Knoten unabhängig von der Distanz gegenüber im ID-Bereich platziert. Wenn das Netzwerk bereits besteht, ist eine gute Platzierung ohne dieses Wissen möglich.

3.6.3. Analyse

In diesem Abschnitt wird eine Analyse der beiden Beitrittsalgorithmen vorgestellt. Beide Algorithmen senden eine oder mehrere Nachrichten um dem DHT-Netz beizutreten. Sei n_{suche} die Anzahl der Nachrichten, die ein Knoten auf seine Suchanfrage bekommt. Die Größe von n_{suche} ist abhängig von den Knoten, die in der Nähe von dem suchenden Knoten liegen. Sobald eine Antwort eingeht, tritt der Knoten beim originalen Algorithmus 3.2 dem DHT bei. Dafür sucht der Knoten seine Nachbarn. Sei $n_{nachbarn}$ die Anzahl der Nachrichten, die benötigt werden, um die Nachbarn zu finden.

Mit dem verbesserten Algorithmus 3.3 wird ebenfalls eine Suchanfrage gesendet. Auch n_{suche} Antworten werden an den Suchenden übermittelt. Allerdings wartet der beitretende Knoten auf mehrere Antworten. Da bei dem verbesserten Algorithmus Knoten mit guter Verbindung nahe gelegene IDs haben, ist die Wahrscheinlichkeit groß, dass unter den antwortenden Knoten bereits Nachbarn gefunden werden können. Daher müssen nur $n'_{nachbarn}$ Nachrichten gesendet werden, um fehlende Nachbarn zu finden. Im schlechtesten Fall ist $n_{nachbarn} = n'_{nachbarn}$. Meistens gilt aber $n_{nachbarn} > n'_{nachbarn}$. Dadurch werden keine zusätzlichen Nachrichten benötigt, sondern sogar welche eingespart.

3.6.4. Evaluation

Der Simulator wurde erweitert, um die Auslastung der einzelnen Verbindungen zu messen. Dafür wurde zuerst ein physikalisches Netzwerk mit 1024 Knoten erzeugt. Die Topologie des Netzwerks entspricht einem Baum mit Tiefe 3. Dabei befindet sich ein Knoten als Backbone auf der ersten Ebene und 15 Knoten auf der zweiten Ebene. Die restlichen Knoten wurden gleichmäßig auf der dritten Ebene verteilt. In diesem Netzwerk mit 1024 Knoten wurden anschließend zufällig die Knoten des DHT ausgewählt. Wieder wurden DHT-Netze mit 50

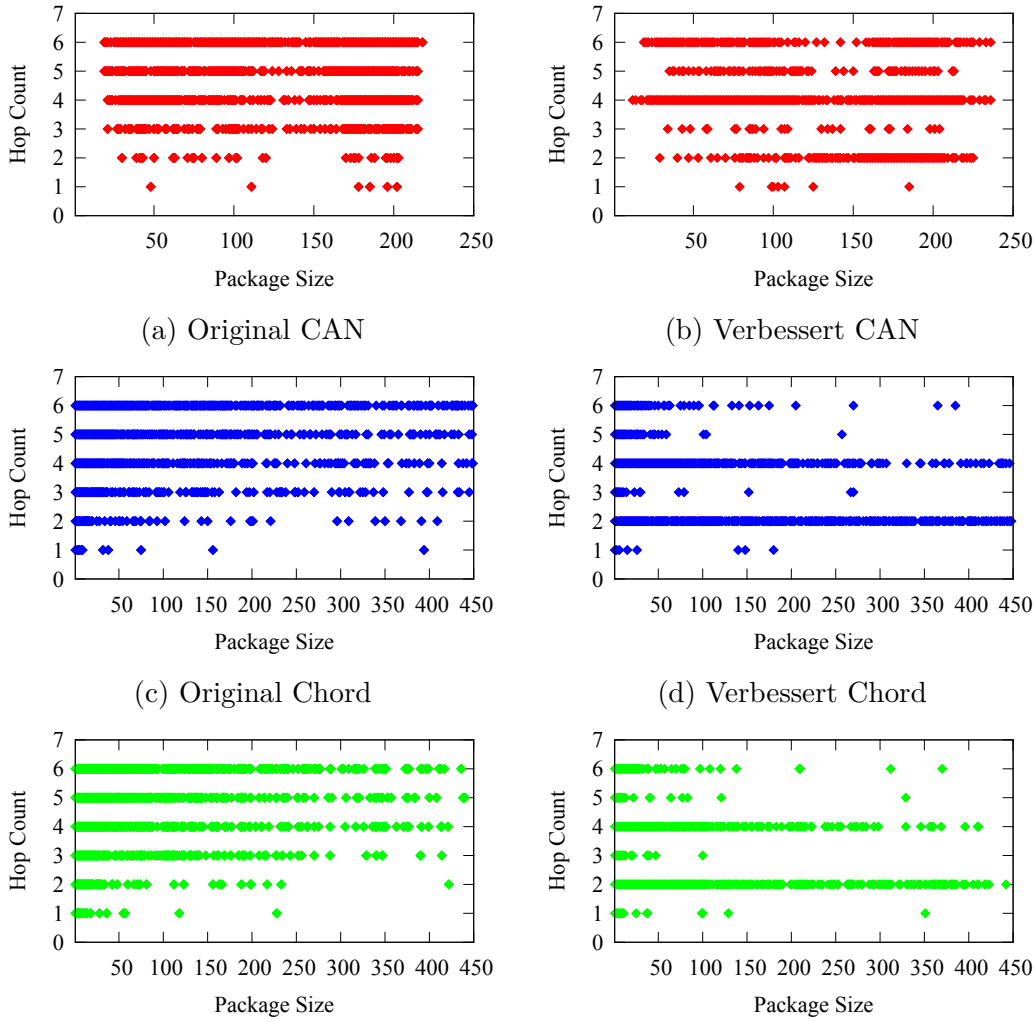
bis 1024 Knoten in Schritt von 50 Knoten erzeugt. Bei Beitritt der Knoten in die DHT wurde die ID einmal mit dem originalen Algorithmus 3.2 und im nächsten Durchlauf mit dem verbesserten Algorithmus 3.3 erzeugt. Nachdem das Netz komplett erstellt war, wurde die Evaluation gestartet. Knoten können nur Nachrichten über physikalische Verbindungen senden. Sollte der Empfänger nicht direkt erreichbar sein, wird die Nachricht weitergeleitet.

3.6.5. Ergebnisse

Abbildung 3.12 zeigt die Verteilung der Pakete bezüglich ihrer Größe und des Hop Counts. Für jede Nachricht wird ein Punkt in den Graphen eingefügt. Je größer eine Nachricht ist, desto weiter rechts wird der Punkt platziert. Der Hop Count wird auf der y-Achse angetragen und nimmt noch oben zu. Durch die Optimierung wird erwartet, dass große Nachrichten einen kleinen Hop Count besitzen und nach Möglichkeit nur kleine Nachrichten über große Distanzen geschickt werden. Je weniger Punkte rechts oben zu finden sind, desto weniger Traffic entsteht im Netzwerk.

Abbildung 3.12a zeigt die Verteilung für CAN Netzwerke mit dem originalen Algorithmus. Im Vergleich zu Abbildung 3.12b ändert sich wenig durch die Verwendung des verbesserten Algorithmus. Anders sieht es bei Plaxton und Chord aus. Abbildung 3.12c zeigt die Verteilung in Chord Netzen mit dem originalen Algorithmus. Durch den verbesserten Algorithmus (Abbildung 3.12d) werden weniger große Pakete über lange Netzwerkverbindungen gesendet. Es ist deutlich zu sehen, dass durch den verbesserten Algorithmus kaum noch Pakete mit mehr als 200 Nachrichten 6 Hops benötigen. Ähnlich verhält es sich bei Plaxton. Durch den verbesserten Algorithmus (Abbildung 3.12e) werden weniger Pakete mit mehr als 150 Nachrichten über 6 Hops verschickt, als bei Verwendung des originalen Algorithmus (Abbildung 3.12f).

Abbildung 3.13 zeigt die Summe des Hop Counts über alle Pakete, für verschiedene Netzwerkgrößen und Algorithmen. Es ist deutlich zu sehen, dass mit dem verbesserten Algorithmus der Hop Count bei allen drei DHT-Algorithmen im Vergleich zur zufälligen ID-Vergabe gesenkt werden kann.



(a) Original CAN

(b) Verbessert CAN

(c) Original Chord

(d) Verbessert Chord

(e) Original Plaxton

(f) Verbessert Plaxton

Abb. 3.12.: Verteilung der Hops und Paketgrößen mit dem original (links) und dem verbesserten Algorithmus (rechts)

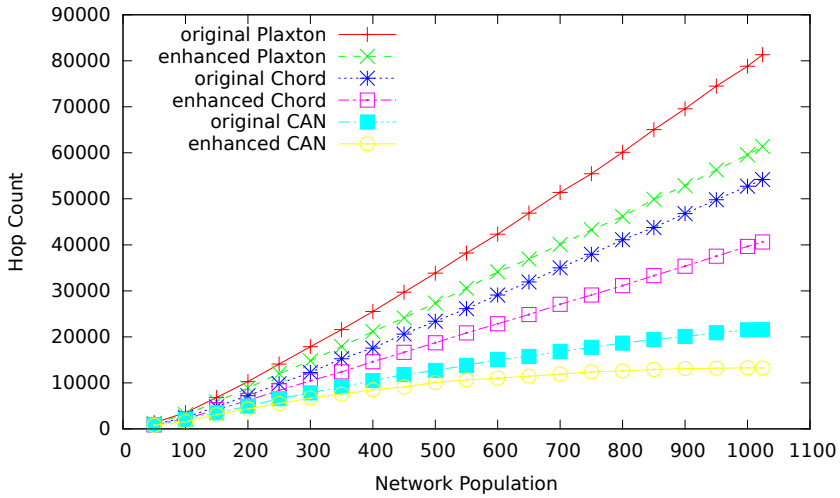


Abb. 3.13.: Hop-Anzahl bei verschiedenen Netzgröße

3.6.6. Diskussion

Die zufällige Verteilung der IDs verhindert eine Clusterbildung der Knoten im ID-Bereich des P2P-Algorithmuses. Durch die Clusterbildung können Plätze in der Routingtabelle nicht besetzt werden, da alle IDs nah an denen der lokalen Knoten-ID liegen. Dadurch bleiben gerade bei Chord und Plaxton viele Routingtabelleneinträge leer, was zu einer Verschlechterung der Laufzeit führen kann. In den Evaluationen wurde kein solches Verhalten festgestellt. Wichtig ist hierbei, dass die Größe des ID-Bereichs auf die maximale Anzahl der Knoten abgestimmt und nicht mehr wie bei P2P-Netzwerken üblich, mehrere Zehnerpotenzen größer gewählt wird.

3.7. Fazit

Durch die vorgestellten Verteilungsalgorithmen kann garantiert werden, dass Informationen im kompletten Netz verteilt werden. Die Dauer der Verteilung variiert nach verwendetem Peer-to-Peer-Algorithmus und der Netzgröße.

Die Chord- und Plaxton-basierten Algorithmen können Informationen ohne redundante Nachrichten im Netz verteilen. Die Anzahl der pro Knoten gesendeten Pakete steigt bei beiden Algorithmen logarithmisch mit der Netzgröße, wobei die Zeit zur Nachrichtenverteilung auch nur logarithmisch wächst. Dage-

gen sendet der CAN-basierte Algorithmus nur an wenige andere Knoten, benötigt dafür länger um die Informationen zu verteilen. Die Anzahl der Nachbarn bleibt konstant, allerdings wächst die benötigte Verteilungszeit in dem hier verwendeten zweidimensionalen ID-Raum mit $\mathcal{O}(\sqrt[2]{n})$ zur Netzgröße. Dadurch kann bei schlechter Netzqualität verhindert werden, dass viele Daten über das Netz gesendet werden.

Mit dem verbesserten Algorithmus zur ID-Vergabe kann die Auslastung des physikalischen Netzwerks gesenkt werden. Durch den Algorithmus werden Knoten mit guter physikalischer Verbindung mit geringer Distanz untereinander im DHT-Netzwerk positioniert. Dadurch werden große Pakete nur noch an Knoten mit einer guten Anbindung gesendet und somit das Netzwerk entlastet.

4. Steuerung

Durch die Peer-to-Peer-Verteilungsalgorithmen besitzt jeder Knoten Informationen über das komplette Netz. Da die Dauer der Verteilung nicht konstant ist, variiert das Alter der Informationen. Eine verteilte Steuerung muss mit diesen teilweise veralteten Daten das System beeinflussen. In diesem Kapitel wird das Cloud-Computing-Szenario präsentiert, das als Beispielsystem für die Analyse und Evaluierung der Informationsverteilungs- und Steuerungsalgorithmen verwendet wird.

Es werden verschiedene Methoden der Datenanalyse und deren Eignung für das verteilt gesteuerte Cloud-Computing-Szenario untersucht. Zur Informationsverteilung wird der im vorherigen Kapitel vorgestellte Chord-basierte Algorithmus verwendet und ermittelt, welche Auswirkungen dieser auf die Steuerung hat.

Zur Auswertung der Daten werden ein arithmetischer und ein Fuzzylogik Ansatz benutzt. Fuzzylogik wurde ausgewählt, weil durch die verständliche Regelsprache auch Benutzer neue Regeln hinzufügen können. Es wird untersucht, in wieweit sich die beiden Ansätze von einer zentral gesteuerten Referenzimplementierung unterscheiden. Zur Untersuchung werden verschiedene Systeme simuliert und die Ergebnisse diskutiert.

4.1. Cloud-Computing-Szenario

Um die verteilte Steuerung von großen verteilten Systemen zu evaluieren, wird hier ein Cloud-Computing-Szenario vorgestellt. Dieses Szenario umfasst mehrere Hundert Instanzen, auf denen die Last verteilt werden muss. Dazu wird angenommen, dass virtuelle Computer bei einem großen Cloud-Computing-Anbieter angemietet werden. Auf den gemieteten Servern wird eine große Internetanwendung betrieben.

Als Optimierungsziel wird die Antwortzeit der Webanwendung benutzt, da dies für den Besucher einen sehr wichtigen Aspekt darstellt. Je mehr Besucher die Anwendungen gleichzeitig benutzen, um so mehr erhöht sich die Bearbeitungszeit. Um dem entgegen zu wirken, können zusätzliche Webserver gestartet werden, um die Bearbeitungszeit zu verkürzen. Sollte die Antwortzeit zu gering ausfallen, können Webserver gestoppt werden um die Kosten zu senken, ohne die Besucher unnötig warten zu lassen. In dem Szenario wird eine Antwortzeit von 300 ms als Ziel gesetzt.

Es werden vier verschiedene Besucherverhalten simuliert. Sechs Szenarien simulieren den spontanen Anstieg bzw. Abfall der Besucherzahlen. Die restlichen sieben Szenarien sind echtem Besucherverhalten nachempfunden. Sie simulieren unterschiedliche Schwankungen und Anstieg der Besucher. Eine genaue Beschreibung folgt im Kapitel 4.4.2.

4.1.1. Komponenten

In dem Cloud-Computing-Szenario werden zwei Komponenten betrachtet, welche auf die Bearbeitungszeit einer Anfrage großen Einfluss haben. Als erstes wird der Load Balancer vorgestellt, der die eingehenden Anfragen der Benutzer auf die Webserver verteilt. Anschließen werden die Webserver genauer betrachtet.

4.1.2. Load Balancer

Load Balancer werden im Netzwerk vor die Anwendungsserver geschaltet und dienen zur Lastverteilung. Anfragen von Benutzern werden an den Load Balancer gerichtet, der die Anfragen an die Webserver verteilt. Load Balancer sind darauf ausgelegt, möglichst viele Anfragen zu bearbeiten. Sie besitzen weniger Rechenleistung als die Webserver, müssen aber die Anfragen mehrerer Webserver verteilen. Daher wird wenig Logik in Load Balancer eingebaut. Die Anfragen können zufällig oder nach dem Round-Robin-Verfahren verteilt werden. Einige Load Balancer können auch die Antwortzeiten der Webserver messen. Falls die Antwortzeit eines Webservers zu lang wird, werden zukünftige Anfragen nicht mehr an diesen Webserver geleitet. Sobald sich die Antwortzeit normalisiert hat, können erneut Anfragen an diesen Webserver gesendet werden. Bei großen Anwendungen existieren mehrere Load Balancer, die An-

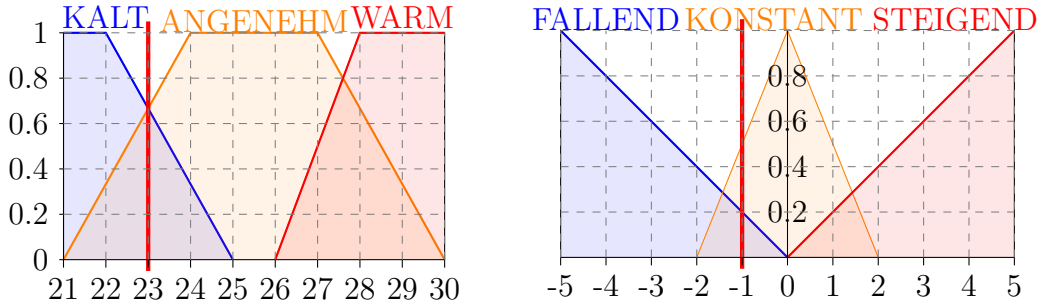
fragen jeweils nur an eine Teilmenge der vorhandenen Webserver weiterleiten. Zu welchem Load Balancer ein Besucher geleitet wird, wird mit dem Round-Robin-Verfahren des Domain Name Servers geregelt. Durch DNS Caches bei den Internetanbietern wird die Last nicht gleichzeitig auf die Load Balancer verteilt. Da zusätzlich die Anzahl und Komplexität der Anfragen von jedem Besucher stark schwankt, kann keine gleiche Auslastung der Webserver garantiert werden. In dem Evaluationsszenario werden die Anfragen der Besucher zufällig an die Webserver verteilt.

4.1.3. Webserver

Die wichtigste Komponente der Cloud-Computing-Anwendung ist der Webserver, der alle Anfragen von Besucher beantwortet. Je nach Komplexität der Anfrage müssen noch unterschiedlich viele Datenbankanfragen gestellt werden. Solche Anfragen verlangsamen die Bearbeitung enorm, da die Datenbank meist auf eigenen Servern läuft. Häufig verwendete Seiten oder Seitenbestandteile können sehr schnell aus dem lokalen Cache generiert werden. Viele Anfragen führen ebenfalls zu häufigeren Cache Misses oder dem Verdrängen von wichtigen Inhalten, was die Bearbeitung noch zusätzlich verlangsamt. Je mehr Anfragen ein Webserver gleichzeitig bearbeitet, desto langsamer werden die Antwortzeiten aller Abfragen. In dem Evaluationsszenario besitzen alle Anfragen die gleiche Komplexität. Um trotzdem eine unterschiedliche Auslastung zu erhalten, werden die Anfragen von den Load Balancern zufällig verteilt.

4.2. Fuzzylogik

In der Regelungstechnik werden die Regelungssysteme mathematisch modelliert. Ein komplexes reales System kann zu einem großen mathematischen Modell führen. Sollte das Modell fehlerhaft sein, führt dies zu Fehlern in der Regelung. Um das Modellieren komplexer Systeme zu erleichtern, veröffentlichte Zadek 1965 [73] eine unscharfe Logik, die er Fuzzylogik (engl. *fuzzy logic*) nannte. In der Booleschen Logik sind Variablen entweder wahr oder falsch. Im Gegensatz können die in der Fuzzylogik verwendeten linguistischen Variablen jeden Wert zwischen 0 und 1 annehmen. Dadurch können nicht eindeutige Zustände besser modelliert werden. Fuzzylogik wird neben der Regelungstechnik



(a) Temperatur

(b) Temperaturänderung

Abb. 4.1.: Eingangsvariablen der Temperaturregelung

ebenfalls in anderen Bereichen, wie z.B. Betriebswirtschaft [7], Unterhaltungselektronik [65] oder zur Spracherkennung [2] eingesetzt.

Ein Fuzzylogik-Regler besteht aus drei Schritten. Im ersten Schritt, der Fuzzyfizierung, werden die Eingangsgrößen in linguistische Variablen überführt. Im nächsten Schritt werden mit den implementierten Regeln und den linguistischen Variablen die Fuzzylogik-Ausgaben bestimmt, dies wird Inferenz genannt. Anschließend werden bei der Defuzzyfizierung die Fuzzylogik-Ausgaben in konkrete Ausgangswerte für den Regler umgewandelt. Im Folgenden werden die drei Stufen genauer anhand eines Beispiels erklärt. Als Beispiel wird eine Temperaturregelung benutzt. Die Temperatur soll in diesem Szenario zwischen 25 und 26 Grad gehalten werden. Dieser Wert wird in der Regelungstechnik als Führungsgröße bezeichnet. Über einen Sensor wird die aktuelle Raumtemperatur gemessen. Zusätzlich wird die Temperaturänderung in der letzten Minute berechnet und gespeichert. Diese beiden Werte bilden die Eingangsgrößen der Steuerung. Um die Temperatur zu beeinflussen, kontrolliert der Regler das Heizungsventil. Der von einer Regelung beeinflusste Wert, wird als Regelgröße bezeichnet und entspricht in dem Beispiel der Stellung des Heizungsventils.

4.2.1. Fuzzyfizierung

In der ersten Phase werden die Eingangswerte der Sensoren gemessen und mit Hilfe von Zugehörigkeitsfunktionen in linguistische Variablen umgewandelt. Für die Temperatur werden die linguistischen Variablen *WARM*, *ANGENEHM* und *KALT* verwendet. *FALLEND*, *KONSTANT* und *STEIGEND* werden für die Temperaturänderung benutzt. Ein Eingangswert kann mehre-

re Variablen beeinflussen. Abbildung 4.1a zeigt die Zugehörigkeitsfunktionen der Beispiel-Temperaturregelung. Als Eingangsgrößen wurden eine Temperatur von 23 Grad und eine Temperaturänderung von -1 Grad gewählt. Die roten Linien markieren die Position in den Graphen. Diese Eingangsgrößen führen zu den Belegungen $WARM = \frac{2}{3}$, $ANGENEHM = \frac{2}{3}$ und $KALT = 0$ für die Temperatur. Durch das Absinken um 1 Grad werden die Variablen *FALLEND* = 0.2, $KONSTANT = 0.5$ und $STEIGEND = 0$ belegt.

4.2.2. Inferenz

Durch einfache WENN-DANN-Regeln wird aus den Eingangsvariablen der Wahrheitswert der Ausgangsvariablen bestimmt. Es können mehrere Variablen in den WENN-Teil mit logischen Verknüpfungen verbunden werden. Da es sich nicht um boolesche Variablen handelt, müssen hier spezielle Fuzzylogik-Operatoren für *UND*, *ODER* und *NICHT* verwendet werden. Da die Fuzzylogik sich an die Fuzzymengen anlehnt, werden Operatoren aus der Mengenlehre benutzt. Für *UND* wird das Minimum, das einem Schnitt zweier Mengen entspricht, verwendet. *ODER* stellt eine Vereinigung und damit das Maximum der beiden Werte dar. Für den *NICHT* Operator wird das Komplement zu 1 benutzt.

```
(1) wenn TEMPERATUR KALT und AENDERUNG FALLEND dann STARK_HEIZEN
(2) wenn TEMPERATUR KALT und AENDERUNG KONSTANT dann HEIZEN
(3) wenn TEMPERATUR KALT und AENDERUNG STEIGEND dann HEIZEN
(4) wenn TEMPERATUR ANGENEHM und AENDERUNG FALLEND dann LEICHT_HEIZEN
(5) wenn TEMPERATUR ANGENEHM und AENDERUNG KONSTANT dann LEICHT_HEIZEN
(6) wenn TEMPERATUR ANGENEHM und AENDERUNG STEIGEND dann NICHT_HEIZEN
(7) wenn TEMPERATUR WARM und AENDERUNG FALLEND dann LEICHT_HEIZEN
(8) wenn TEMPERATUR WARM und AENDERUNG KONSTANT dann NICHT_HEIZEN
(9) wenn TEMPERATUR WARM und AENDERUNG STEIGEND dann NICHT_HEIZEN
```

Fuzzylogik-Regeln 1: Heizungssteuerung

Für die Heizungssteuerung werden die Fuzzylogik-Regeln 1 benutzt. In den Fuzzylogik-Regeln 2 sind die Wahrheitswerte für die Temperatur von 22 Grad und einer Temperaturänderung von -1 Grad eingetragen.

```
(1) wenn 0 und 0,2 dann STARK_HEIZEN(0)
(2) wenn 0 und 0,5 dann HEIZEN(0)
(3) wenn 0 und 0 dann HEIZEN(0)
(4) wenn 0,6 und 0,2 dann LEICHT_HEIZEN(0,2)
(5) wenn 0,6 und 0,5 dann LEICHT_HEIZEN(0,5)
(6) wenn 0,6 und 0 dann NICHT_HEIZEN(0)
(7) wenn 0,3 und 0,2 dann LEICHT_HEIZEN(0,2)
(8) wenn 0,3 und 0,5 dann NICHT_HEIZEN(0,3)
(9) wenn 0,3 und 0 dann NICHT_HEIZEN(0)
```

Fuzzylogik-Regeln 2: Heizungssteuerung mit Wahrheitswerten

NICHT_HEIZEN	0,3
LEICHT_HEIZEN	0,5
HEIZEN	0
STARK_HEIZEN	0

Tab. 4.1.: Wahrheitswert der Ausgangsvariablen

Nach dem Auswerten der einzelnen Regeln werden die Wahrheitswerte der Ausgangsvariablen zusammen gefasst. Dies kann auf verschiedene Arten erfolgen. In der aktuellen Anwendung wird das Maximum der Wahrheitswerte verwendet. Tabelle 4.1 zeigt die Wahrheitswerte der Ausgangsvariablen.

4.2.3. Defuzzyfizierung

Die in der Interferenz ermittelten Wahrheitswerte werden während der Defuzzyfizierung benutzt, um zu bestimmen, wie der Fuzzylogik-Regler die Stellgröße beeinflussen soll. Üblicherweise werden, wie bei den Eingangsgrößen, Zugehörigkeitsfunktionen definiert. Je nach Wahrheitswert der Ausgangsvariablen wird nur ein Teil der Fläche ausgewählt und aus allen ausgewählten Flächen der Schwerpunkt berechnet. Der x-Wert des Schwerpunkts entspricht dann der Stellgröße.

NICHT_HEIZEN	0
LEICHT_HEIZEN	25 %
HEIZEN	50 %
STARK_HEIZEN	100 %

Tab. 4.2.: Stellgrößen der Ausgangsvariablen

In diesem Beispiel werden Singletons verwendet. Dabei werden statt Flächen die x-Position der Schwerpunkte benutzt. Diese werden mit den Wahrheitswerten gewichtet und ein gewichtetes Mittel berechnet. Tabelle 4.2 zeigt die Ventilöffnung der Heizung zu den entsprechenden Ausgangsvariablen.

Die Formel 4.1 zeigt die Berechnung der Stellgröße in dem Beispiel. Eine Temperatur von 22 Grad und einer Temperaturänderung von -1 Grad führt zu einer Ventilöffnung von 15,625 %.

$$\frac{0,3 \cdot 0\% + 0,5 \cdot 25\% + 0 \cdot 50\% + 0 \cdot 100\%}{0,3 + 0,5 + 0 + 0} = \frac{12,5\%}{0,8} = 15,625\% \qquad (4.1)$$

4.3. Steuerungsalgorithmen

Im letzten Kapitel wurde diskutiert, wie Informationen möglichst effizient in großen Netzwerken verteilt werden können. Um eine verteilte Steuerung zu ermöglichen, werden in diesem Abschnitt verschiedene Methoden vorgestellt, wie mit diesen Informationen eine verteilte Steuerung implementiert werden kann. Im Folgenden wird eine arithmetische Methode und eine auf Fuzzylogik basierende Steuerung zur Analyse der Lastinformationen vorgestellt. Als Beispiel dient eine verteilte Steuerung für eine Cloud-Computing-Anwendung. Dabei muss die Antwortzeit der Webserver auf 300 ms gehalten werden. Längere Werte verärgern Besucher, verringern die Nutzung der Anwendung und damit die Werbeeinnahmen. Kürzere Antwortzeiten deuten auf zu viele virtuelle Maschinen hin und verursachen daher höhere Kosten als notwendig.

Da bei den Peer-to-Peer Algorithmen die Informationen meistens mehrere Knoten zwischen Sender und Empfänger passieren, sind die Informationen bei ihrer Ankunft schon veraltet. Daher wird das Alter der Informationen betrachtet und ältere Informationen weniger gewichtet als aktuelle.

Die tatsächliche Reaktionszeit der Webserver wird alle 60 Sekunden ermittelt und für die Analyse weitergeleitet. Um leichte Fluktuationen zu vermeiden, wird das System nicht verändert, wenn die Reaktionszeit nur wenige ms vom Zielwert abweicht.

4.3.1. Arithmetischer Ansatz

Beim arithmetischen Ansatz werden alle bekannten Antwortzeiten gemittelt. Wie oben beschrieben, sind einige Information älter, daher wird ein gewichtetes Mittel gebildet. Dazu wird folgende Formel benutzt:

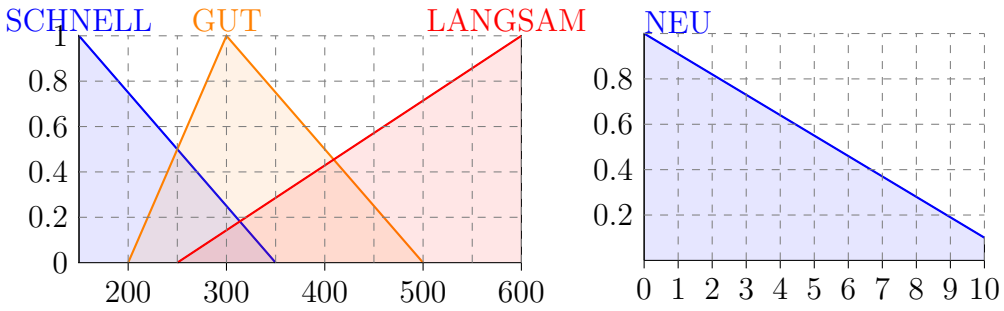
$$\frac{\sum_{N \in Messwerte} \frac{N_{Antwortzeit}}{Messintervall + N_{Alter}}}{\sum_{N \in Messwerte} \frac{1}{Messintervall + N_{Alter}}} \tag{4.2}$$

Da die Daten des lokalen Knotens kurz vor der Berechnung gesetzt werden, ist beim Alter eine Konstante zu addieren, da sonst durch Null geteilt werden müsste. Um die älteren Werte nicht komplett irrelevant werden zu lassen, eignet sich als Konstante die Länge des Messintervalls. In diesem Szenario sind das 60 Sekunden. Dadurch werden Werte, die 2 Minuten alt sind noch mit $\frac{1}{3}$ eingerechnet und 3 Minuten alte, werden noch zu $\frac{1}{4}$ berücksichtigt.

Der arithmetische Ansatz verändert das System nicht, wenn der gemessene Wert um weniger als 10 ms vom Zielwert abweicht. Es ergibt sich daraus ein Intervall von 290 ms bis 310 ms. Um nicht auf jede kleine Änderung des Besucherverhaltens zu reagieren, werden die letzten Werte gespeichert. Sollten alle gespeicherten Werte und der aktuelle Wert außerhalb des Intervalls liegen, wird das System angepasst. Die optimale Anzahl der Instanzen wird mit folgendem Dreisatz ermittelt:

$$\frac{300ms}{Messwert} \cdot \#Instanzen \tag{4.3}$$

Anschließend werden Instanzen gestartet oder gestoppt, um die optimale Anzahl zu erreichen. Dieser Ansatz beschränkt nicht die Anzahl der Instanzen.



(a) Fuzzylogik-Zugehörigkeitsfunktion für die Antwortzeit

(b) Fuzzylogik-Zugehörigkeitsfunktion für das Alter der Nachrichten

Abb. 4.2.: Eingangsvariablen der Cloud-Steuerung

Es werden immer so viele Instanzen gestartet oder gestoppt, um mit einem Schritt auf die optimale Antwortzeit zu kommen.

4.3.2. Analyse mit Fuzzylogik

Als zweite Methode wird die Analyse mit Fuzzylogik untersucht. Hierbei wird auch die Lastinformation und das Alter der Nachricht als Eingabe benutzt.

Abbildung 4.2 zeigt die verwendeten Zugehörigkeitsfunktionen. Die linke Funktion klassifiziert die *ANWORTZEIT* in drei Kategorien *LANGSAM*, *GUT* und *SCHNELL*. Als Wertebereich wurden 150 bis 600ms gewählt. Das entspricht der halben und doppelten Zielantwortzeit von 300ms. Die rechte Funktion bestimmt das *ALTER* einer Nachricht und besitzt nur den Wert *NEU*. Versuche haben gezeigt, dass der Einfluss von Informationen, die älter als 10 Minuten sind, kaum noch messbar sind. Dies führt zu einem maximalem Alter von 10.

In der Interferenz Phase werden die Fuzzylogik-Regeln 3 benutzt. Dabei werden die *ANTWORTZEITEN* verwendet um die Veränderung der *SERVERANZAHL* zu bestimmen. Das *ALTER* wird benutzt, um den Einfluss von alten Werten zu verringern. Je älter ein Wert ist, desto mehr wird die *SERVERANZAHL* auf *KONSTANT* gesetzt.

Bei einer langsamen Antwortzeit, die mit 150 ms genau die Hälfte der geforderten Antwortzeit entspricht, wird die Serveranzahl verdoppelt. Bei einer schnellen Zeit von 600 ms, wird die Anzahl halbiert. Daraus ergibt sich die in Tabelle 4.3 angegebenen Ausgangswerte der Fuzzylogik.

```
wenn ANWORTZEIT LANGSAM und ALTER NEU dann SERVERANZAHL VERGROESSERN
wenn ANWORTZEIT LANGSAM und nicht ALTER NEU dann SERVERANZAHL KONSTANT
wenn ANWORTZEIT GUT und ALTER NEU dann SERVERANZAHL KONSTANT
wenn ANWORTZEIT SCHNELL und nicht ALTER NEU dann SERVERANZAHL KONSTANT
wenn ANWORTZEIT SCHNELL und ALTER NEU dann SERVERANZAHL VERRINGERN
```

Fuzzylogik-Regeln 3: Verteilte Cloud Steuerung

Linguiste Variable	Ausgabe
VERRINGERN	- 50%
KONSTANT	0
VERGROESSERN	100%

Tab. 4.3.: Fuzzylogik Ausgabewert SERVERANZAHL

Abbildung 4.3 zeigt den Graphen, der durch diese Zugehörigkeitsfunktionen und Regeln definierten Fuzzylogik. Da bei einer Antwortzeit kleiner 200 oder größer 500 ms nur eine Zugehörigkeitsfunktion aktiv ist, wird ab diesen Werten bereits die Serveranzahl halbiert bzw. verdoppelt. Je älter eine Information ist, desto weniger stark ist die Änderung. Bei einem Alter von 10 Minuten, wird die Serveranzahl nur um $\frac{1}{11}$ angepasst. Die Eingangs- und Ausgangswerte der Fuzzylogik sind begrenzt. Sollten die Zeiten beispielsweise unter 150 ms fallen, so wird in der Fuzzylogik mit 150 ms gerechnet.

Da bei der Cloud-Steuerung mehrere Werte bearbeitet werden müssen, wird für jeden Wert durch die Fuzzylogik eine Stellgröße bestimmt und anschließend über alle Ausgangswerte ein gewichteter Mittelwert gebildet. Dabei werden die Fuzzylogik-Ergebnisse anhand ihres Alters gewichtet. Dieser Mittelwert stellt das Ergebnis der Fuzzylogik-Steuerung dar.

4.3.3. Aggregation von Informationen

Um die Netzwerkbelastung zu minimieren, werden die arithmetische und die Fuzzylogik-Steuerung zusätzlich noch mit aggregierten Informationen evalu-

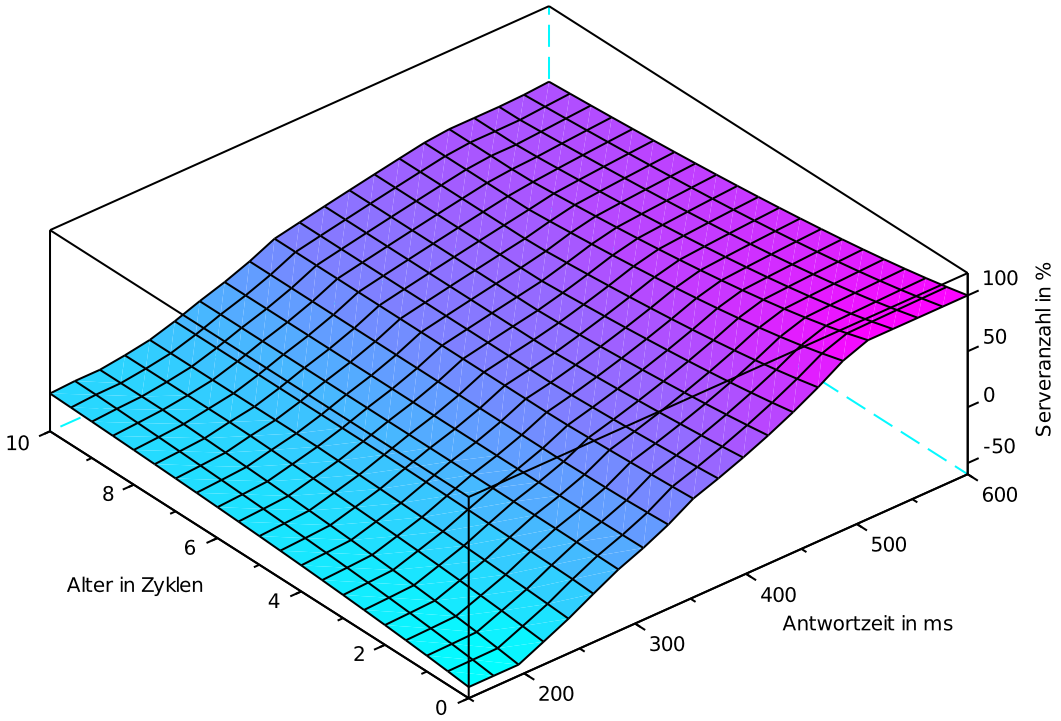


Abb. 4.3.: Graph der verwendeten Fuzzylogik

iert. Dabei wird von allen eingehenden Lastinformationen der Mittelwert gebildet und dieser beim nächsten Sendeoperation weitergeleitet. Formel 4.4 wird benutzt, um aus den eingehenden Messwerten den Ausgangswert zu bestimmen.

$$Wert = \frac{\sum Messwerte}{Anzahl} \tag{4.4}$$

Auf diese Weise erhält jeder Knoten nicht mehr ein genaues Bild der Auslastung anderer Knoten, sondern nur den Mittelwert. Der aggregierte Wert der Nachbarknoten enthält im nächsten Sendezyklus die eigenen gesendeten Informationen wieder, aber nur noch zu einem niederen Prozentsatz. Auf diese Weise haben ältere Werte noch einen geringen Einfluss auf das System, der sich mit jedem Sendezyklus verringert. Dies verhindert das Oszillieren des Gesamtsystems, führt allerdings zu längeren Reaktionszeiten.

4.4. Evaluation

Um die verschiedenen Ansätze zu vergleichen, wurde in dem Netzwerksimulator ns3 [28] ein Rechenzentrum simuliert. Hierbei wird angenommen, dass mehrere Server ein Rack bilden. Die Netzwerkverbindung in Racks ist mit 10 Gb/s sehr hoch. Die unterschiedlichen Racks sind mit einer Geschwindigkeit von 1 Gb/s verbunden. Als Anwendung wird eine Cloud-Computing-Anwendung benutzt, die Webseiten verteilt erstellt. Es werden Load Balancer simuliert, welche die Anfragen an die laufenden Instanzen der Anwendung verteilen.

Als Ziel wird für diese Arbeit eine Antwortzeit von 300 ms gewählt. Da die Antwortzeit jeder einzelnen Anfrage variieren kann, wird der Mittelwert der letzten Anfragen ermittelt. Zusätzlich wird ein Intervall um den Zielwert definiert, in dem das System sich nicht verändert. Als Intervall wurde [290 ms, 310 ms] benutzt.

Die Steuerung wird jede Minute aktiv und optimiert das System, falls es notwendig ist. Bei der dezentralen Steuerung berechnet jeder aktive Knoten wie das Netzwerk angepasst werden muss. Sollte die Anpassung zu gering ausfallen, wird keine Aktion durchgeführt. Falls mehrere Knoten Änderungen durchführen wollen, wird nur die Änderung des schnellsten Knotens angewendet. Nach einer Anpassung des Systems werden für drei Minuten keine weiteren Anpassungen durchgeführt. Dies ist notwendig, da ansonsten zu viele Schwankungen im System stattfinden.

Bei dem hier verwendeten Beispiel einer Public-Cloud ist der Cloud-Computing-Anbieter die zentrale Stelle für die Anpassung der Cloud. Hier könnte mit Hilfe des Anbieters eine Wartezeit bis zum Akzeptieren der nächsten Anpassung implementiert werden. Sollte der Cloud-Computing-Anbieter solch eine Wartezeit nicht implementieren oder eine Private-Cloud verwendet werden, so muss ein verteilter Lock (z.B. [39] mit $\mathcal{O}(\sqrt{N})$ Nachrichten bei N Knoten) implementiert werden.

Zur Implementierung der Fuzzylogik wurde die fuzzylite-Bibliothek [49] benutzt. Es wird ein Zeitraum von 2 Stunden evaluiert. Die nächsten beiden Abschnitte beschreiben, wie das Lastverhalten der Webserver und das Besucherverhalten modelliert wurden. Anschließend wird eine Referenzimplemtierung vorgestellt, wie sie zur Zeit in Rechenzentren benutzt wird. Im vierten Abschnitt wird die Implementierung des Cloud-Computing-Szenario genauer

vorgestellt. Der fünfte Abschnitt zeigt die Ergebnisse, welche dann im letzen Abschnitt diskutiert werden.

4.4.1. Webserver Lastverhalten

Als Optimierungsgröße des Evaluierungsszenarios wird die Antwortzeit des Webservers auf eine Anfrage verwendet. Um eine realistische Simulation zu erhalten, wurde die Antwortzeit einer einfachen Webanwendung bei unterschiedlichen gleichzeitigen Zugriffen untersucht. Als Webserver wird ein Intel Core Quad Q9550 mit 2,83 Ghz und 8 GB Arbeitsspeicher benutzt. Als Betriebssystem wird Ubuntu 13.10[1] benutzt und Apache 2.4.6 mit MySQL 5.5.35 installiert. Wordpress 3.8.1[2] wird als Anwendung mit PHP 5.5.3 betrieben.

Von einem anderen Rechner im gleichen lokalen Netzwerk werden mit dem Linux-Tool *siege*[3] mehrere gleichzeitige Zugriffe auf die Startseite der Wordpress-Installation durchgeführt. Die Last wird für 5 Minuten gehalten und dann erhöht. Die komplette Messung wurde zehnmal wiederholt und gemittelt.

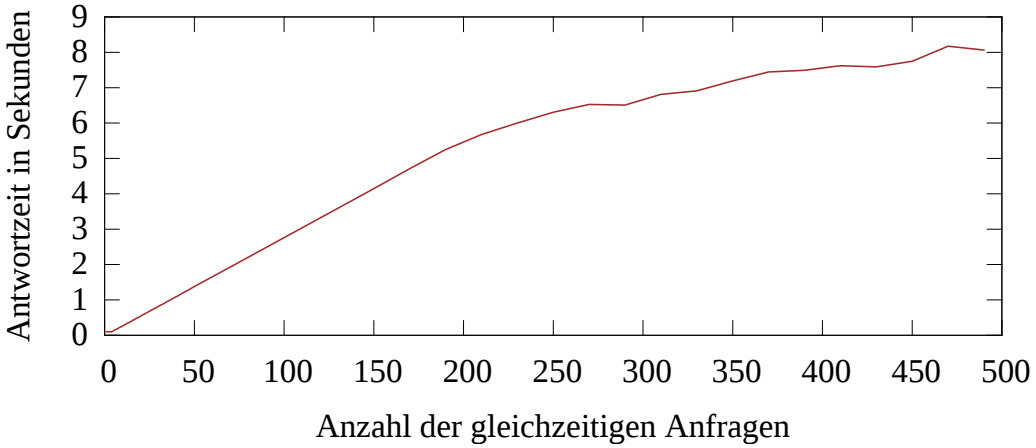


Abb. 4.4.: Antwortzeit der erfolgreichen Anfragen

Abbildung 4.4 zeigt die Antwortzeit des Webserver bei unterschiedlich vielen gleichzeitigen Anfragen. Hierbei werden nur Anfragen berücksichtigt, die eine Rückantwort liefern. Bis zirka 180 gleichzeitigen Anfragen steigt die Antwort-

[1] `http://www.ubuntu.com`
[2] `http://wordpress.com`
[3] `http://www.joedog.org/siege-home/`

zeit linear an. Bei mehr als 180 Anfragen steigt die Antwortzeit der Anwendung nur noch langsam.

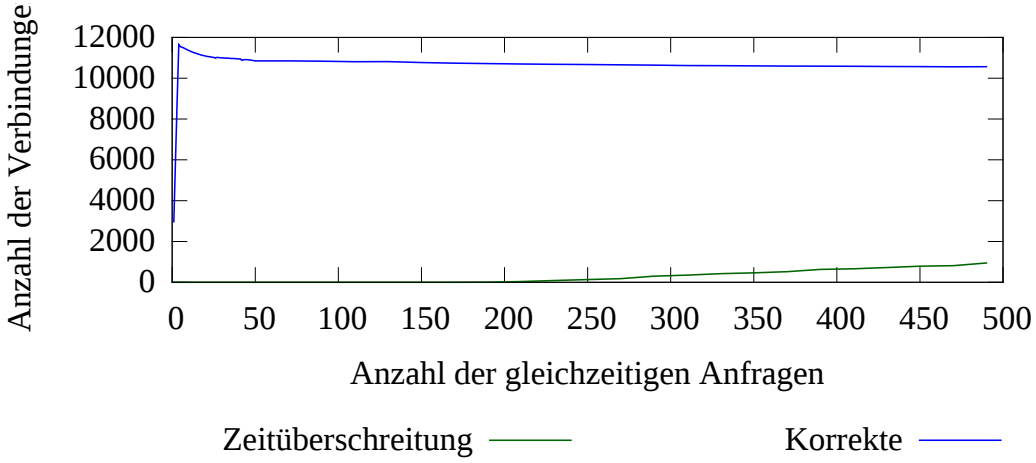


Abb. 4.5.: Anzahl der Anfragen mit Zeitüberschreitung

Der Grund für die langsame Steigung der Antwortzeiten zeigt Abbildung 4.5. Dort sind die Anzahl der korrekt bearbeiteten und der abgebrochen Anfragen zu sehen. Dabei ist erkennbar, dass bis zirka 180 gleichzeitigen Anfragen keine Zeitüberschreitung auftritt und alle Anfragen beantwortet werden. Ab 180 gleichzeitigen Anfragen steigt die Anzahl der nicht bearbeiteten Anfragen, die aufgrund einer Zeitüberschreitung abgebrochen wurden.

Diese Evaluation hat gezeigt, dass für die Antwortzeit eines Webservers eine lineare Funktion benutzt werden kann, solange der Webserver noch alle Anfragen bearbeiten kann. Da die Webserver bei Cloud-Computing-Anbietern auf das schnelle Bearbeiten von Anfragen optimiert sind, werden für die Evaluation nicht die ermittelten Antwortzeiten übernommen, sondern nur der Trend verwendet und die Werte angepasst. In der Evaluation wird die Formel 4.5 zur Ermittlung der Antwortzeit benutzt. 40 ms ist die reine Rechenzeit einer Anfrage. Für jede weitere gleichzeitige Anfrage verlangsamt sich die Antwortzeit um 5ms.

$$Antwortzeit = 40ms + 5ms \cdot \#Verbindungen \tag{4.5}$$

4.4.2. Besucherverhalten

Um das Besucherverhalten zu modellieren, wurden öffentlich verfügbare Logdateien von *The Internet Traffic Archive*[4] untersucht. Es wurden Logdateien aus dem Jahr 1995 verwendet, welche die Benutzung über eine Woche oder einen Monat zeigen. Es werden auf der Seite keine neueren brauchbaren Traffic-Logs bereitgestellt. Die Abbildungen 4.6 zeigen vier dieser Logdateien. Für die Auswertung wurden die Zugriffe pro Minute gezählt. Die ersten beiden Graphen stellen die Besucher eines öffentlichen NASA-Servers während den Monaten Juli und August dar. Die beiden unteren Graphen zeigen die Besucher des Webservers eines Internetanbieters aus Baltimore-Washington DC.

In den vier Grafiken ist deutlich erkennbar, dass die Zugriffe pro Minute stark schwanken und die Punkte eine dicke Linie bilden. Obwohl die minütlichen Zugriffszahlen stark schwanken, ist ein Trend erkennbar. Die Anzahl der Zugriffe ist tageszeitabhängig. Bei allen vier Logdateien zeigt sich, dass tagsüber die meisten Zugriffe erfolgen und nachts die wenigstens.

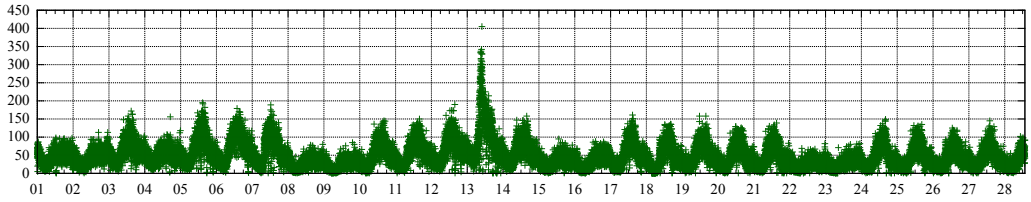
Da die Logdateien aus dem Jahr 1995 stammen, ist anzunehmen, dass sich die Anzahl der Zugriffe stark vergrößert hat. Da der Trend aber gleich geblieben ist, werden für die Simulation der Benutzerzugriffe zwei überlagerte Sinusfunktionen benutzt. Die Funktion mit der langen Periode stellt den tageszeitabhängigen Trend dar und die mit der kurzen Periode simuliert die Schwankungen der minütlichen Anfragen.

$$Besucher = 9000 \cdot \sin(t/700) + A \cdot \sin(t/25) \cdot +10000 \tag{4.6}$$

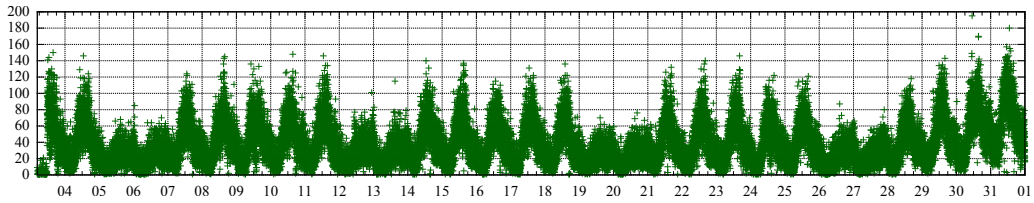
$$Besucher = 9000 \cdot \sin(t/700) + A \cdot \sin(t/25) + 10000 + t \tag{4.7}$$

In der Evaluierung werden die Formeln 4.6 und 4.7 verwendet. t ist die aktuelle Simulationszeit in Sekunden. Bei einer Evaluationszeit von 2 Stunden werden eineinhalb Schwingungen der langsamen Sinusfunktion erzeugt. Zusätzlich wird eine Sinusfunktion mit höherer Frequenz addiert. Die Wellenlänge beträgt hier zirka 1,25 Sekunden. Die Amplitude der langsamen Schwingung beträgt 9000 Besucher, die der schnellen wird vom Parameter A festgelegt. Bei Funktion 4.6 werden noch 10000 Besucher addiert, was zu einer Schwingung um diesen Wert mit ± 9000 führt. Funktion 4.7 addiert zusätzlich noch

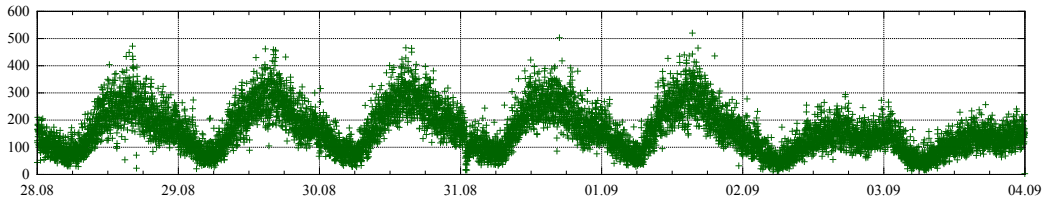
[4] `http://ita.ee.lbl.gov/`



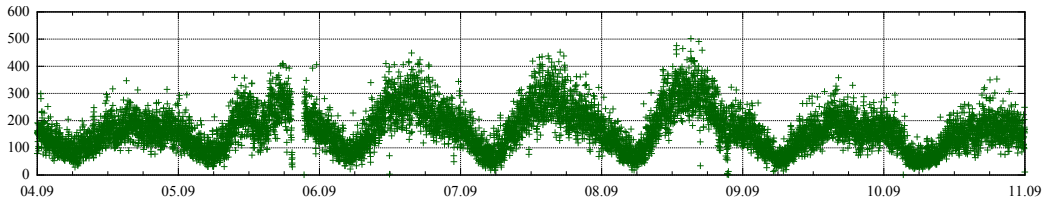
(a) Traffic NASA für den Juli 1995



(b) Traffic NASA für den August 1995



(c) Traffic Clarknet für eine Woche ab 28. August 1995



(d) Traffic Clarknet für eine Woche ab 4. September 1995

Abb. 4.6.: Traffic Auswertungen in Besucher pro Minuter von öffentlichen Logdateien

die Zeit. Dies führt dazu, dass die Besucherzahlen schwanken, aber einen positiven Trend besitzen. Die in der Simulation verwendeten Parameter für A können aus Anhang A.2 entnommen werden.

4.4.3. Referenzimplementierung

Um die Qualität der verschiedenen Algorithmen zu bewerten, wurde eine zentrale Steuerung implementiert. Dabei übernimmt eine Instanz die Überwachung und Analyse. Alle 60 Sekunden werden Anfragen an alle aktiven Instanzen gesendet. Diese antworten mit einer kurzen Statusmeldung, welche die aktuelle Antwortzeit enthält. Die gemittelte Reaktionszeit wird mit Hilfe der Formel 4.2 des arithmetischen Ansatzes aus den Antworten berechnet. Da alle Nachrichten gleich alt sind, kürzt sich die Formel auf den arithmetischen Mittelwert. Anhand dieses Durchschnitts wird nun berechnet, wie viele Instanzen gestartet bzw. gestoppt werden können. Dazu wird wie beim arithmetischen Ansatz das Verhältnis von der gemessenen zur gewünschten Antwortzeit mit Formel 4.3 bestimmt und die Anzahl der Instanzen um diesen Faktor angepasst.

4.4.4. Szenarien

Zusätzlich zur zentralen Referenzimplementierung gibt es noch eine zentrale Fuzzylogik-Steuerung. Sie entspricht der in Kapitel 4.3.2 vorgestellten Methode. Da alle Nachrichten gleich alt sind, wurde auch hier das Alter nicht betrachtet.

Die restlichen Methoden benutzen eine dezentrale Steuerung. Die Reaktionszeiten der Webserver werden dabei durch die im letzten Kapitel vorgestellte Peer-to-Peer-Verteilungsalgorithmen auf Chord-Basis weitergeleitet. Bei den dezentralen Steuerungen existieren jeweils arithmetische und Fuzzylogik-Ansätze. Außerdem wird untersucht, welche Auswirkungen aggregierte Statusinformationen auf die Steuerung haben. Dazu wird jedes Verfahren mit und ohne Aggregation untersucht. Daraus ergeben sich vier dezentrale Ansätze, die untersucht werden.

Es wurden 9 Szenarien mit unterschiedlichem Besucherverhalten untersucht. In Szenario A verdoppeln sich die aktiven Benutzer von 7500 auf 15000 Be-

sucher nach 15 Minuten. Die Steuerung muss dies schnell erkennen und die Anzahl der Instanzen anpassen. Es gibt keine zusätzlichen Schwankungen der Besucherzahlen.

In Szenario B werden die Besucherzahlen halbiert. Auch hier wird mit 7500 Besuchern gestartet. Nach 15 Minuten halbiert sich die Zahl auf 3750. Wieder gibt es keine zusätzlichen Schwankungen.

Für die Simulation der restlichen Besucherzahlen werden die Formeln 4.6 und 4.7 aus dem Kapitel 4.4.2 verwendet. Szenario XA benutzt Formel 4.6. Mit ihr wird eine Schwankung der Besucherzahlen simuliert ohne linearen Faktor. Daraus ergibt sich eine neue Herausforderung an die Steuerungsalgorithmen. Es muss nicht mehr sofort auf jede Schwankung reagiert werden. Wichtiger ist es, sich dem Trend anzupassen. Zusätzlich sinken hier die Anfragen stark ab.

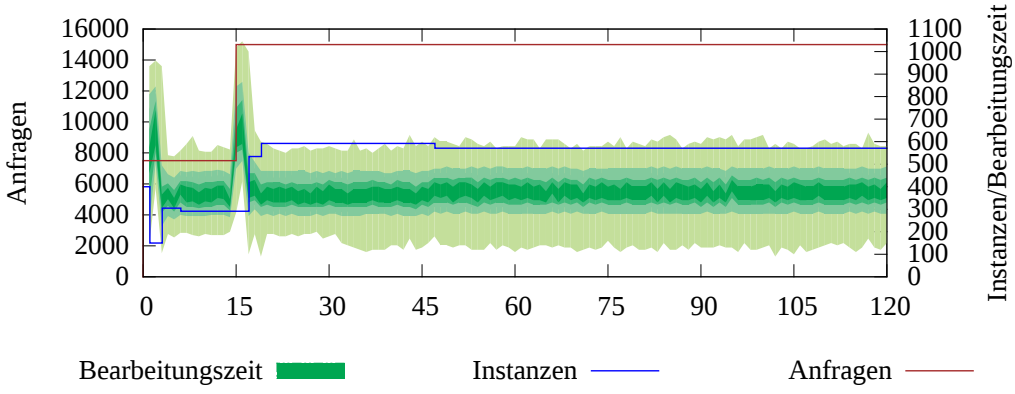
In den letzten Szenarien kommt zu den beiden Sinus-Schwingungen noch eine linear steigende Funktion hinzu. Hier soll ein Anstieg der Besucherzahlen mit starken Schwankungen simuliert werden. Auch hierbei muss der Trend erkannt werden, statt auf jede kleine Änderung zu reagieren. Die genauen Formeln können den Überschriften des Anhangs A.2 entnommen werden.

4.4.5. Ergebnisse

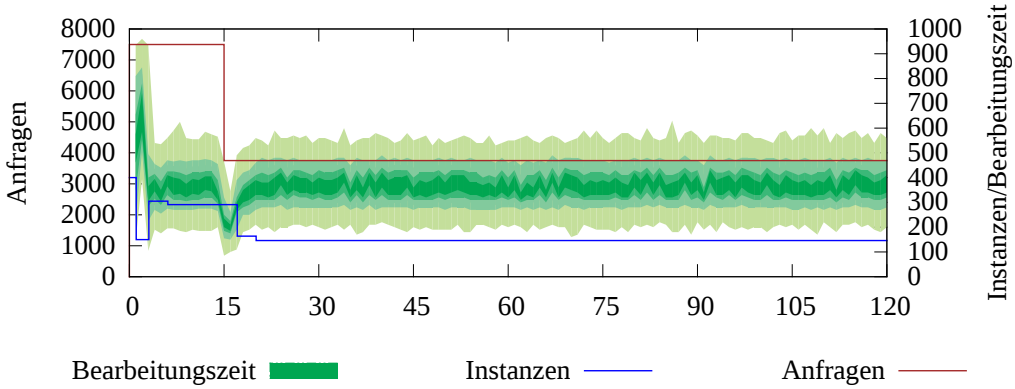
In diesem Abschnitt werden einige Ergebnisse der Cloud-Computing-Evaluation vorgestellt und untersucht. Die kompletten Ergebnisse aller durchgeführten Simulationen sind im Anhang A ab Seite 105 zu finden.

Die Abbildungen 4.7 zeigen das Verhalten der Referenzimplementierung bei Verdoppelung und Halbierung der Besucheranfragen. Die obere Grafik 4.7a zeigt das erste Szenario mit der Verdoppelung der Besucheranfragen. In der unteren Grafik 4.7b ist die Halbierung der Besucheranfragen dargestellt. Die braune Linie repräsentiert jeweils die Anzahl der Anfragen und ist an der linken Skala abzulesen. Die Anfragen sind konstant, bis sich nach 15 Minuten die Verdoppelung bzw. Halbierung ereignet. Die x-Achse stellt den zeitlichen Verlauf dar. Die angegebenen Werte sind Minuten.

Die Bearbeitungszeit und Anzahl der Instanzen sind an der rechten Skala angegeben. Die grünen Flächen stellen die Häufung der Antwortzeiten dar. In der hellgrünen Fläche liegen 80% der Antwortzeiten. Es wurden 10% der oben und unteren Ausreißer abgeschnitten. Je dunkler der Farbton wird, desto weni-



(a) Szenario A: Verdoppelung der Besucherzahlen

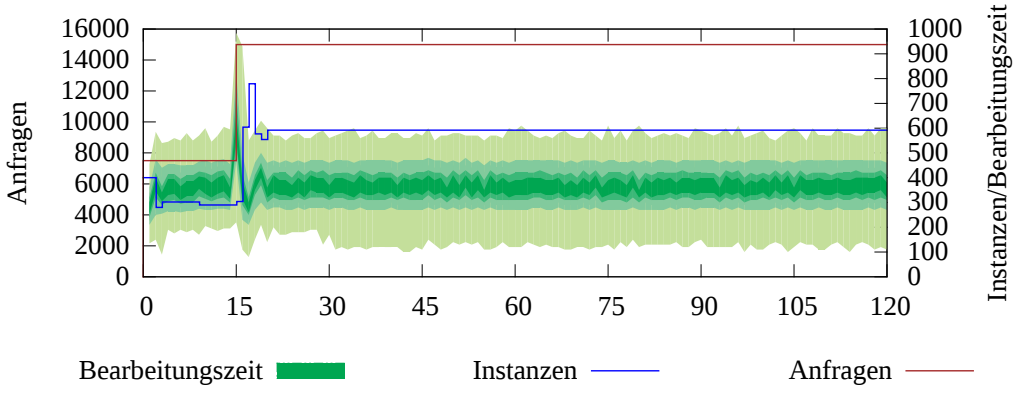


(b) Szenario B: Halbierung der Besucherzahlen

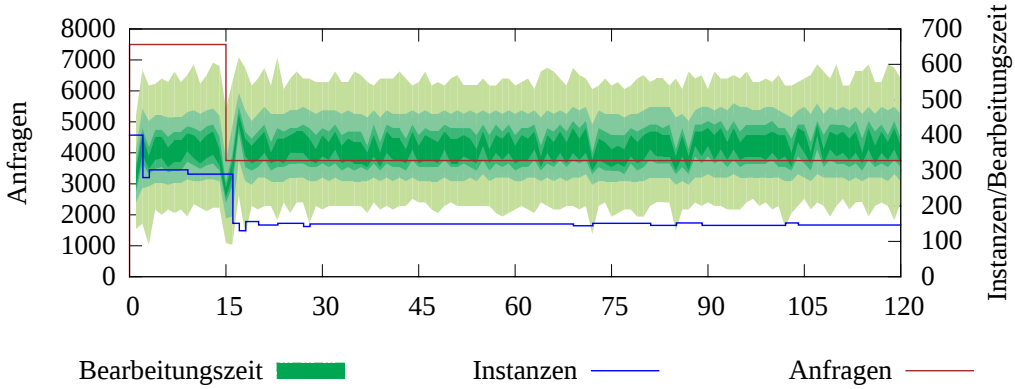
Abb. 4.7.: Zentrale Referenzimplementierung

ger Prozent der Antwortzeiten sind enthalten. Pro Farbtonänderungen werden 20% weniger betrachtet. Der dunkelste Farbton entspricht der Häufung der mittleren 20% der Antwortzeiten. Eine gute Steuerung hält diese dunkle Fläche nahe der Zielzeit von 300 ms. Es ist deutlich zu sehen, dass die Last der Server nicht gleichmäßig verteilt ist und die Bearbeitungszeiten daher sehr unterschiedlich ausfallen. Die Antwortzeit ist die Regelgröße und soll konstant auf 300 ms gehalten werden. Falls die Bearbeitungszeit von 300 ms abweicht, greift die Steuerung ein und startet oder stoppt Webserver-Instanzen. Ein Häufung ist zwischen 300 und 400 ms zu erkennen.

Die blaue Linie entspricht der Anzahl der aktiven Webserver. Die Steuerung ändert diesen Wert, um die Antwortzeit konstant zu halten. Daher ist dies die Stellgröße des Systems. Eine gute Steuerung passt die Serveranzahl nach einer



(a) Szenario A: Verdoppelung der Besucherzahlen

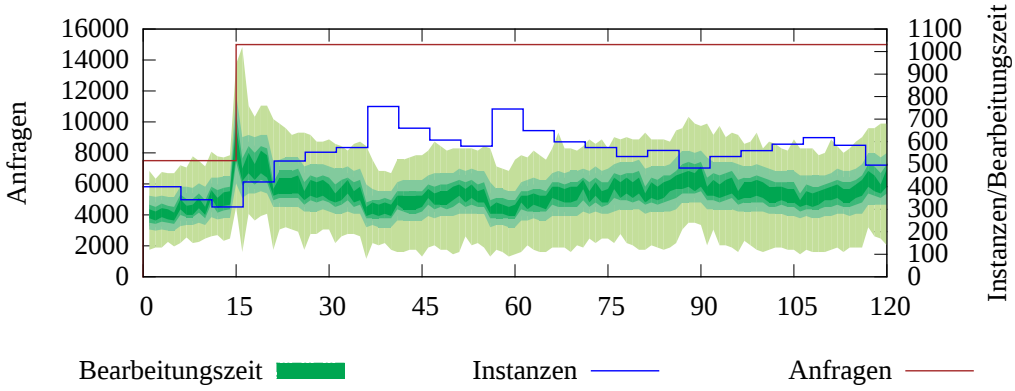


(b) Szenario B: Halbierung der Besucherzahlen

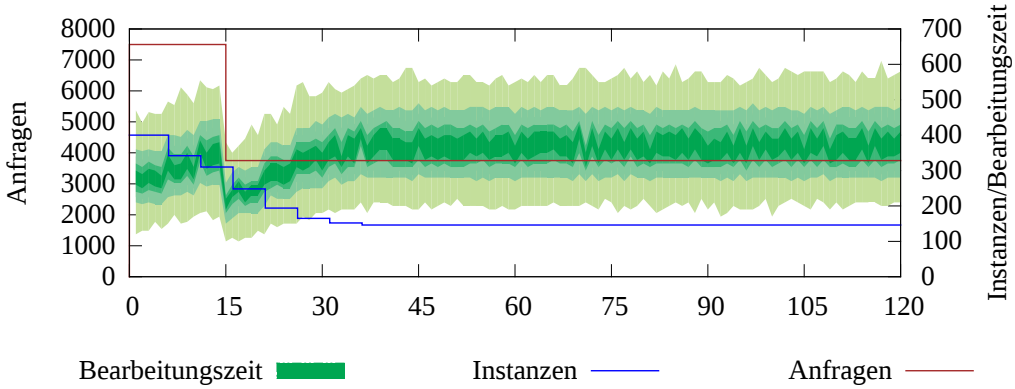
Abb. 4.8.: Zentrale Fuzzylogik

Änderung der Besucherzahlen schnell an, ohne Abweichung der Antwortzeit. Da das System allerdings eine Änderung der Antwortzeit benötigt um einen Wechsel im Besucherverhalten zu erkennen, ist dies nicht möglich. Daher sind kleine Änderungen der Antwortzeit akzeptabel und in der Systemmodellierung vorgesehen.

Der zentrale Ansatz zeigt ein gutes Verhalten. Hierbei bleibt die Anzahl der Webserver konstant und wird nur bei einer Änderung der Besucherzahlen angepasst. Die meisten Anfragen werden mit einer Antwortzeit um die 300 ms beantwortet. Es ist erkennbar, dass das System auf Änderungen schnell reagiert und keine Schwankungen stattfinden. Abbildung 4.7a zeigt nur zwei Korrekturen nach der Anpassung, Abbildung 4.7b zeigt nur eine.



(a) Szenario A: Verdoppelung der Besucherzahlen

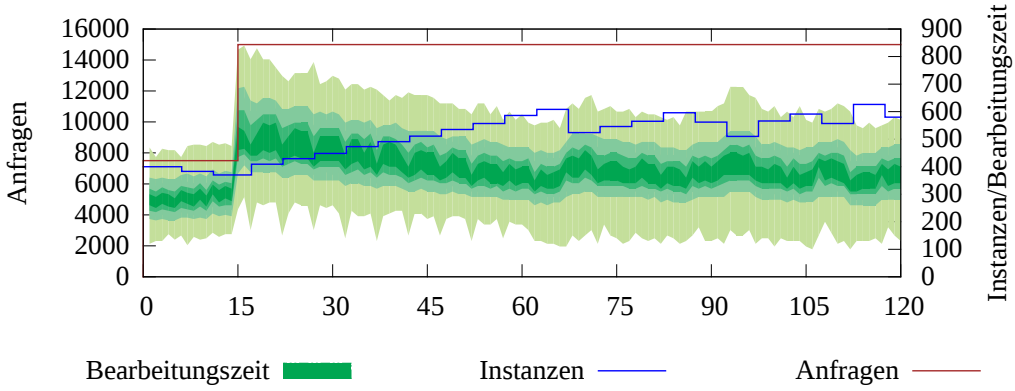


(b) Szenario B: Halbierung der Besucherzahlen

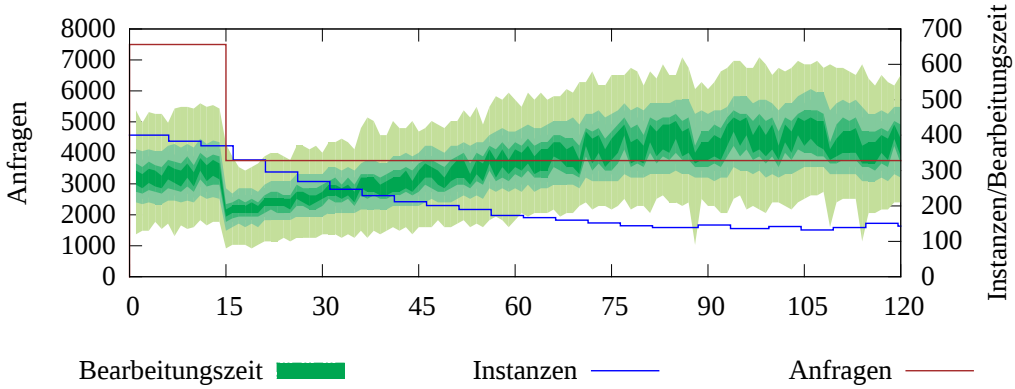
Abb. 4.9.: Dezentrale Arithmetische Methode ohne Aggregation

Der Grund für die schnelle Anpassung ist das globale Wissen. Da die zentrale Überwachungsinstanz jede Minute von allen beteiligten Servern die Lastdaten sammelt, sind die Daten immer aktuell. Die Steuerung der Antwortzeiten ist einfacher, wenn weniger Instanzen gestartet sind. Dies ist in der Abbildung 4.7b deutlich zu erkennen.

Die Abbildungen 4.8 zeigen das Verhalten der zentralen Fuzzylogik-Steuerung. Es ist ersichtlich, dass auch hier schnell auf Änderungen reagiert wird. Die Steuerung verhält sich aber instabiler, obwohl globales Wissen über das System vorliegt. In beiden Szenarien kommt es zu leichten Schwankungen der Instanzenanzahl, obwohl sich die Besucherzahlen nicht ändern. Die Anzahl an maximal gleichzeitig gestarteten Instanzen in Abbildung 4.8a ist höher als bei der zentralen Referenzimplementierung. Der Spitzenwert wird nur kurz



(a) Szenario A: Verdoppelung der Besucherzahlen

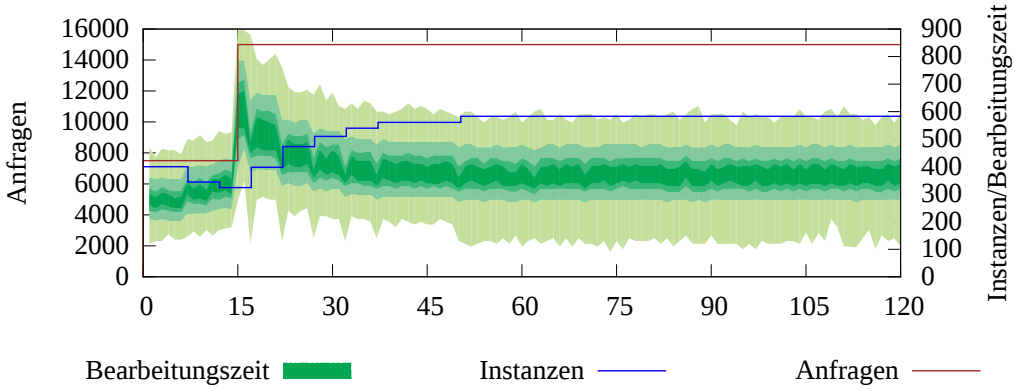


(b) Szenario B: Halbierung der Besucherzahlen

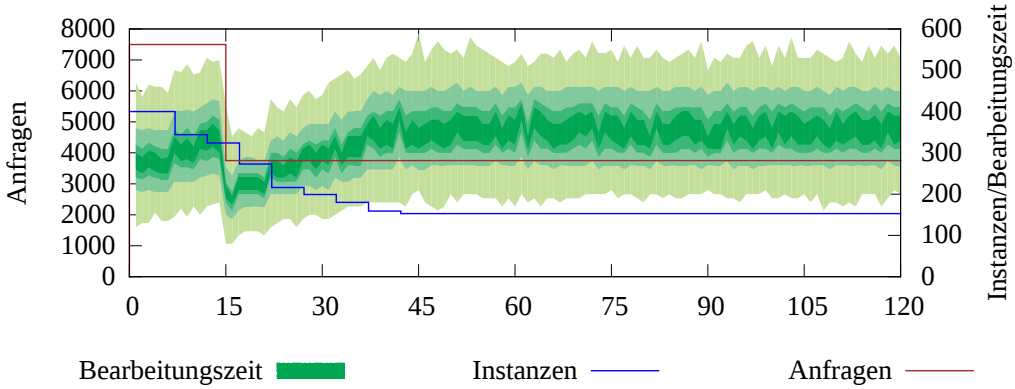
Abb. 4.10.: Dezentrale Arithmetische Methode mit Aggregation

gehalten und sinkt dann auf ein Niveau, das vergleichbar ist, mit den Werten der zentralen Referenzimplementierung. In Abbildung 4.8b werden bei der Halbierung der Zugriffe ungefähr gleich viele Instanzen wie bei der zentralen Steuerung gestartet.

Die Abbildung 4.9 zeigt die Ergebnisse des ersten dezentralen Ansatzes. Hierbei handelt es sich um die arithmetische Steuerung ohne Aggregation. Es ist deutlich erkennbar, dass die Instanzen langsam angepasst werden. In Abbildungen 4.9a wird das Maximum der aktiven Server bei 40 Minuten erreicht. Es sind deutliche Schwankungen erkennbar. In Abbildung 4.9b stabilisiert sich das System schneller und es sind keine Schwankungen erkennbar. Damit ist der arithmetische Ansatz ohne Änderungen auch für den dezentralen Einsatz



(a) Szenario A: Verdoppelung der Besucherzahlen

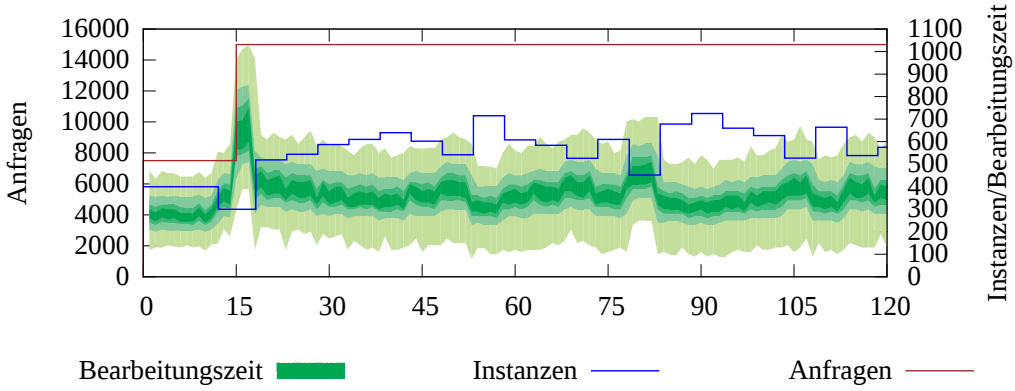


(b) Szenario B: Halbierung der Besucherzahlen

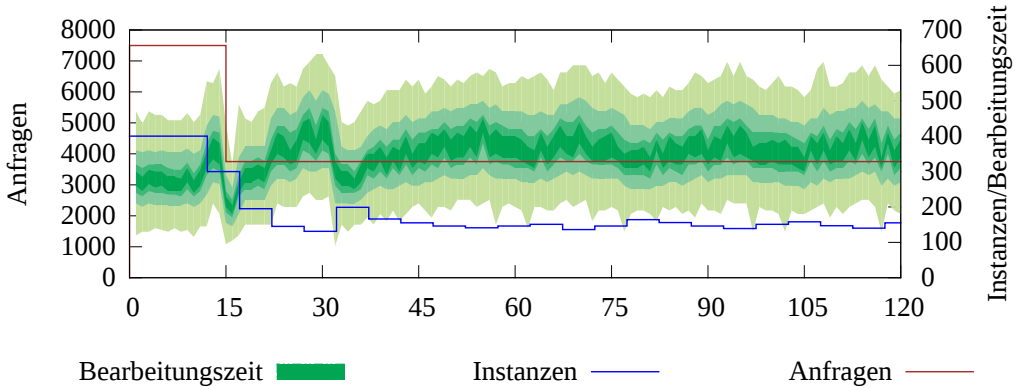
Abb. 4.11.: Dezentrale Fuzzylogik-Steuerung ohne Aggregation

geeignet. Es wird die Reaktionszeit verlängert und zusätzliche Schwankungen erzeugt, aber das System stabilisiert sich.

In den Abbildungen 4.10 werden die Ergebnisse des dezentralen arithmetischen Ansatzes mit aggregierten Informationen dargestellt. Hier ist zu erkennen, dass die Anzahl der Instanzen stark schwankt und langsamer auf Änderungen reagiert. Abbildung 4.10a zeigt, dass das Maximum der Instanzen erst bei 70 Minuten erreicht wird. Im Gegensatz zu der vorherigen Evaluation ohne aggregierten Werten, kommt es hier zu häufigeren Schwankungen. Durch die Aggregation verlangsamt sich die Reaktionszeit der arithmetischen Methode in beiden Szenarien.



(a) Szenario A: Verdoppelung der Besucherzahlen



(b) Szenario B: Halbierung der Besucherzahlen

Abb. 4.12.: Dezentrale Fuzzylogik-Steuerung mit Aggregation

Abbildung 4.11 zeigt die Ergebnisse der dezentralen Steuerung mit Fuzzylogik. Hierbei werden alle Informationen weitergeleitet und nicht aggregiert. Nach der Stabilisierung das Systems finden keine Schwankungen mehr statt.

Mit der Fuzzylogik-Steuerung werden die Änderung der Besucherzahlen sehr schnell erkannt und entsprechend Instanzen gestartet bzw. gestoppt. In Abbildung 4.11a stabilisiert sich das System erst bei 50 Minuten. Nach der Änderung der Anfragen reagiert das System schnell mit einer großen Anpassung. Anschließend werden noch mehrere kleine Anpassungen vorgenommen, die zusätzlich Zeit benötigen. Trotz dieses Verhaltens sind die Antwortzeiten stabiler als bei den beiden dezentralen arithmetischen Methoden.

Abbildung 4.12 zeigt die Ergebnisse der dezentralen Fuzzylogik-Steuerung mit aggregierten Informationen. Nach der Änderung der Besucherzahlen wird zeit-

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	12,19	856	285	1078,33	0,0113
zentral Fuzzylogik	10,34	949	357	1120,43	0,0092
dezentral Arithmetisch	11,75	1173	677	1110,53	0,0106
dezentral Arithmetisch aggr	13,45	860	280	1037,76	0,0130
dezentral Fuzzylogik	12,42	659	76	1073,25	0,0116
dezentral Fuzzylogik aggr	12,05	1444	869	1135,00	0,0106

Tab. 4.4.: Statistische Kenngrößen bei Verdoppelung der Besucher (Szenario A)

nah mit einer sprunghaften Anpassung der Instanzen reagiert. Das System ist bei konstanten Besucherzahlen instabiler, da durch die Aggregation die Genauigkeit der Messwert abnimmt.

Da die Graphen der einzelnen Szenarien nicht ausreichen um einen Vergleich durchzuführen, wurden noch statische Auswertungen durchgeführt. Tabelle 4.4 zeigt die zusätzlichen Auswertungen zu Szenario A mit den verdoppelten Besucherzahlen. Pro Zeile wird ein anderer Steuerungsalgorithmus betrachtet. Der untersuchte Algorithmus ist in der ersten Spalte angegeben. In der zweiten Spalte wird die in der Statistik benutzte Standardabweichung σ, welche die Abweichung der gemessen Antwortzeit zu der angepeilten Antwortzeit von 300 ms angibt, eingetragen. Je kleiner die Zahl ist, desto weniger Abweichung wurde festgestellt. Die geringste Standardabweichung besitzt die zentrale Fuzzylogik-Steuerung mit 10,34 ms. Danach folgt die dezentrale Fuzzylogik mit Aggregation und einer Standardabweichung von 12,19 ms. Der zentrale arithmetische Ansatz belegt mit einer Standardabweichung von 11,75 ms den dritten Platz. Alle anderen Steuerungen besitzen eine größere Standardabweichung.

Die nächsten beiden Werte sind die Anzahl der gestarteten und gestoppten Instanzen. Am Anfang werden bei allen Szenarien 400 Instanzen gestartet. Diese Start-Instanzen sind in der Auswertung enthalten. Da bei Cloud-Computing-

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,74	555	409	334,59	0,0351
zentral Fuzzylogik	9,72	481	335	341,32	0,0285
dezentral Arithmetisch	9,25	400	254	364,97	0,0253
dezentral Arithmetisch aggr	9,04	432	289	428,13	0,0211
dezentral Fuzzylogik	8,35	400	247	389,95	0,0214
dezentral Fuzzylogik aggr	9,21	540	385	370,61	0,0248

Tab. 4.5.: Statistische Kenngrößen bei Halbierung der Besucher (Szenario B)

Providern nur die Anzahl der laufenden Instanzen bezahlt werden muss, wurde auch noch dieser Wert betrachtet. Dafür werden in der vierten Spalte die laufenden virtuellen Maschinen pro Stunde angegeben. Dieser Wert berechnet sich durch das Integral über die Anzahl der Instanzen. Je größer der Wert ist, desto mehr Miete muss bezahlt werden. Den kleinsten Wert hat der dezentrale arithmetische Ansatz mit aggregierten Informationen. Da dieser Ansatz aber auch die höchste Standardabweichung aufweist, wurde hier auf Kosten der Antwortzeit gespart. Szenarien mit geringer Standardabweichung müssen für diese guten Werte häufiger VMs starten und schnell reagieren. Daher fällt bei mehreren Instanzen die Standardabweichung geringer aus. Sollten allerdings zu viele Instanzen gestartet werden, hat dies auch negative Auswirkungen auf die Standardabweichung. Da in diesem Szenario die Besucherzahl ansteigt, sollten nur Instanzen gestartet, aber nicht gestoppt werden. Die dezentrale Fuzzylogik-Steuerung stoppt am wenigsten Instanzen.

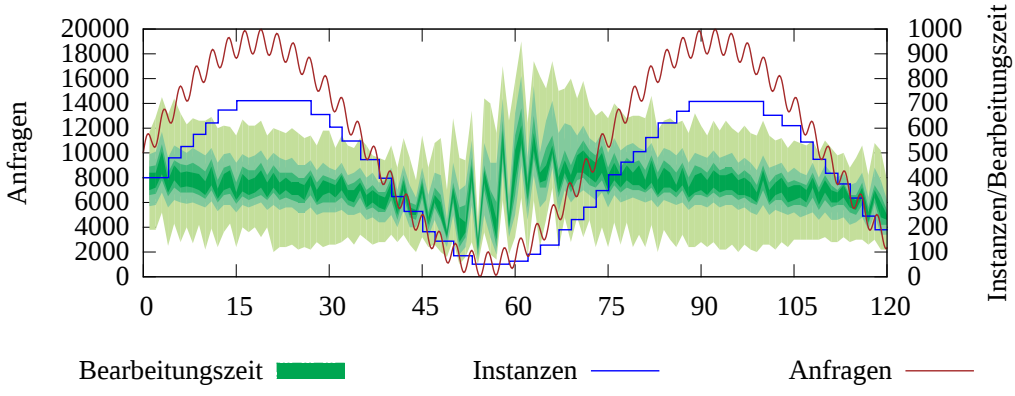
Die letzte Spalte zeigt den Quotienten $\frac{Standardabweichung}{VM/h}$. Je kleiner der Wert ist, desto besser ist das Verhältnis der Standardabweichung zu den verbrauchten Instanzstunden. Optimal ist hier ein möglichst kleiner Wert. Den besten Wert erreichen die zentrale Fuzzylogik-Steuerung. Den zweitbesten Wert erzielen die dezentrale arithmetische Steuerung mit Aggregation und die dezentrale

Fuzzylogik-Steuerung mit Aggregation. Damit sind diese beiden Steuerungen effizienter als die zentrale Referenzimplementierung.

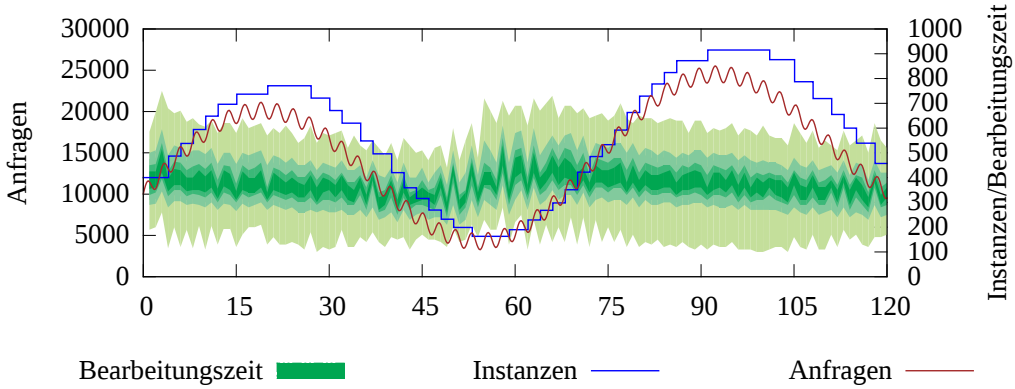
Tabelle 4.5 zeigt die statischen Kenngrößen für das Szenario B. Die Standardabweichungen sind geringer als im Szenario A. Der Grund dafür ist die geringere Anzahl an Instanzen und Anfragen. Dadurch können die Anfragen gleichmäßiger verteilt werden, was zu weniger Ausreißern führt. Die geringste Standardabweichung mit 8,35 besitzt die dezentrale Fuzzylogik-Steuerung ohne Aggregation. Den schlechtesten Wert hat die zentrale arithmetische Steuerung mit einer Standardabweichung von 11,74. Die restlichen 4 Steuerungen haben sehr ähnliche Standardabweichung zwischen 9,04 und 9,72. Die dezentrale arithmetische Steuerung mit Aggregation hat die höchste Anzahl an Instanzen pro Stunde, ebenso besitzt sie den besten Quotienten. Den nächst besten Quotient weist die dezentrale Fuzzylogik-Steuerung auf. Diese ist auch die einzige Steuerung, die keine zusätzlichen Instanzen startet. Es werden nur die am Simulationsanfang benötigten 400 Instanzen gestartet. Damit liefert diese Steuerung das beste Ergebnis in diesem Szenario. Dort sind die dezentralen Steuerungen besser als die beiden zentralen.

Durch die Tabellen hat sich gezeigt, dass bei einer großen Anzahl von Anfragen und Instanzen die Unterschiede zwischen den einzelnen Steuerungen steigen. Bei großen Systemen liefern die zentralen Steuerungen den besten Wert.

Die beiden betrachteten Szenarien zeigen ein sehr stabiles Verhalten. Bis auf eine Änderung nach 15 Minuten bleiben die Besucherzahlen konstant. Um die Steuerungen unter realistischeren Anforderungen zu testen, wurden noch mehrere Szenarien mit schwankenden Besucherzahlen untersucht. Es werden die im Kapitel 4.4.2 vorgestellten Funktionen benutzt, um die Anfragen zu generieren. Im folgenden werden zwei zusätzliche Szenarien vorgestellt. Im Szenario XA werden zwei überlagerte Sinus-Schwingungen benutzt. Die Steuerungsalgorithmen müssen dabei nicht auf jede kleine Änderung reagieren, sondern sich dem Trend der Entwicklung anpassen. In der Mitte des Szenarios sinken die Anfragen stark ab um anschließend wieder zu steigen. Hier wird getestet, wie schnell sich die Algorithmen auf solche großen Schwankungen einstellen können. Im Szenario XB wird diese Schwankung noch durch eine linear steigende Komponente ergänzt. Damit erreicht dieses Szenario die höchsten Besucherzahlen. Hier wird untersucht, wie sich die Algorithmen bei großen Systemen verhalten.



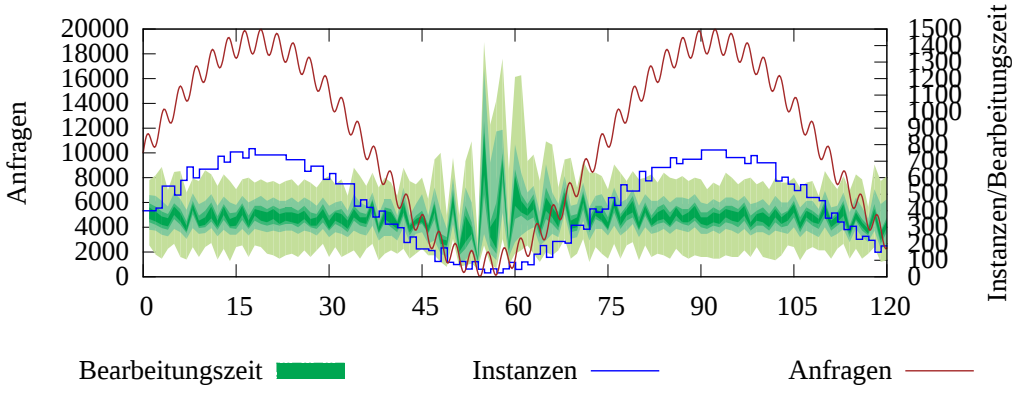
(a) Szenario XA: Schwankende Besucherzahlen



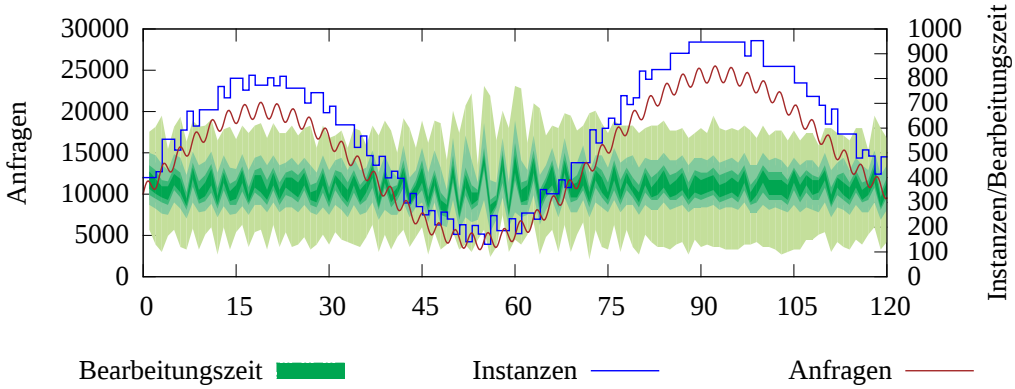
(b) Szenario XB: Steigende schwankende Besucherzahlen

Abb. 4.13.: Zentrale Referenzimplementierung

Abbildung 4.13 zeigt die Ergebnisse der beiden Szenarien für die zentrale arithmetische Methode. Im oberen Graph ist zu erkennen, dass zwischen 50 und 60 Minuten, wenn die Besucherzahlen ihr Minimum erreichen, die Anzahl der Instanzen sehr stark reduziert wird und daher nicht mehr ausreichend schnell auf die Schwankungen reagiert werden kann. Dies führt bei diesem Zeitpunkt teilweise zu sehr langen Bearbeitungszeiten von bis zu 900 ms. Die restliche Zeit kann die Antwortzeit zwischen 200 und 400 ms gehalten werden. Die Anzahl der Instanzen passt sich der langsamen großen Schwingung der Anfragen an. Im unteren Graphen 4.13a ist das vierte Szenario mit den steigenden und schwankenden Besucherzahlen. Die größten Abweichungen der Antwortzeit sind hier ebenfalls zwischen 50 und 60 Minuten zu finden. Hier erreichen die Besucherzahlen ihr globales Minimum. Bei zirka 90 Minuten wird das Ma-



(a) Szenario XA: Schwankende Besucherzahlen



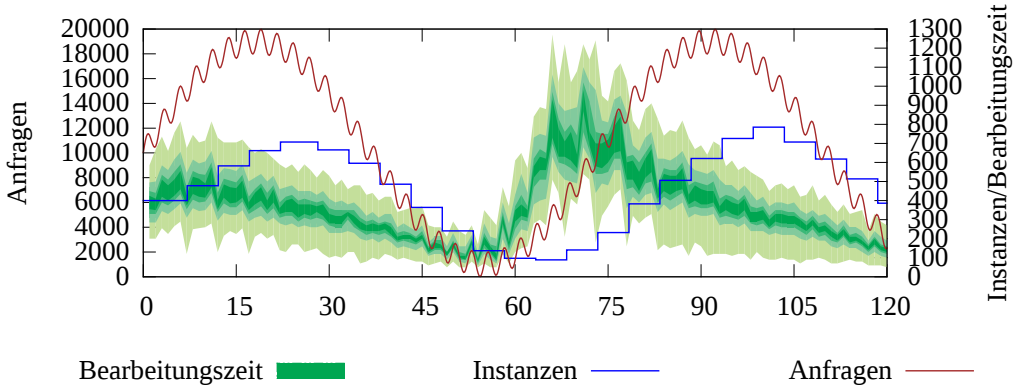
(b) Szenario XB: Steigende schwankende Besucherzahlen

Abb. 4.14.: Zentrale Fuzzylogik

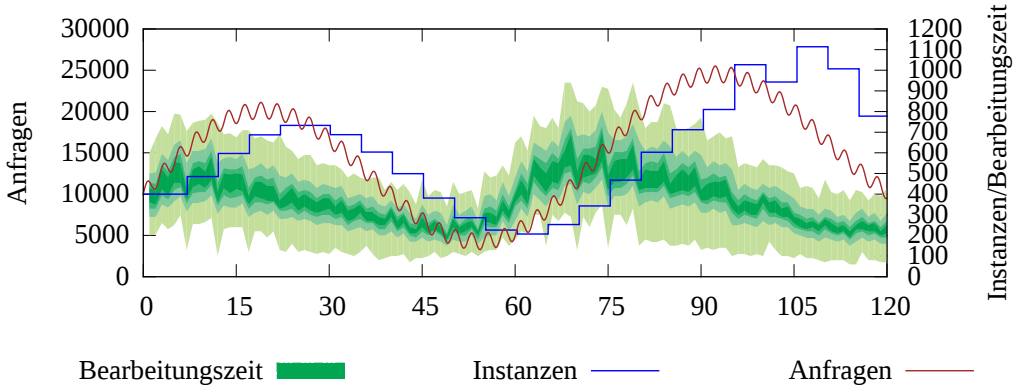
ximum erreicht. Es werden die meisten Anfragen zwischen 200 und 400 ms bearbeitet.

In Abbildung 4.14 sind die Graphen für die zentrale Fuzzylogik zu sehen. Der obere Graph ähnelt dem Graph 4.13a der zentralen arithmetischen Methode. Wenn die Besucherzahlen ihr Minimum erreichen, ist die stärkste Abweichung der Bearbeitungszeit erkennbar. Die Anzahl der Instanzen ist instabiler, passt sich aber dem Trend des Besucherverhaltens an. Es werden trotz eines positiven Trends Instanzen gestoppt. Die Bearbeitungszeit liegt trotzdem im gewünschten Intervall um 300 ms.

Der untere Graph 4.14b zeigt bei den Instanzen ebenfalls Schwankungen, wobei auch hier der Trend der Besucherzahlen verfolgt wird. Es sind kaum Ausreißer der Antwortzeit zu erkennen.



(a) Szenario XA: Schwankende Besucherzahlen

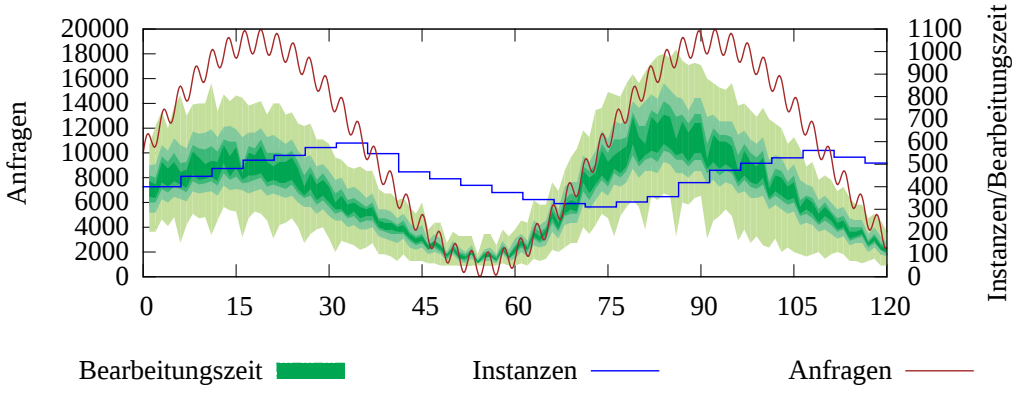


(b) Szenario XB: Steigende schwankende Besucherzahlen

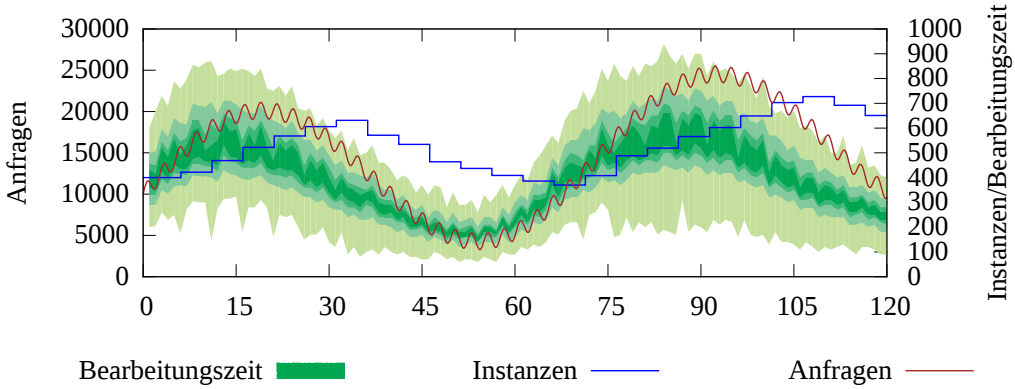
Abb. 4.15.: Dezentrale Arithmetische Methode ohne Aggregation

Die Abbildung 4.15 zeigt das Verhalten der dezentralen Steuerung ohne Aggregation. In der oberen Grafik 4.15a ist eine deutliche Abweichung der Antwortzeiten zwischen 60 und 90 Minuten zu erkennen. Durch die älteren Informationen der Peer-to-Peer-Informationsverteilung, reagiert das System zu langsam auf die schnell steigenden Anfragen. Erst mit dem Erreichen des Maximums bei 90 Minuten kann sich das System wieder stabilisieren.

Ein ähnliches Verhalten ist im unteren Graph 4.15b erkennbar. Die Abweichung der Antwortzeit ist geringer als im dritten Szenario. Der Grund dafür ist die hohe Anzahl an noch laufenden Instanzen. Um die Anfragen zu bearbeiten, müssen prozentual weniger Instanzen gestartet werden als im dritten Szenario, in dem die Anzahl der laufenden Instanzen auf 100 absinkt. Dadurch können



(a) Szenario XA: Schwankende Besucherzahlen



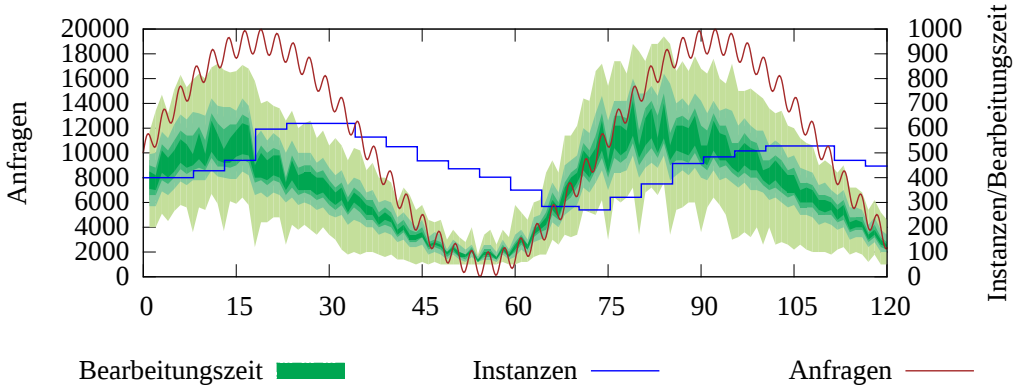
(b) Szenario XB: Steigende schwankende Besucherzahlen

Abb. 4.16.: Dezentrale Arithmetische Methode mit Aggregation

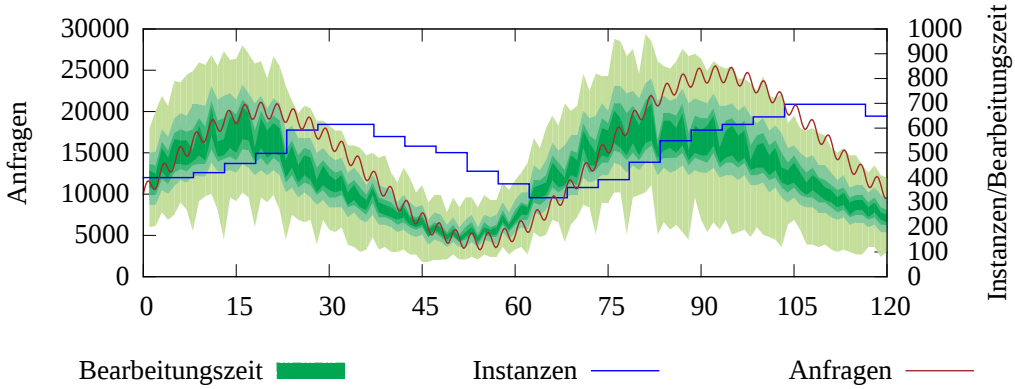
die Anfragen auf mehr Instanzen verteilt werden, das zu einem geringeren Anstieg der Antwortzeit führt.

Beide Graphen weisen keine unnötigen Schwankungen der Instanzen auf. Wenn eine Änderung der Besucherzahlen festgestellt wird, ist der Trend auch in der Instanzenanzahl erkennbar.

Abbildung 4.16 zeigt die Graphen für die arithmetische Methode mit Aggregation. Es ist deutlich in beiden Graphen ersichtlich, dass die Abweichung der Antwortzeit größer ist als in der Abbildung 4.15 für die nicht aggregierten Daten. Grund dafür ist die gestiegene Reaktionszeit. Das System braucht länger um auf eine Änderung der Besucherzahlen zu reagieren. Durch die langsame Reaktion, gibt es keine Schwankungen der Antwortzeit zwischen 50 und 60



(a) Szenario XA: Schwankende Besucherzahlen



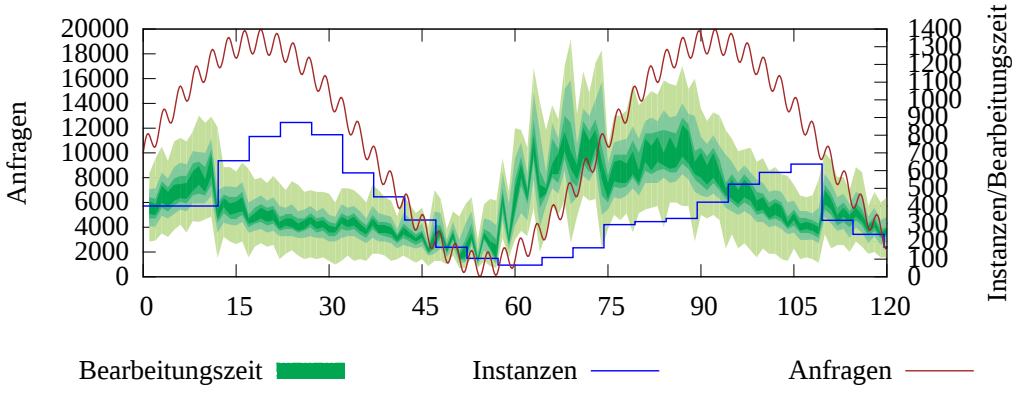
(b) Szenario XB: Steigende schwankende Besucherzahlen

Abb. 4.17.: Dezentrale Fuzzylogik-Steuerung ohne Aggregation

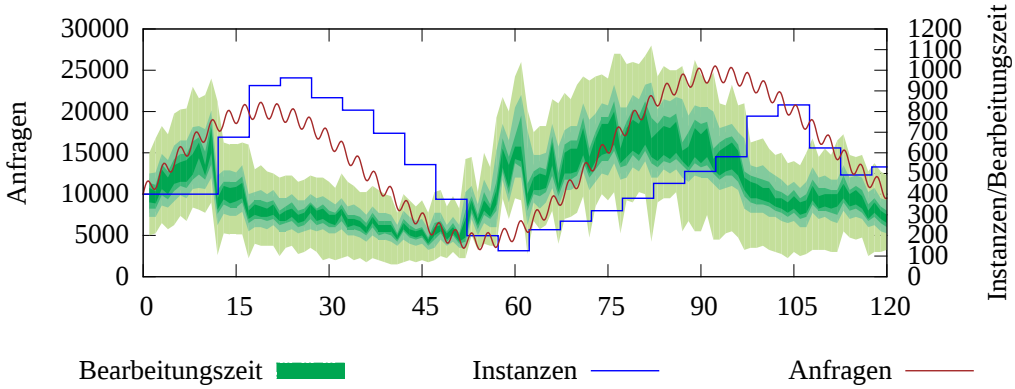
Minuten. Dafür sind die Antwortzeiten im Rest der Simulation außerhalb des Zielbereichs von 300 ms.

Die Abbildung 4.17 zeigt die Graphen für die dezentrale Fuzzylogik-Steuerung ohne Aggregation. In Graph 4.17a ist zu erkennen, dass die Antwortzeiten während der Zeitspanne zwischen 50 und 60 Minuten keine Ausreißer nach oben aufweisen. Allerdings wird die Bearbeitungszeit von 300 ms hier deutlich unterschritten. Die Fuzzylogik passt sich sehr langsam dem Besuchertrend an. Dadurch werden trotz geringer Besucherzahlen nicht alle unnötigen Instanzen gestoppt. Dies führt zu schnellen Bearbeitungszeiten und der Anpassung an die wieder steigenden Anfragen.

Ähnlich verhält es sich im vierten Szenario. Im Graphen 4.17b ist ersichtlich, dass die Anzahl der Instanzen nur langsam sinkt. Dies führt zu Ausreißern



(a) Szenario XA: Schwankende Besucherzahlen



(b) Szenario XB: Steigende schwankende Besucherzahlen

Abb. 4.18.: Dezentrale Fuzzylogik-Steuerung mit Aggregation

bei der Bearbeitungszeit nach Erreichen des lokalen Minimums. Dafür müssen während des Anstiegs nur wenige neue Instanzen gestartet werden. Da dieses Szenario einen steigenden Trend annimmt, erreichen die Besucherzahlen bei 90 Minuten ein globales Maximum. Die Steuerung benötigt auch hier lange, um sich dem neuen Maximum anzupassen. Erst mit dem Ende der Evaluation bei 120 Minuten liegt die Bearbeitungszeit zwischen 200 und 400 ms.

Die Abbildung 4.18 zeigt die Graphen für die dezentrale Fuzzylogik-Steuerung mit Aggregation. In Graph 4.18a sind wieder Ausreißer der Bearbeitungszeit beim Absinken der Besucherzahlen zu sehen. Grund dafür ist die schnellere Anpassung an das geänderte Besucherverhalten. Trotz ungenaueren Informationen durch die Aggregation reagiert das System schneller. Die Antwortzeit kann meistens auf dem gewünschten Wert von 300 ms gehalten werden.

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,65	1368	1178	898,99	0,0130
zentral Fuzzylogik	11,20	2632	2519	918,38	0,0122
dezentral Arithmetisch	19,50	1403	1017	939,83	0,0208
dezentral Arithmetisch aggr	21,42	845	340	920,01	0,0233
dezentral Fuzzylogik	20,60	877	430	922,97	0,0223
dezentral Fuzzylogik aggr	24,73	1443	1275	820,32	0,0301

Tab. 4.6.: Statistische Kenngrößen bei schwankenden Besucherzahlen (Szenario XA)

Graph 4.18b zeigt das Verhalten während des vierten Szenarios. Hier wird ebenfalls schnell auf Änderungen der Besucherzahlen reagiert. Meistens gelingt es die Antwortzeit auf dem gewünschten Wert zu halten.

Um die Ergebnisse besser vergleichen zu können, wurde auch für diese beiden Szenarien eine statistische Auswertung durchgeführt. Tabelle 4.6 zeigt die zusätzlichen Auswertungen zum dritten Szenario. Die geringsten Standardabweichung und die kleinsten Quotienten weisen die beiden zentralen Methoden auf, wobei die Fuzzylogik etwas besser ausfällt. Bei den dezentralen Methoden ist die arithmetische Steuerung ohne Aggregation mit einer Standardabweichung von 19,50 und einem Quotienten von 0,021 am besten. Danach folgt die dezentrale Fuzzylogik-Steuerung ohne Aggregation mit einer Standardabweichung von 20,60 und einem Quotienten von 0,023. Danach folgen die beiden dezentralen Steuerungen mit Aggregation. Die arithmetische Methode ohne Aggregation startet und stoppt mehr Instanzen als die Fuzzylogik-Steuerung ohne Aggregation. Die Fuzzylogik-Steuerung ohne Aggregation besitzt die geringste Standardabweichung mit wenigen Start-/Stopp-Aktionen und den zweitbesten Quotienten unter den dezentralen Steuerungen.

Tabelle 4.7 zeigt die zusätzlichen Auswertungen zum vierten Szenario XB. Wieder liefern die zentralen Steuerungen die geringsten Standardabweichungen.

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,20	1523	1066	1165,99	0,0096
zentral Fuzzylogik	10,44	2851	2434	1197,98	0,0087
dezentral Arithmetisch	15,94	1724	1127	1213,89	0,0131
dezentral Arithmetisch aggr	20,44	988	337	1071,74	0,0191
dezentral Fuzzylogik	22,01	992	344	1045,57	0,0211
dezentral Fuzzylogik aggr	23,57	1708	1176	1085,91	0,0217

Tab. 4.7.: Statistische Kenngrößen bei schwankenden steigenden Besucherzahlen (Szenario XB)

Die besten Werte unter den dezentralen Steuerungen besitzt die arithmetische Steuerung ohne Aggregation. Sie verfügt über eine Standardabweichung von 10,44 und einen Quotienten von 0,0087. Es werden nur 4% mehr Instanzen pro Stunde benötigt als bei den zentralen Steuerungen, obwohl wesentlich ungenauere Informationen vorhanden sind. Die dezentrale arithmetische Methode ohne Aggregation weist mit 20,44 eine deutlich größere Standardabweichung auf; verbraucht dafür weniger Instanzstunden. Die dezentralen Fuzzylogik-Steuerungen weisen in diesem Szenario die schlechtesten Werte auf. Bei den restlichen Szenarien, deren statische Auswertungen in den Tabellen im Anhang A.2 zu finden sind, erzielen die dezentralen Fuzzylogik-Steuerungen durchgehend bessere Werte als die dezentralen arithmetischen Steuerungen. Im hier vorgestellten Szenario XB besitzt die hochfrequente Sinusfunktion eine Amplitude von 1000. In den anderen Szenarien ist dieser Wert zwischen 1500 und 3500. Daraus lässt sich schließen, dass bei steigenden Schwankungen die Fuzzylogik-Steuerung besser geeignet ist.

Abbildung 4.19 zeigt die Standardabweichungen der Steuerungsalgorithmen bei verschiedenen Amplituden. Es ist deutlich erkennbar, dass bei größeren Amplituden die dezentrale Fuzzylogik-Steuerung mit Aggregation die besseren Werte liefert. Bei steigender Amplitude verschlechtern sich auch die Werte

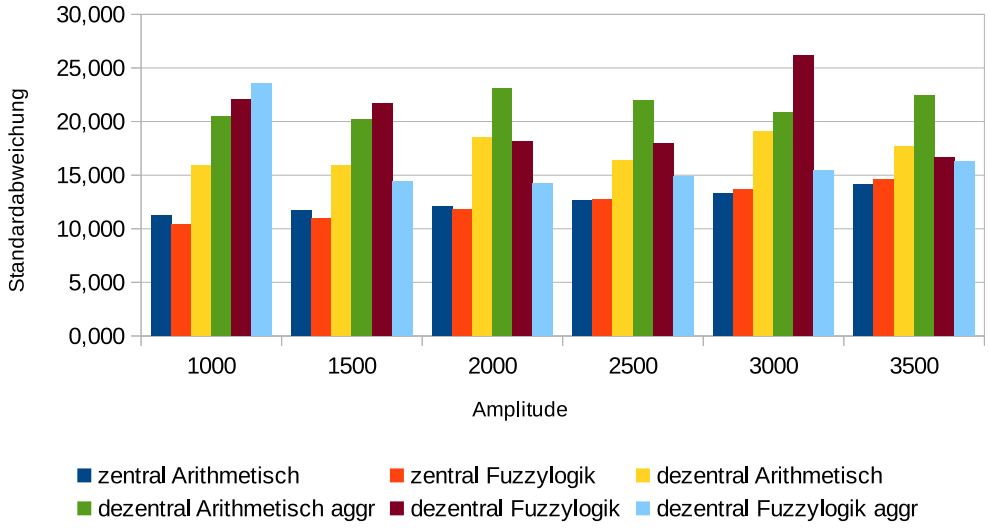


Abb. 4.19.: Standardabweichung bei verschiedenen Amplituden (Parameter A)

der zentralen Steuerungen. Bei der größten Amplitude von 3500 erreicht die dezentrale Fuzzylogik-Steuerung mit Aggregation (16,26) nur einen um 15% schlechteren Wert als die zentrale arithmetische Steuerung (14,01).

Abbildung 4.20 zeigt die Zeit, die benötigt wird, um auf eine sprunghafte Änderung der Besucherzahlen zu reagieren. An der x-Achse sind die sechs Fälle bei einer Änderung der Besucherzahlen um die Faktoren $\frac{1}{6}$, $\frac{1}{4}$, $\frac{1}{2}$, 2, 4 und 6 dargestellt. Auf der y-Achse wird die Zeit angetragen, die das System zur Stabilisierung benötigt. Es werden wie bereits vorher zwei Stunden simuliert und die Änderung bei 15 Minuten durchgeführt. Eine Stabilisierungszeit von 105 Minuten bedeutet, dass das System während der Simulationszeit keinen stabilen Zustand erreicht hat.

Es ist erkennbar, dass die beiden zentralen Steuerungen sehr schnell auf jede der Änderungen reagieren. Bei einer Verringerung der Besucherzahlen erzielen die dezentralen Methoden mit Aggregation sehr schlechte Reaktionszeiten. Durch das ständige Wiederverwenden der alten Werte, sinkt deren Einfluss. Da gleichzeitig aber neue große Lastinformationen aufgezeichnet werden, ergibt die Mittelwertbildung noch einen hohen Wert. Dadurch werden die langen Antwortzeiten maskiert und das System reagiert verzögert.

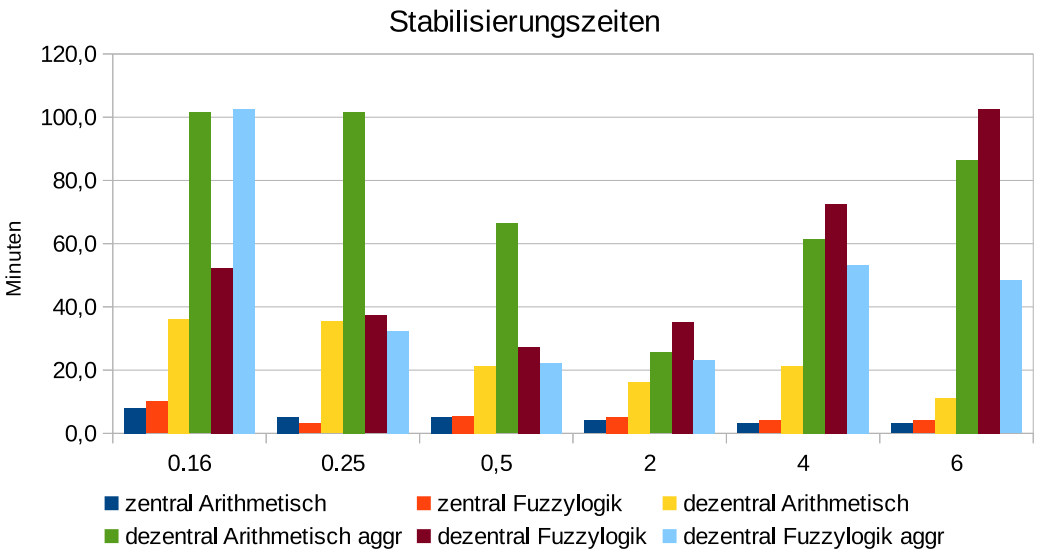


Abb. 4.20.: Stabilisierungszeiten

Da die Fuzzylogik nur für eine Verdoppelung bzw. Halbierung der Besucherzahlen ausgelegt ist, verschlechtert sich die Stabilisierungszeit bei zu großen Veränderungen.

Abbildung 4.21 zeigt die Standardabweichung der erzielten Antwortzeiten zur gewünschten Antwortzeit von 300 ms. Es wurde der Mittelwert über 7 Simulationen (XA - XG) gebildet. Die geringste Standardabweichung von 12,2 erzielt die zentrale Fuzzylogik-Steuerung. Den zweitbesten Wert 12,4 erreicht die zentrale arithmetische Steuerung. Damit liegen die beiden zentralen Steuerungen vor den dezentralen und besitzen nur einen Unterschied von 1,5 Prozent. Eine Standardabweichung von 17,6 besitzen zwei dezentrale Steuerungen, die arithmetische ohne Aggregation und die Fuzzylogik-Steuerung mit Aggregation. Die schlechtesten Werte weisen die dezentrale Fuzzylogik-Steuerung ohne Aggregation (20,5) und die arithmetische Steuerung mit Aggregation (21,5) auf. Damit erzielt die beste dezentrale Steuerung eine um 44 Prozent höhere Standardabweichung als die zentrale Fuzzylogik-Steuerung.

Abbildung 4.22 zeigt die Anzahl der gestarteten und gestoppten Instanzen für die sechs Steuerungsmethoden als Mittelwert über 7 Simulationen (XA - XG) mit verschiedenen Besucherverhalten. Der grüne Balken entspricht den genutzten Instanzen pro Stunde. Dafür wurde über die Anzahl der laufenden

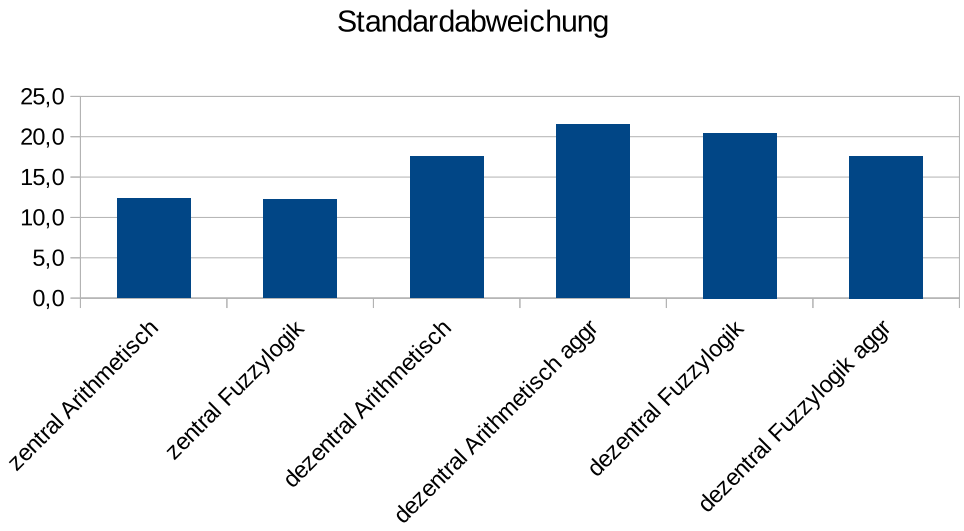


Abb. 4.21.: Standardabweichung der erzielten Antwortzeiten zur gewünschten Antwortzeit von 300ms

Instanzen ein Integral gebildet. Die zentrale Fuzzylogik erreicht die geringe Standardabweichung durch viele Eingriffe in das System. Dies führt zu vielen Start- und Stopp-Aktionen. Die verbrauchten Maschinenstunden sind bei allen Steuerungsmethoden sehr ähnlich. Sie schwanken nur um 2 bis 12 Prozent. Die wenigsten Start- und Stopp-Aktionen werden von der dezentralen arithmetischen Methode mit Aggregation und der dezentralen Fuzzylogik-Steuerung ohne Aggregation erreicht.

Aus den vorgestellten Graphen ergibt sich, dass die dezentrale Fuzzylogik-Steuerung ohne Aggregation mit sehr wenigen Start- und Stopp-Aktionen das System mit einer Standardabweichung von 20,5 ms steuern kann. In dem vorgestellten Szenario entspricht das einer Abweichung von 6,8% vom gewünschten Zielwert. Die dezentrale Fuzzylogik-Steuerung mit Aggregation benötigt mehr Start-/Stopp-Operationen, erreicht dafür aber eine Standartvarianz von 17,6 ms. Dies entspricht einer Abweichung von 5,8% vom Zielwert.

Tabelle B.10 enthält die in den Graphen verwendeten Zahlen.

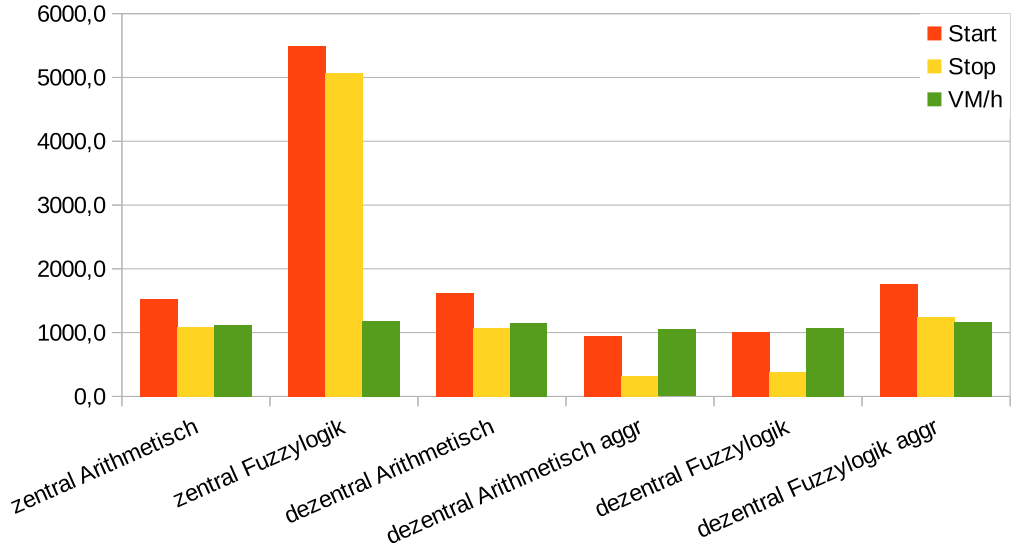


Abb. 4.22.: Instanzennutzung

4.5. Fazit

In diesem Kapitel wurde ein Cloud-Computing-Szenario vorgestellt und mit Hilfe zweier Methoden gesteuert. Die arithmetische Methode ist ein einfacher Dreisatz, bei dem die Antwortzeit nach dem Alter der Messung gewichtet wird. Hier wird die Anzahl der Instanzen einfach um das Verhältnis vom Messwert zum Sollwert vergrößert oder verkleinert. Als zweite Methode wurde Fuzzylogik angewandt. Beide Methoden benutzen nur die gemessenen Antwortzeiten der Webserver und das Alter der Messung um Entscheidungen zu treffen. Es wird keine Vorhersage getroffen oder der Trend ermittelt. Zusätzlich wurden verschiedene Besucherverhalten ausgewählt um beide Steuerungsmethoden zu vergleichen. Für jede Methode wurde eine zentrale, eine dezentrale und eine dezentrale Steuerung mit aggregierten Informationen untersucht. Bei der zentralen Steuerung sind die Informationen immer aktuell. Die dezentrale Methode ohne Aggregation erhält die Lastdaten aller Knoten, wobei das Alter der Nachrichten stark schwankt. Die Steuerung kann das Alter benutzen um die Antwortzeiten zu gewichten. Bei der dezentralen Steuerung mit aggregierten Daten, wird nur eine Information gesendet. Jeder Knoten erhält die Einschätzungen seiner Nachbarn, kann aber nicht nach Alter der Informationen

unterscheiden. Allerdings wird durch die Mittelwertbildung beim Weiterleiten automatisch die Gewichtung älterer Informationen gesenkt.

Die Evaluierung hat ergeben, dass der zentrale Ansatz die besten Ergebnisse liefert, wobei die Fuzzylogik besser ist als die arithmetische Steuerung. Auch in den weiteren Szenarien hat die Steuerung mit Fuzzylogik bessere Werte erzielt als die arithmetische. Die Ergebnisse der Simulationen mit oder ohne Aggregation der Daten zeigen kein eindeutiges Verhalten. Bei der arithmetischen Methode sind die Werte mit aggregierten Daten schlechter als ohne. Bei größeren Schwankungen der Besucherzahlen verbessern sich die Werte der Fuzzylogik.

Die verteilte Steuerung des Cloud-Computing-Systems konnte mit der Fuzzylogik-Steuerung realisiert werden. Dabei wurden Abweichungen der Antwortzeit von 5,8% (mit Aggregation) und 6,8% (ohne Aggregation) festgestellt. Bei der zentralen Steuerung entspricht die Abweichung 4,1% vom Zielwert.

Die dezentrale Fuzzylogik-Steuerung ohne Aggregation benötigt 4,6% weniger Maschinenstunden als die zentrale arithmetische Steuerung. Die dezentrale Fuzzylogik-Steuerung mit Aggregation benötigt 3,5% mehr.

5. Zusammenfassung

Cloud-Computing wird immer wichtiger bei großen und mittelständischen Betrieben. Um sich nicht von einem Anbieter abhängig zu machen, sollten Cloud-Computing-Lösungen von verschiedenen Anbietern übernommen werden. Da die Steuerungs- und Überwachungsdienste einzelner Cloud-Computing-Anbieter nicht miteinander kommunizieren können, muss die Steuerung einer solchen auf mehrere Anbieter verteilten Cloud-Computing-Infrastruktur selbst implementiert werden. Um einen Single-Point-of-Failure zu verhindern, wird in dieser Arbeit eine dezentrale Steuerung untersucht. Dabei werden die beiden Bereiche der Informationsverteilung und Steuerung genauer betrachtet.

Die Informationsverteilung ist verantwortlich für das Verteilen der Lastinformationen an die Steuerungseinheit. Bei einer zentralen Steuerung werden die Lastdaten der einzelnen virtuellen Maschinen per SNMP oder anderen Protokollen von der Steuerung gesammelt. Da bei einem dezentralen Management alle virtuellen Maschinen an der Steuerung beteiligt sind, müssen auch die Lastinformationen an alle Maschinen verteilt werden. Die Anzahl der aktiven Maschinen kann sich ständig ändern, daher sind den einzelnen Maschinen nicht alle anderen Maschinen bekannt. Um trotzdem die Lastinformationen an alle Teilnehmer verteilen zu können, wurde im Kapitel 3 Informationsverteilungsalgorithmen, die auf Peer-to-Peer-Algorithmen basieren, vorgestellt und evaluiert. Dabei wird das Routingverfahren der Peer-to-Peernetze CAN, Chord und Pastry untersucht. Jeder VM ist nur eine kleine Anzahl an Nachbarn bekannt. Diese leiten die Lastdaten dann weiter. Mit diesen Algorithmen ist eine effiziente Informationsverteilung möglich. Es werden keine zusätzlichen Nachrichten, wie bei Flooding, versendet. Bei einer zentralen Überwachung muss für eine Analyse 1 Nachricht bei n Instanzen abgefragt werden. Das ergibt n Nachrichten, die pro Überwachungsintervall gesammelt werden müssen. Bei der kompletten Informationsverteilung benötigt jede der n Instanzen Daten der $n-1$ anderen Instanzen. Daher benötigen die P2P-Algorithmen $n(n-1)$ Nachrichten pro Überwachungsintervall. Durch Aggregation der Lastdaten kann die

Anzahl der Nachrichten gesenkt werden, wobei auch der Informationsgehalt abnimmt. Da die Lastdaten mehrere Knoten passieren, bis sie ihr Ziel erreichen, sind die Informationen auch älter als im zentralen Fall. Der von Besuchern erzeugte Traffic ist bei den untersuchten Cloud-Computing-Systemen sehr hoch, daher liegt der durch die dezentrale Überwachung erzeugte Overhead bei unter 1% der benutzten Netzwerkbandbreite.

Die von den Verteilungsalgorithmen gelieferten ungenauen Informationen müssen dann im nächsten Schritt in der Steuerung benutzt werden um das komplette System zu verwalten. Im Kapitel 4 wird das Beispiel einer großen Cloud-Computing-Anwendung vorgestellt. Es wurden mehrere Steuerungsmethoden mit dieser Anwendung untersucht. Als Referenzimplementierung wurde die gemessene Antwortzeit mit der gewünschten Antwortzeit verglichen und dann die Anzahl der VMs um diesen Faktor vergrößert bzw. verkleinert. Zusätzlich wurde eine auf Fuzzy-Logik basierende Steuerung vorgestellt. Beide Steuerungen wurden als zentrale und als dezentrale Variante implementiert. Zusätzlich wurde der Einfluss der Aggregation während der Informationsverteilung untersucht. Bei den dezentralen Steuerungen wurde der Chord-basierte Verteilungsalgorithmus aus Kapitel 3 verwendet. Die zentralen Steuerungsalgorithmen konnten auf Grund des globalen Wissens das System mit kürzeren Reaktionszeiten stabiler halten. Die vorgestellte Fuzzylogik-Steuerung kann das System dezentral mit einer Abweichungen der Antwortzeit von 5,8% erfolgreich steuern. Die Anzahl der laufenden Instanzen erhöht sich um 3,5 %.

Es wurde gezeigt, dass die vorgestellten Informationsverteilungs- und Steuerungsalgorithmen für große selbst-organisierende Systeme geeignet sind. Durch die dezentrale Steuerung ist keine dedizierte Steuerungseinheit, die einen Single-Pointe-of-Failure darstellt, mehr notwendig. In den Simulationen konnten Systeme mit bis zu 1000 Rechnern erfolgreich gesteuert werden.

A. Ergebnisse Cloud-Computing-Evaluation

A.1. Konstant mit einem Sprung

A.1.1. Eingabe A: Verdoppelung der Besucher bei 15 Minuten

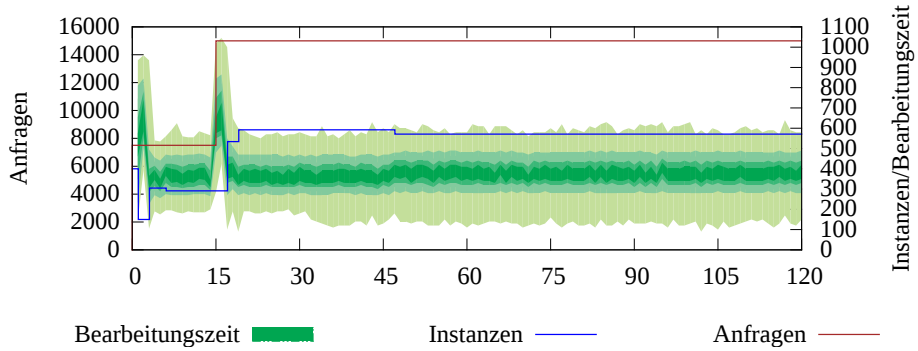


Abb. A.1.: Zentrale Arithmetische Methode

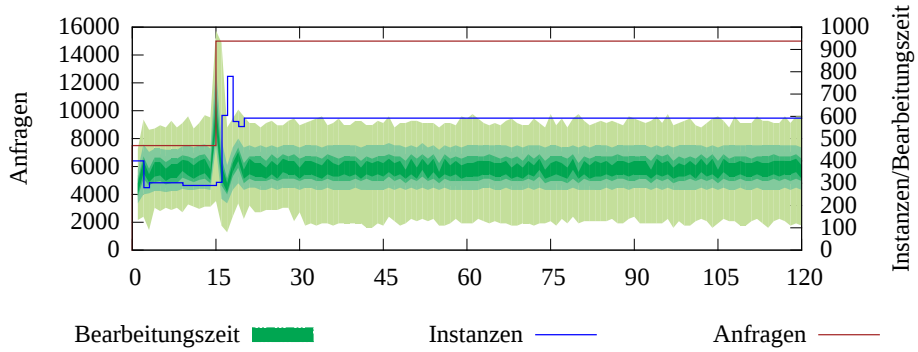


Abb. A.2.: Zentrale Fuzzylogik

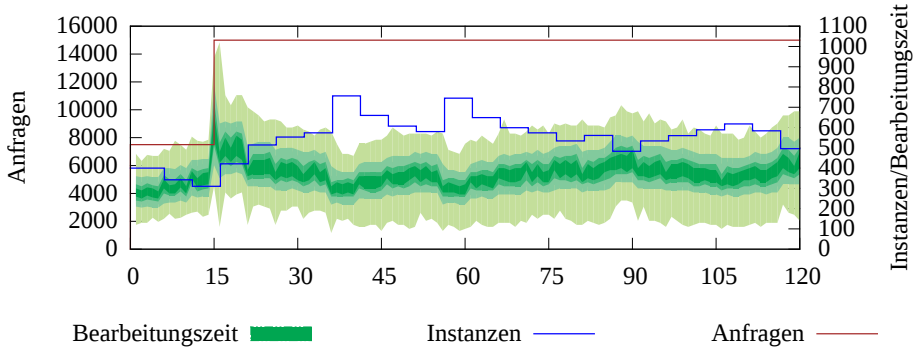


Abb. A.3.: Dezentrale Arithmetische Methode ohne Aggregation

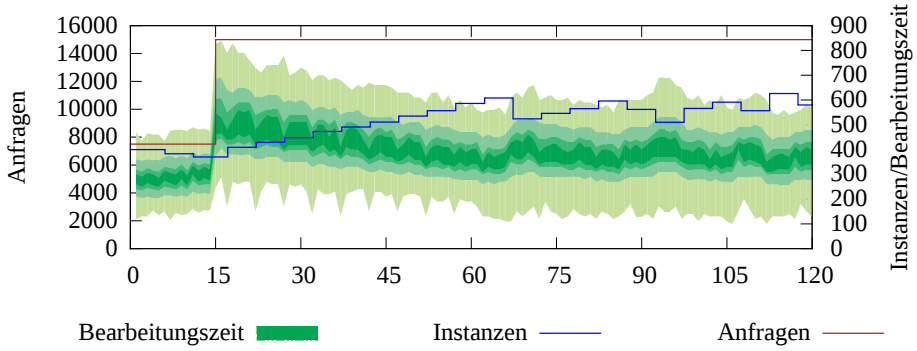


Abb. A.4.: Dezentrale Arithmetische Methode mit Aggregation

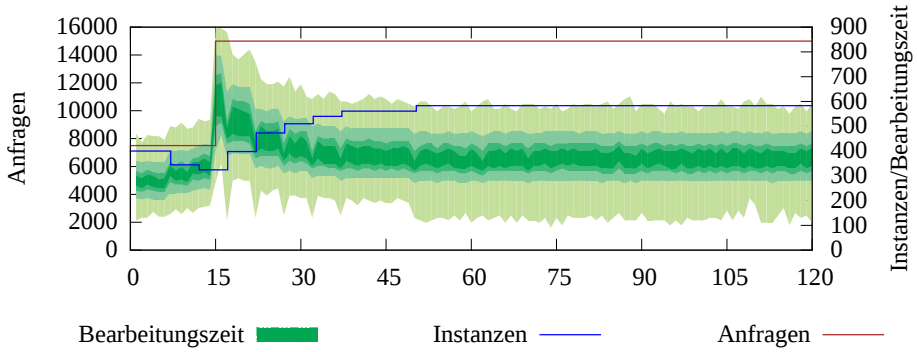


Abb. A.5.: Dezentrale Fuzzylogik ohne Aggregation

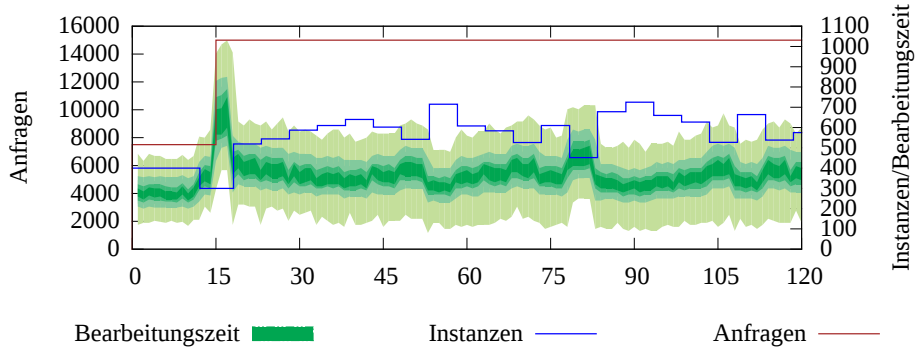


Abb. A.6.: Dezentrale Fuzzylogik mit Aggregation

A.1.2. Eingabe B: Halbierung der Besucher bei 15 Minuten

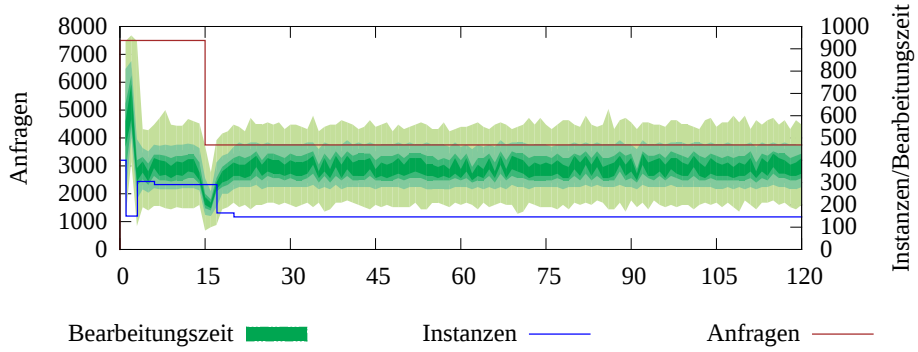


Abb. A.7.: Zentrale Arithmetische Methode

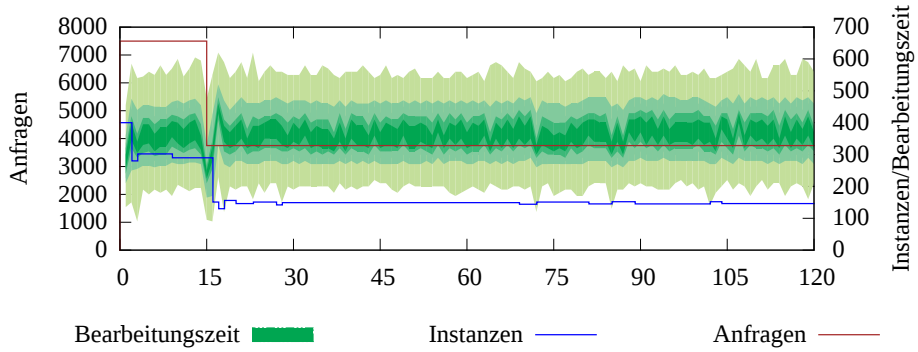


Abb. A.8.: Zentrale Fuzzylogik

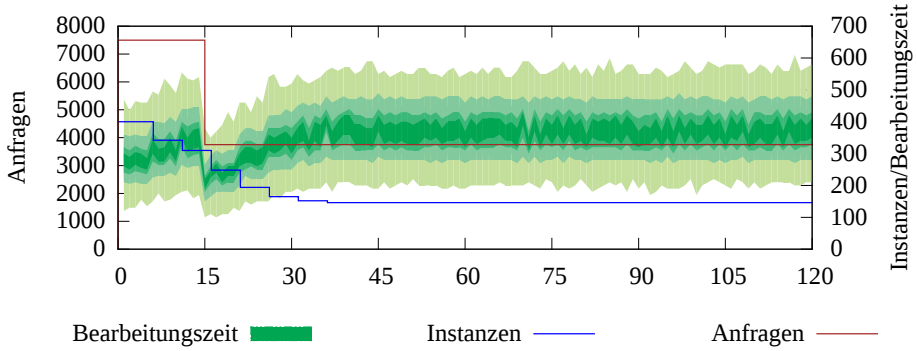


Abb. A.9.: Dezentrale Arithmetische Methode ohne Aggregation

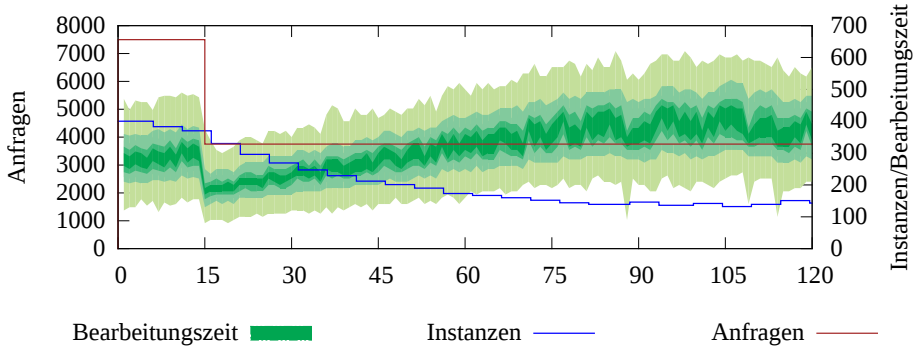


Abb. A.10.: Dezentrale Arithmetische Methode mit Aggregation

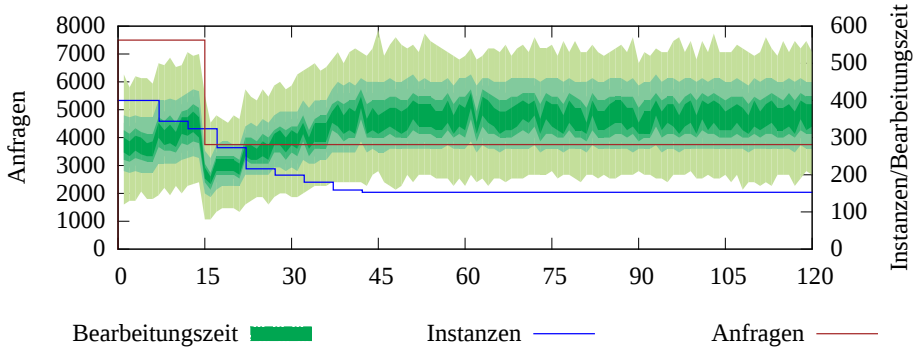


Abb. A.11.: Dezentrale Fuzzylogik ohne Aggregation

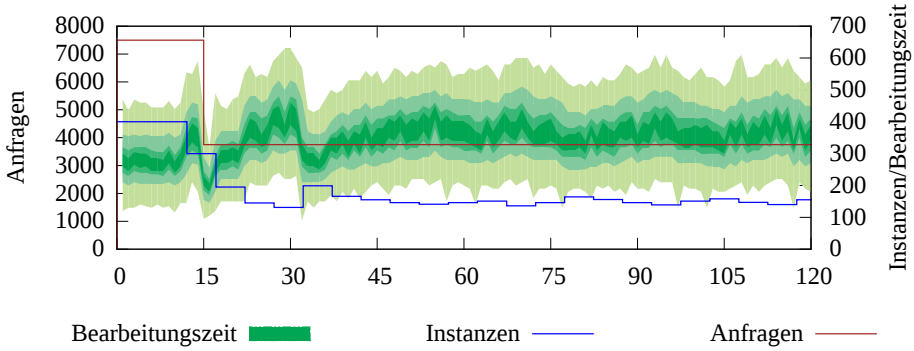


Abb. A.12.: Dezentrale Fuzzylogik mit Aggregation

A.2. Schwankende Besucherzahlen

A.2.1. Eingabe XA: $9000\sin(\frac{t}{700}) + 1000\sin(\frac{t}{25}) + 10000$

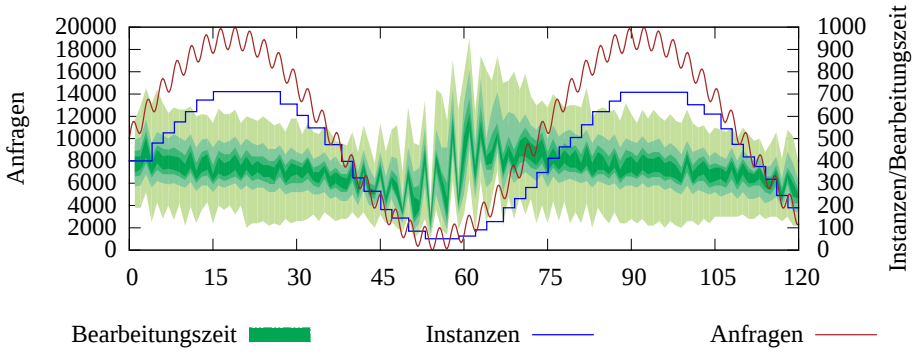


Abb. A.13.: Zentrale Arithmetische Methode

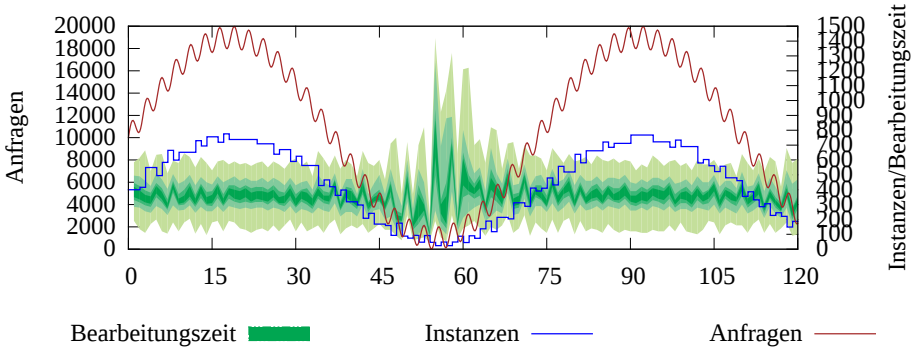


Abb. A.14.: Zentrale Fuzzylogik

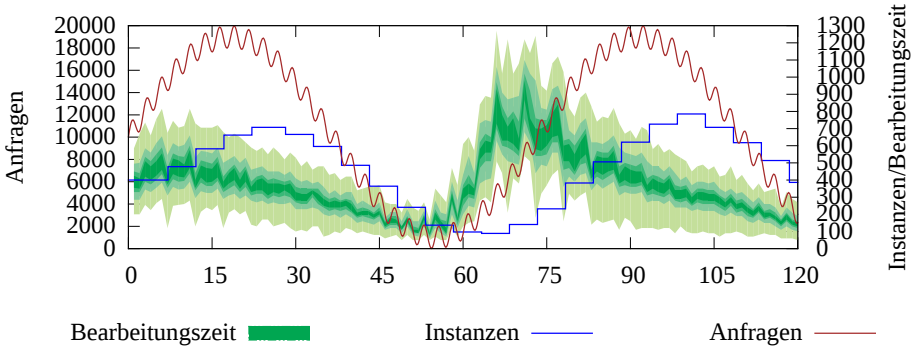


Abb. A.15.: Dezentrale Arithmetische Methode ohne Aggregation

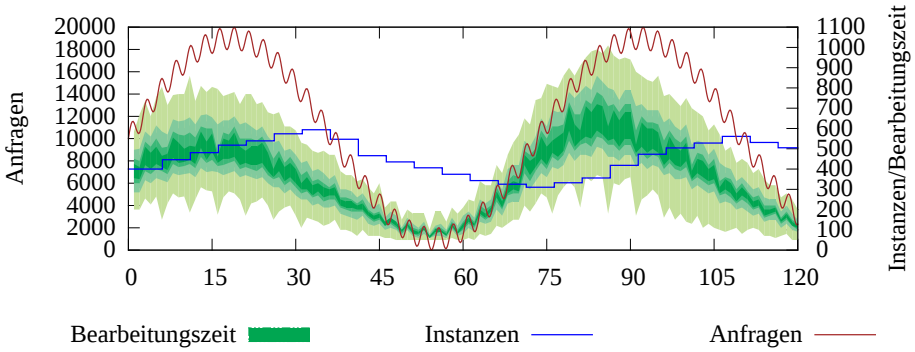


Abb. A.16.: Dezentrale Arithmetische Methode mit Aggregation

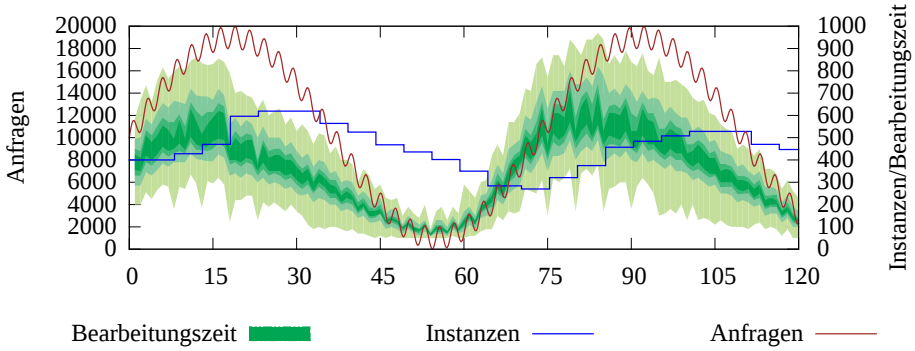


Abb. A.17.: Dezentrale Fuzzylogik ohne Aggregation

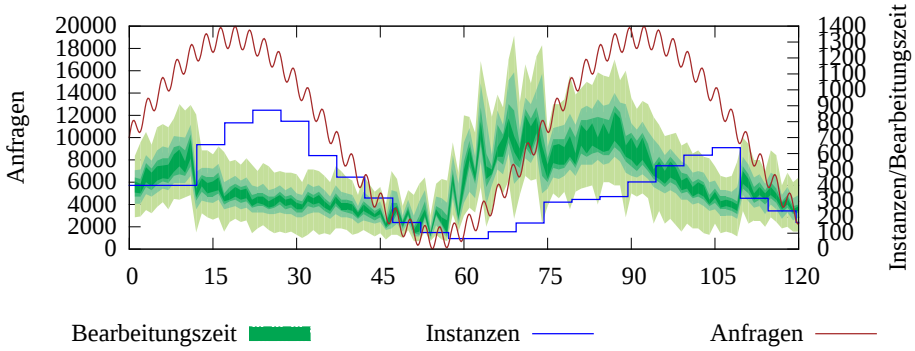


Abb. A.18.: Dezentrale Fuzzylogik mit Aggregation

A.2.2. Eingabe XB: $9000\sin(\frac{t}{700}) + 1000\sin(\frac{t}{25}) + 10000 + t$

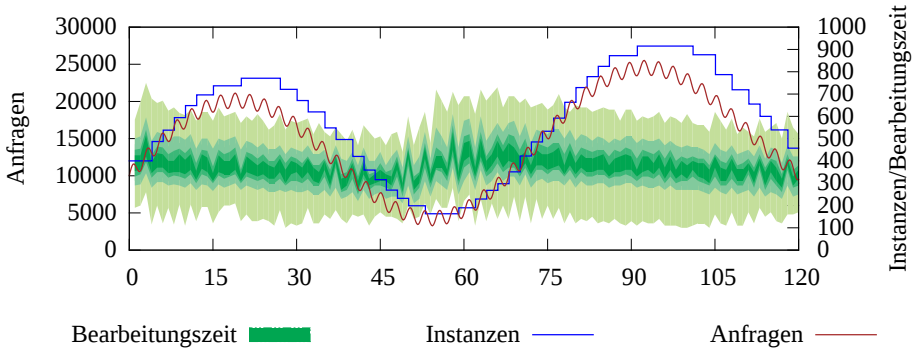


Abb. A.19.: Zentrale Arithmetische Methode

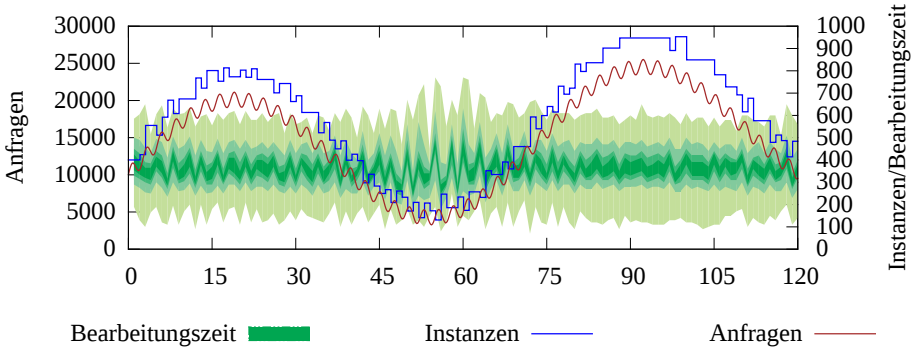


Abb. A.20.: Zentrale Fuzzylogik

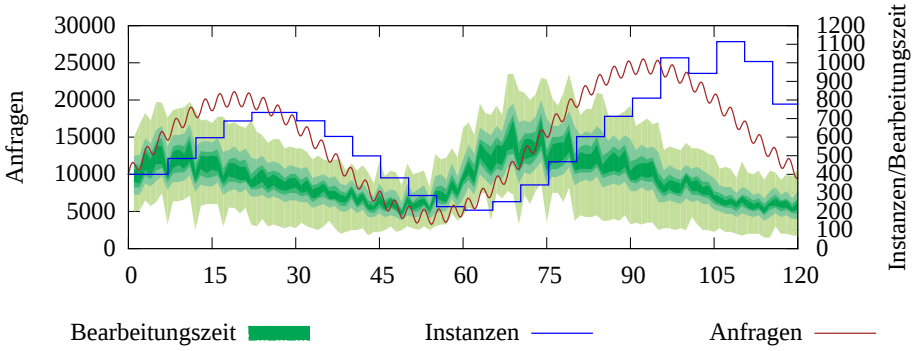


Abb. A.21.: Dezentrale Arithmetische Methode ohne Aggregation

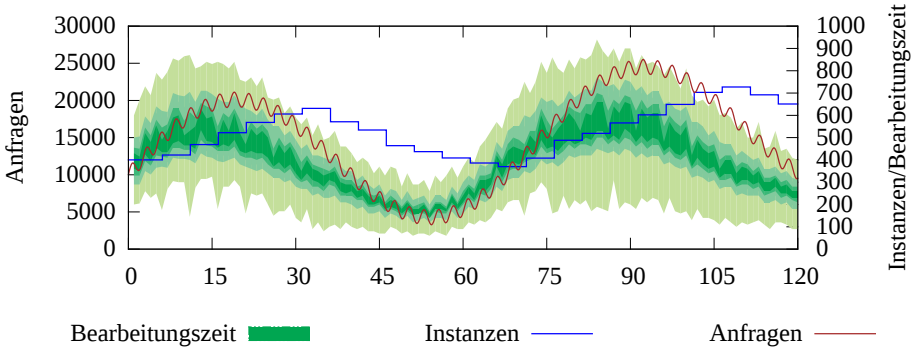


Abb. A.22.: Dezentrale Arithmetische Methode mit Aggregation

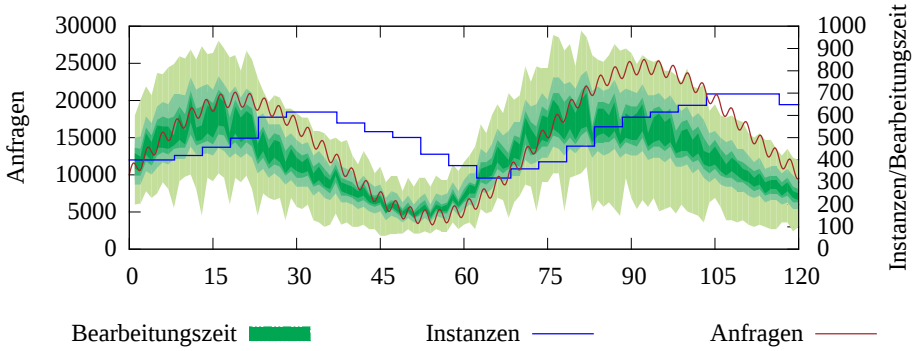


Abb. A.23.: Dezentrale Fuzzylogik ohne Aggregation

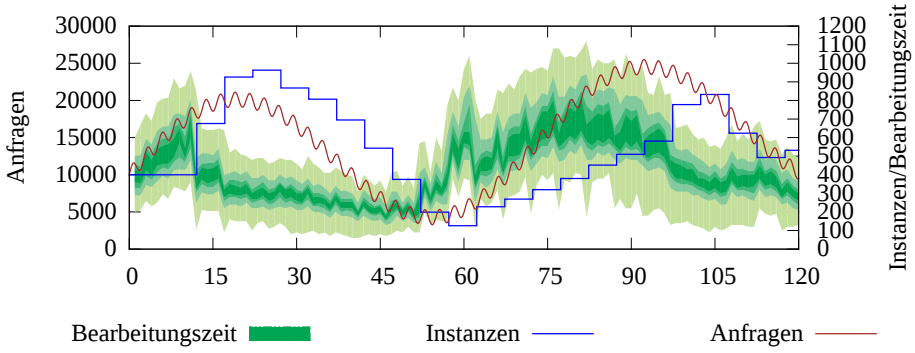


Abb. A.24.: Dezentrale Fuzzylogik mit Aggregation

A.2.3. Eingabe XC: $9000\sin(\frac{t}{700}) + 3500\sin(\frac{t}{25}) + 10000 + t$

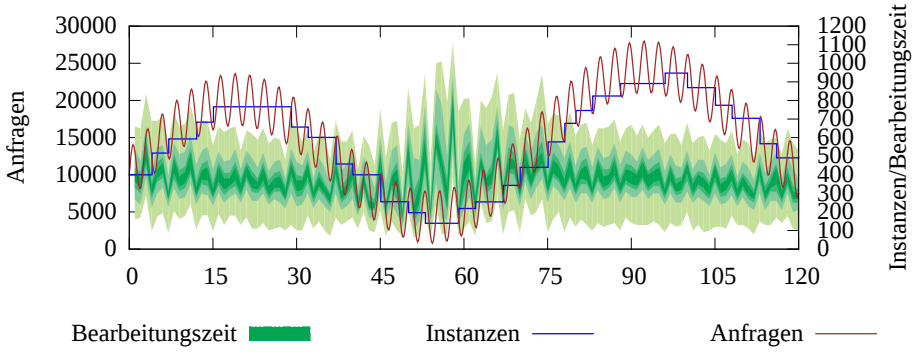


Abb. A.25.: Zentrale Arithmetische Methode

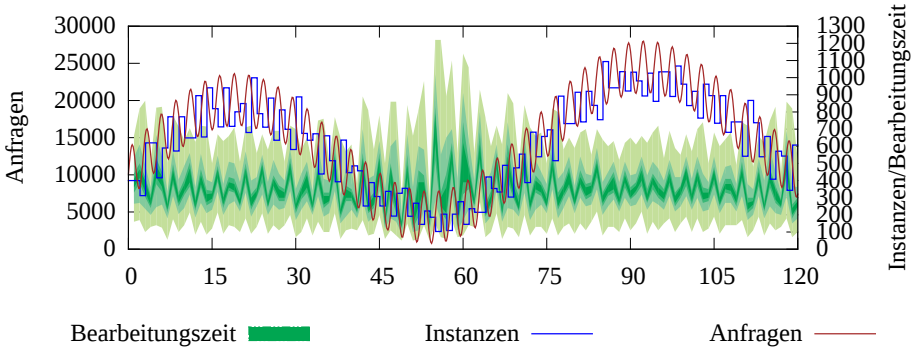


Abb. A.26.: Zentrale Fuzzylogik

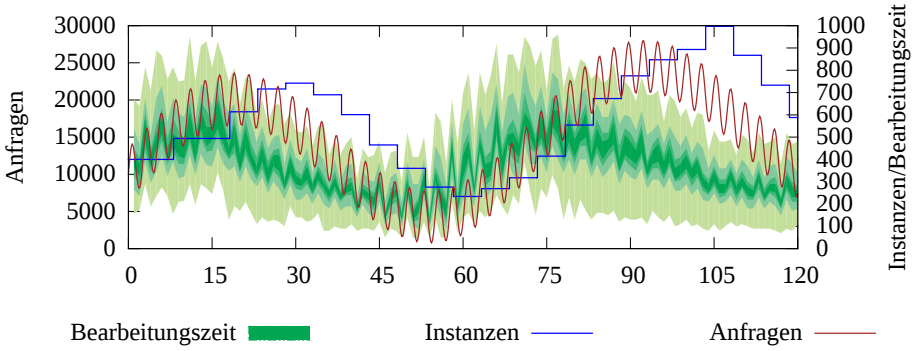


Abb. A.27.: Dezentrale Arithmetische Methode ohne Aggregation

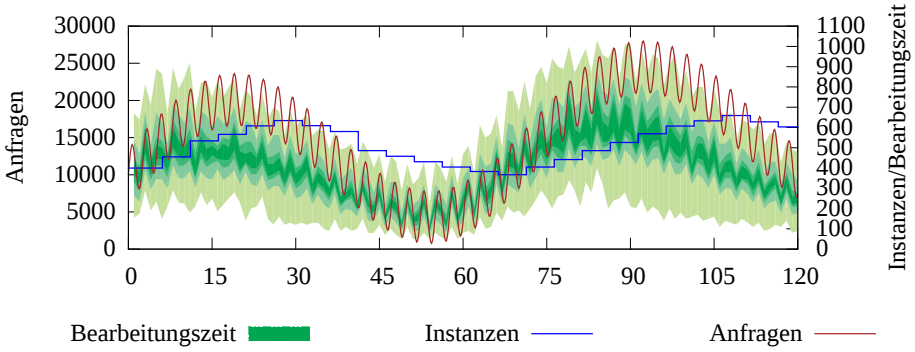


Abb. A.28.: Dezentrale Arithmetische Methode mit Aggregation

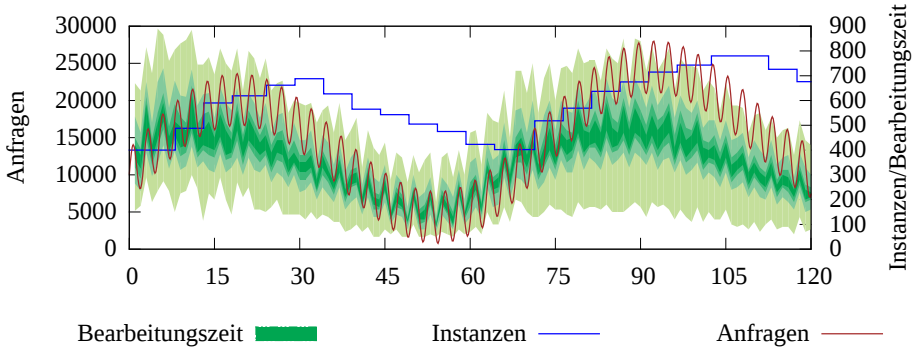


Abb. A.29.: Dezentrale Fuzzylogik ohne Aggregation

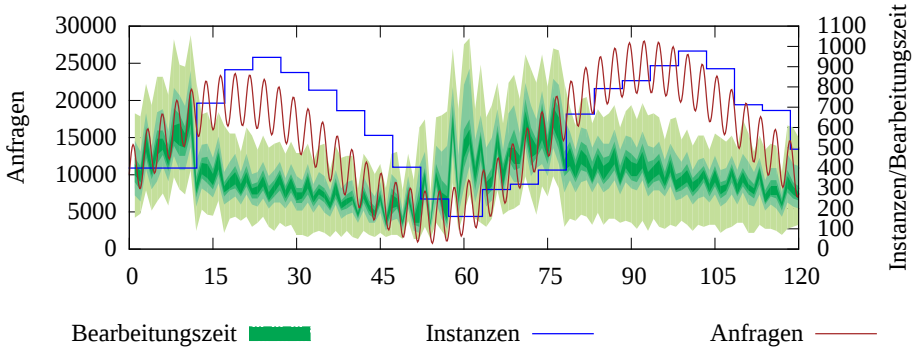


Abb. A.30.: Dezentrale Fuzzylogik mit Aggregation

A.2.4. Eingabe XD: $9000\sin(\frac{t}{700}) + 2000\sin(\frac{t}{25}) + 10000 + t$

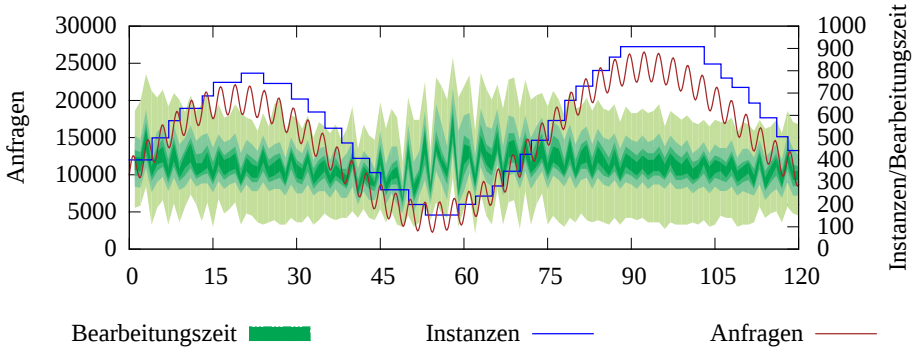


Abb. A.31.: Zentrale Arithmetische Methode

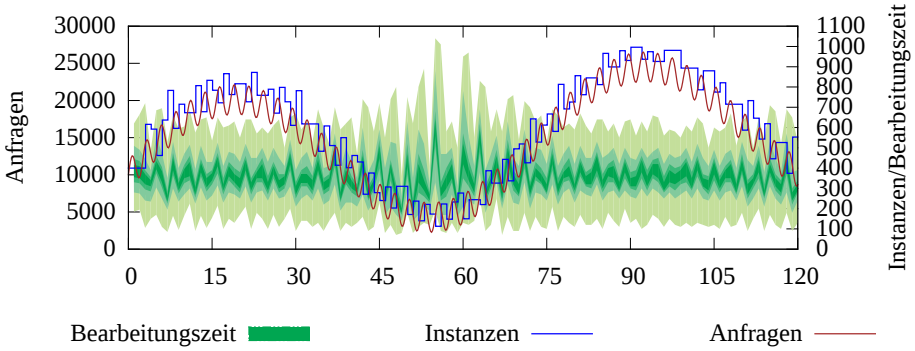


Abb. A.32.: Zentrale Fuzzylogik

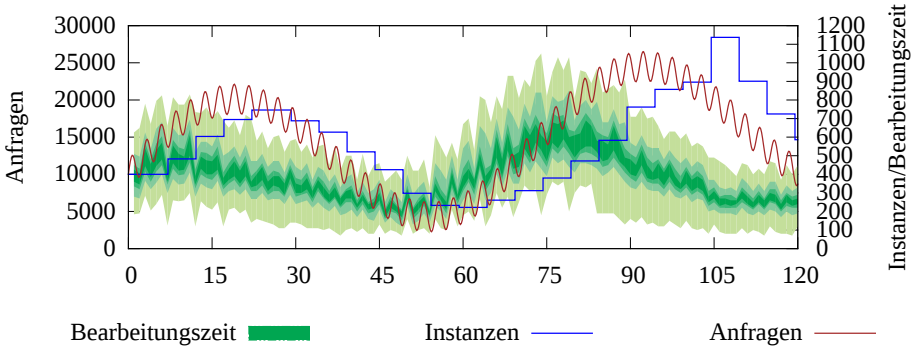


Abb. A.33.: Dezentrale Arithmetische Methode ohne Aggregation

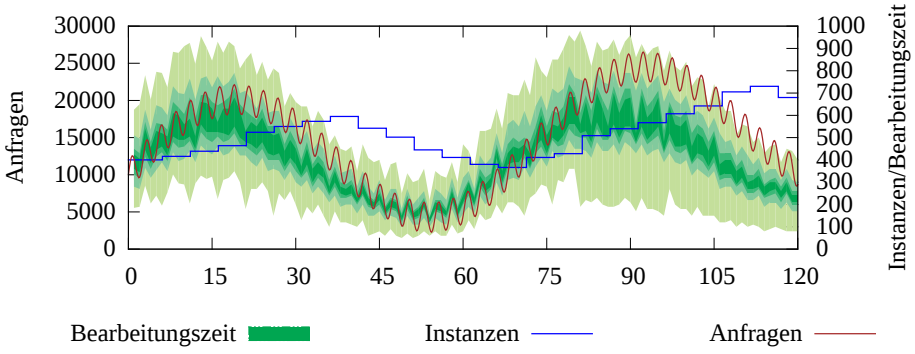


Abb. A.34.: Dezentrale Arithmetische Methode mit Aggregation

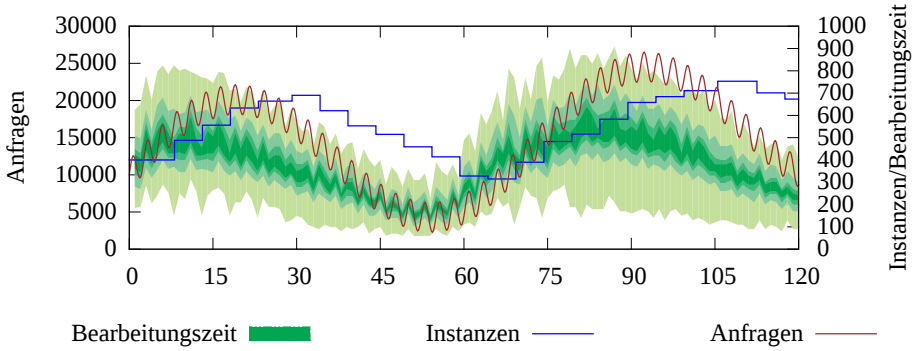


Abb. A.35.: Dezentrale Fuzzylogik ohne Aggregation

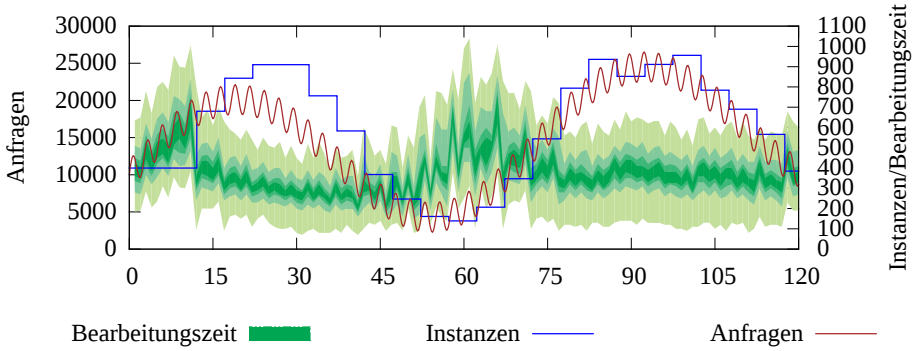


Abb. A.36.: Dezentrale Fuzzylogik mit Aggregation

A.2.5. Eingabe XE: $9000\sin(\frac{t}{700}) + 2500\sin(\frac{t}{25}) + 10000 + t$

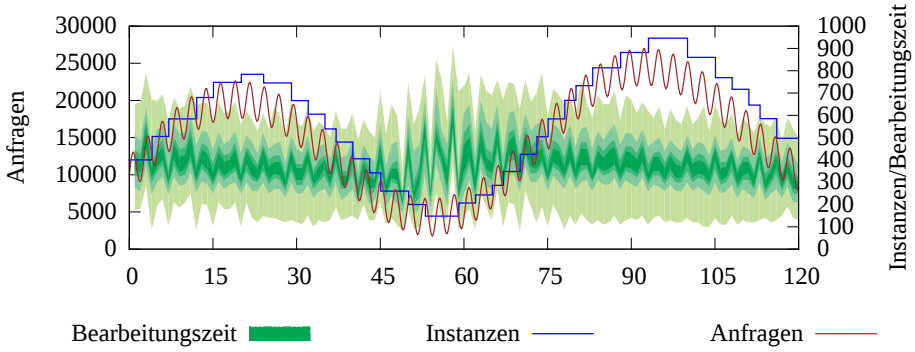


Abb. A.37.: Zentrale Arithmetische Methode

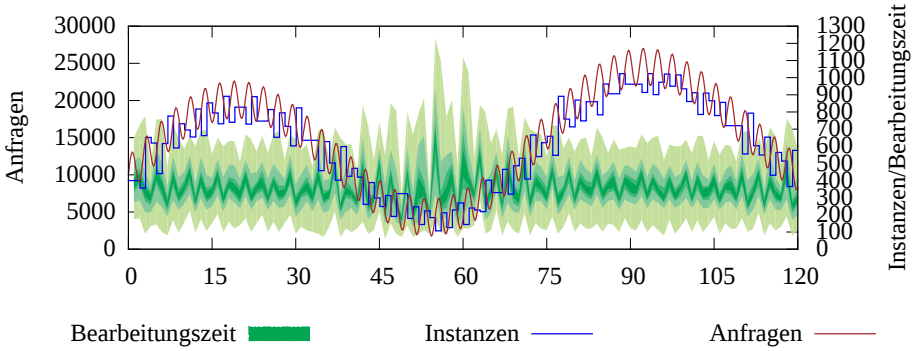


Abb. A.38.: Zentrale Fuzzylogik

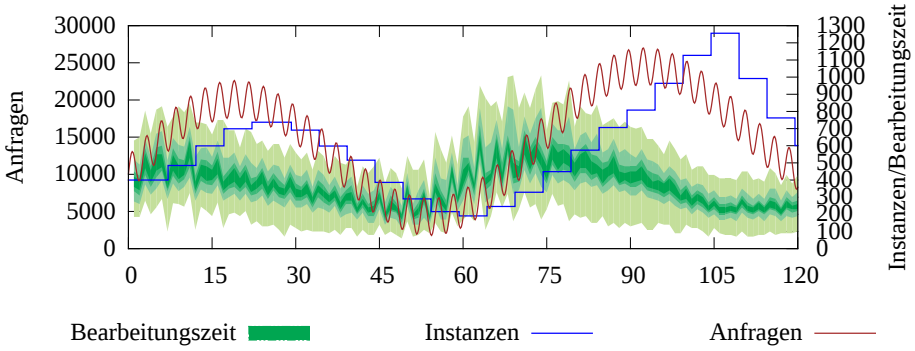


Abb. A.39.: Dezentrale Arithmetische Methode ohne Aggregation

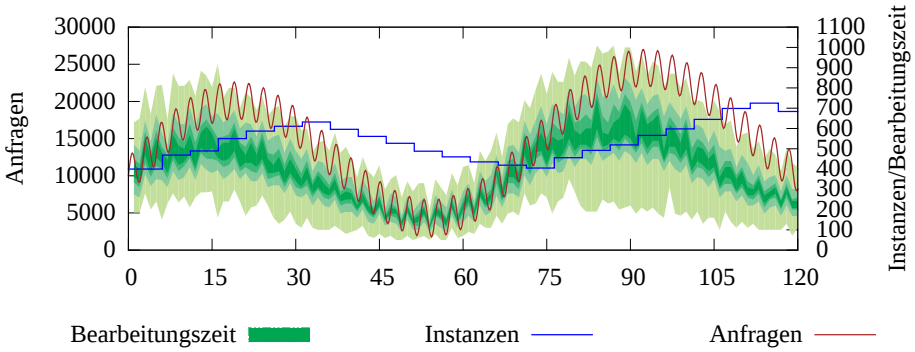


Abb. A.40.: Dezentrale Arithmetische Methode mit Aggregation

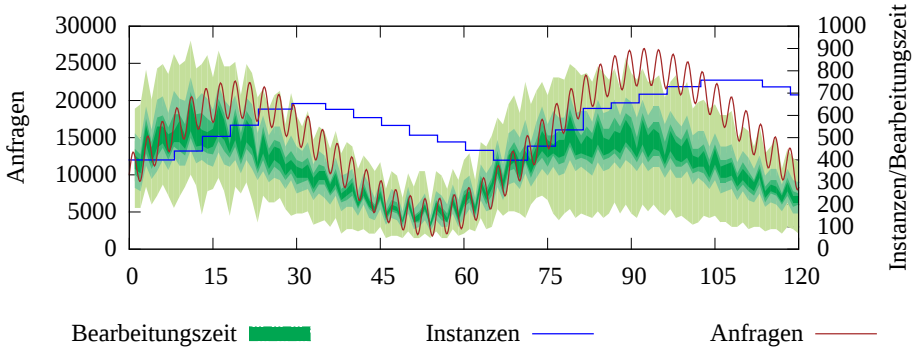


Abb. A.41.: Dezentrale Fuzzylogik ohne Aggregation

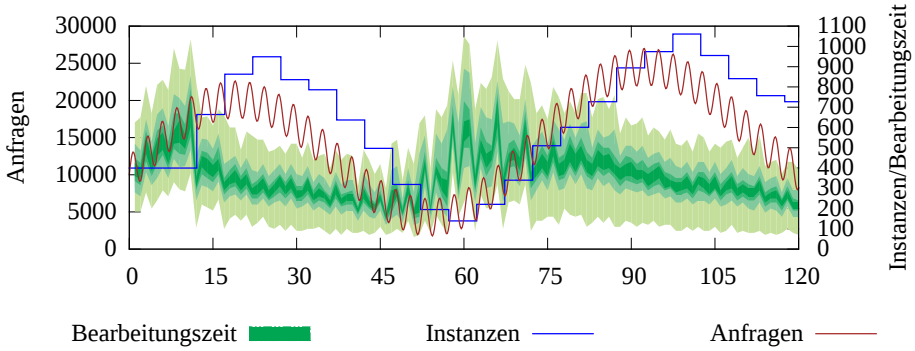


Abb. A.42.: Dezentrale Fuzzylogik mit Aggregation

A.2.6. Eingabe XF: $9000\sin(\frac{t}{700}) + 3000\sin(\frac{t}{25}) + 10000 + t$

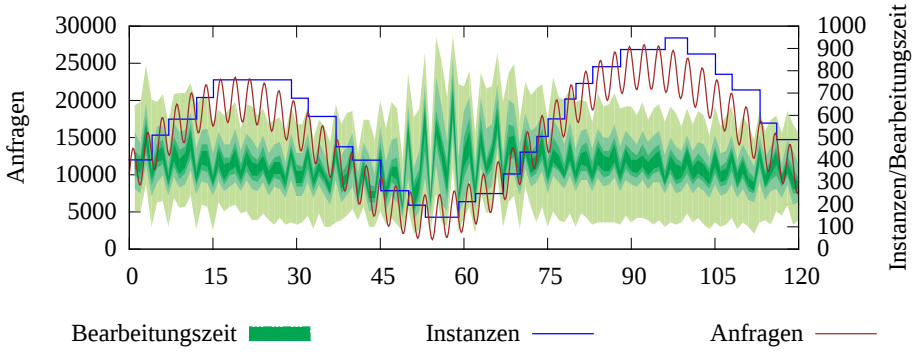


Abb. A.43.: Zentrale Arithmetische Methode

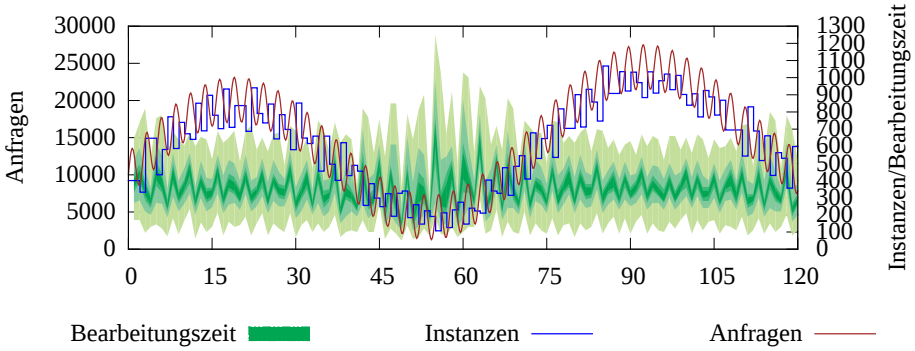


Abb. A.44.: Zentrale Fuzzylogik

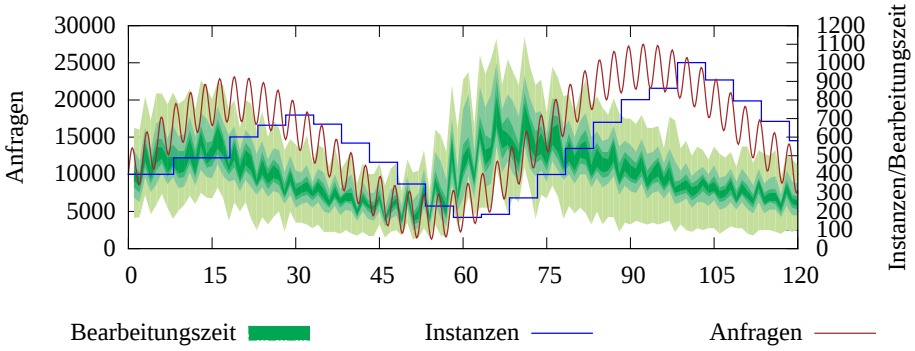


Abb. A.45.: Dezentrale Arithmetische Methode ohne Aggregation

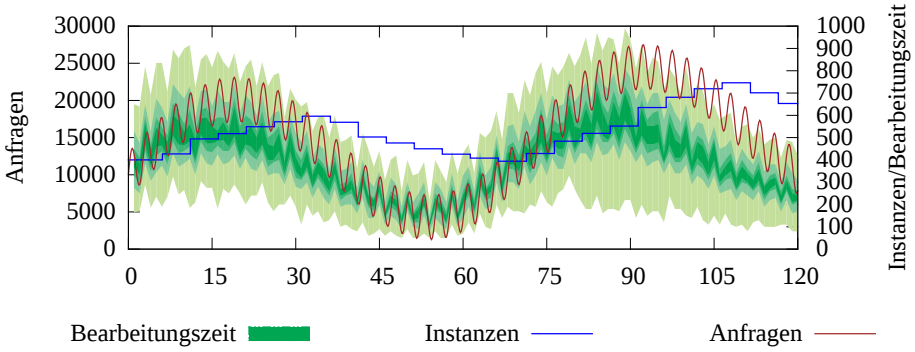


Abb. A.46.: Dezentrale Arithmetische Methode mit Aggregation

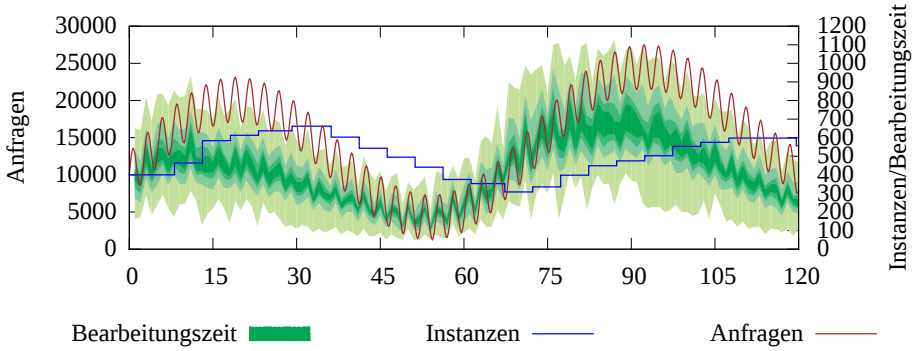


Abb. A.47.: Dezentrale Fuzzylogik ohne Aggregation

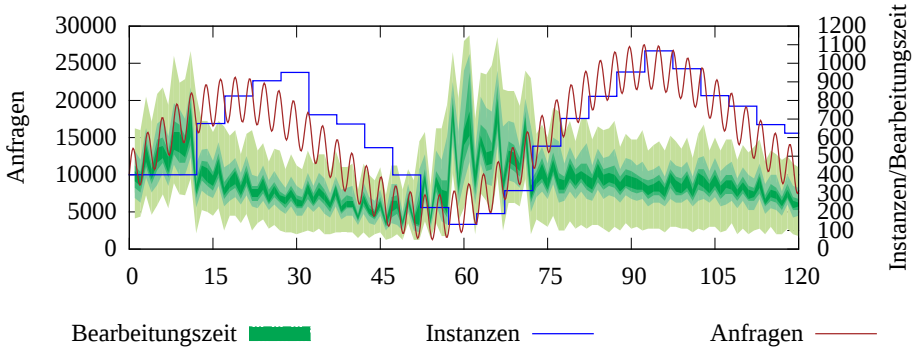


Abb. A.48.: Dezentrale Fuzzylogik mit Aggregation

A.2.7. Eingabe XG: $9000\sin(\frac{t}{700}) + 1500\sin(\frac{t}{25}) + 10000 + t$

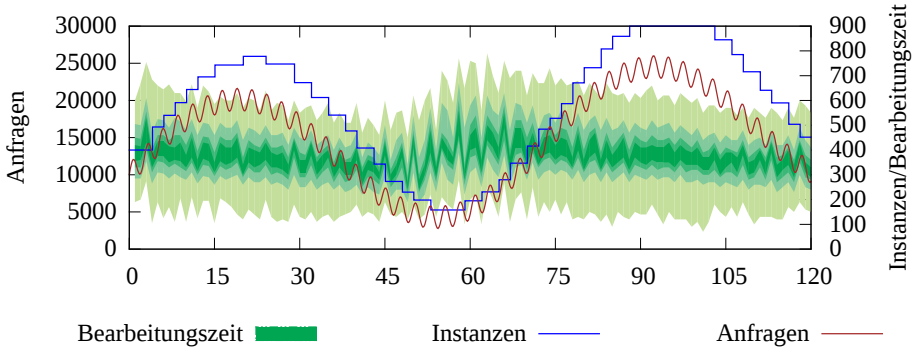


Abb. A.49.: Zentrale Arithmetische Methode

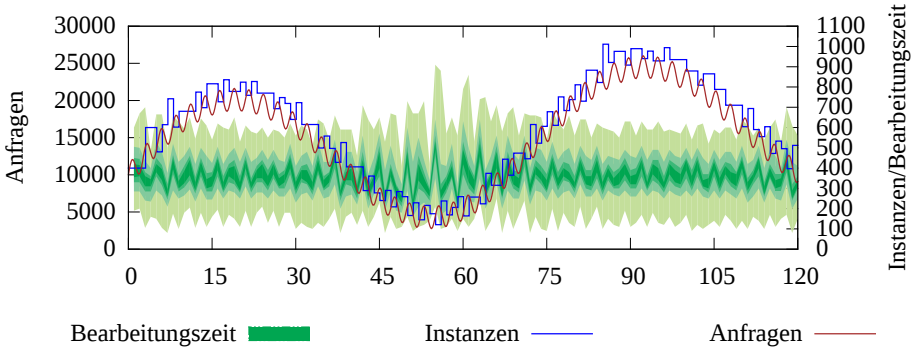


Abb. A.50.: Zentrale Fuzzylogik

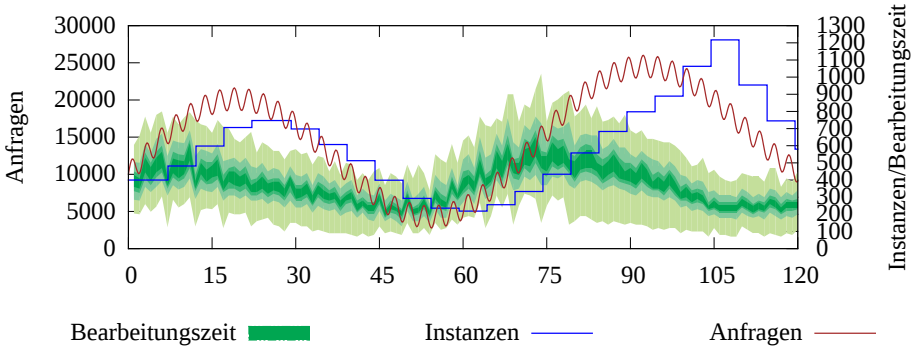


Abb. A.51.: Dezentrale Arithmetische Methode ohne Aggregation

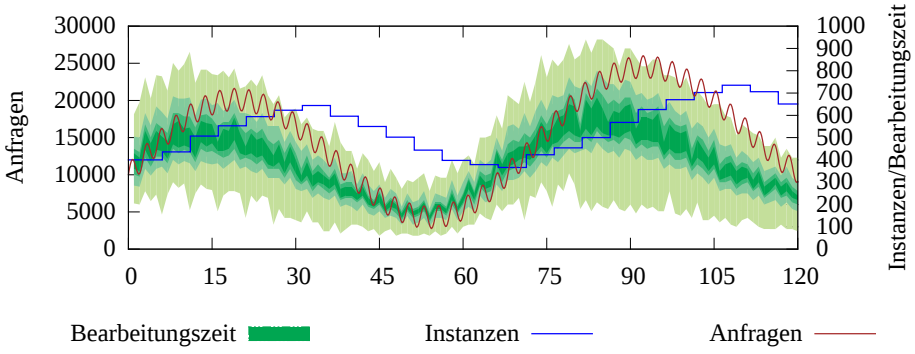


Abb. A.52.: Dezentrale Arithmetische Methode mit Aggregation

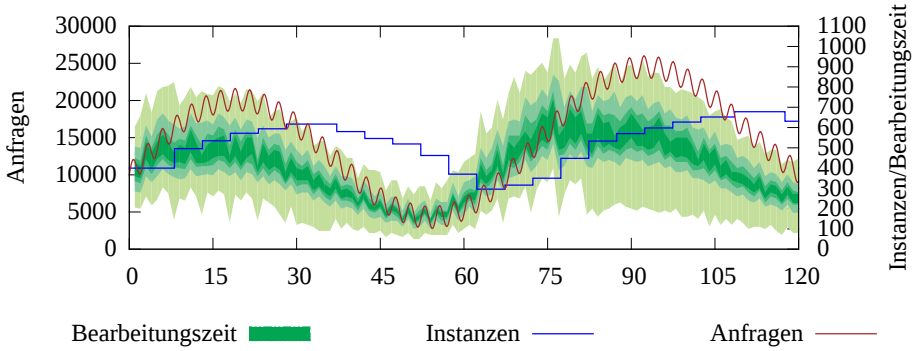


Abb. A.53.: Dezentrale Fuzzylogik ohne Aggregation

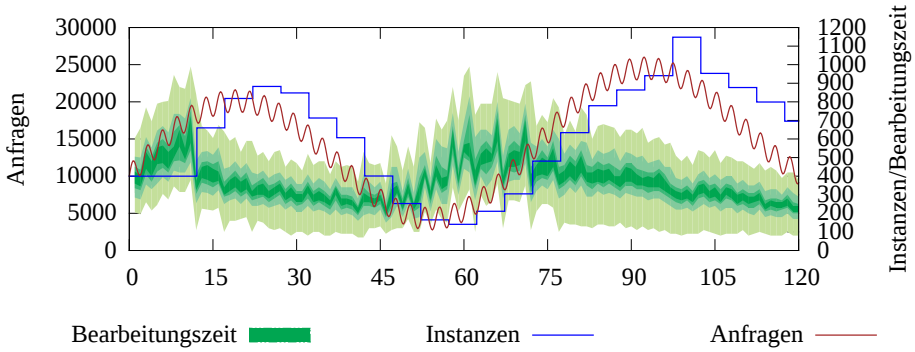


Abb. A.54.: Dezentrale Fuzzylogik mit Aggregation

B. Statistische Auswertung Cloud-Computing-Evaluation

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	12,186	856	285	1078,33	0,0113
zentral Fuzzylogik	10,340	949	357	1120,43	0,0092
dezentral Arithmetisch	11,742	1173	677	1110,53	0,0106
dezentral Arithmetisch aggr	13,453	860	280	1037,76	0,0130
dezentral Fuzzylogik	12,419	659	76	1073,25	0,0116
dezentral Fuzzylogik aggr	12,045	1444	869	1135,00	0,0106

Tab. B.1.: Statistische Kenngrößen für Szenario A

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,737	555	409	334,59	0,0351
zentral Fuzzylogik	9,719	481	335	341,32	0,0285
dezentral Arithmetisch	9,250	400	254	364,97	0,0253
dezentral Arithmetisch aggr	9,041	432	289	428,13	0,0211
dezentral Fuzzylogik	8,349	400	247	389,95	0,0214
dezentral Fuzzylogik aggr	9,208	540	385	370,61	0,0248

Tab. B.2.: Statistische Kenngrößen für Szenario B

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,650	1368	1178	898,99	0,0130
zentral Fuzzylogik	11,196	2632	2519	918,38	0,0122
dezentral Arithmetisch	19,503	1403	1017	939,83	0,0208
dezentral Arithmetisch aggr	21,419	845	340	920,01	0,0233
dezentral Fuzzylogik	20,595	877	430	922,97	0,0223
dezentral Fuzzylogik aggr	24,728	1443	1275	820,32	0,0301

Tab. B.3.: Statistische Kenngrößen für Szenario XA

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,195	1523	1066	1165,99	0,0096
zentral Fuzzylogik	10,444	2851	2434	1197,98	0,0087
dezentral Arithmetisch	15,941	1724	1127	1213,89	0,0131
dezentral Arithmetisch aggr	20,440	988	337	1071,74	0,0191
dezentral Fuzzylogik	22,011	992	344	1045,57	0,0211
dezentral Fuzzylogik aggr	23,572	1708	1176	1085,91	0,0217

Tab. B.4.: Statistische Kenngrößen für Szenario XB

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	14,098	1574	1083	1163,11	0,0121
zentral Fuzzylogik	14,583	8771	8230	1251,47	0,0117
dezentral Arithmetisch	17,686	1505	917	1167,89	0,0151
dezentral Arithmetisch aggr	22,436	926	323	1045,35	0,0215
dezentral Fuzzylogik	16,611	1066	390	1194,27	0,0139
dezentral Fuzzylogik aggr	16,263	1762	1269	1251,79	0,0130

Tab. B.5.: Statistische Kenngrößen für Szenario XC

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	12,091	1544	1102	1163,47	0,0104
zentral Fuzzylogik	11,801	5408	4949	1221,95	0,0097
dezentral Arithmetisch	18,559	1661	1075	1159,76	0,0160
dezentral Arithmetisch aggr	23,096	959	279	1040,61	0,0222
dezentral Fuzzylogik	18,168	1128	455	1121,32	0,0162
dezentral Fuzzylogik aggr	14,180	1811	1426	1206,61	0,0118

Tab. B.6.: Statistische Kenngrößen für Szenario XD

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	12,655	1582	1085	1160,12	0,0109
zentral Fuzzylogik	12,724	6639	6166	1231,67	0,0103
dezentral Arithmetisch	16,412	1802	1201	1226,17	0,0134
dezentral Arithmetisch aggr	21,999	952	268	1089,70	0,0202
dezentral Fuzzylogik	17,909	1012	320	1167,77	0,0153
dezentral Fuzzylogik aggr	14,839	1871	1144	1274,60	0,0116

Tab. B.7.: Statistische Kenngrößen für Szenario XE

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	13,308	1563	1072	1164,42	0,0114
zentral Fuzzylogik	13,699	7884	7387	1238,00	0,0111
dezentral Arithmetisch	19,058	1552	971	1126,75	0,0169
dezentral Arithmetisch aggr	20,879	948	295	1080,56	0,0193
dezentral Fuzzylogik	26,197	952	395	1010,02	0,0259
dezentral Fuzzylogik aggr	15,391	1885	1261	1279,45	0,0120

Tab. B.8.: Statistische Kenngrößen für Szenario XF

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	11,683	1520	1068	1162,46	0,0101
zentral Fuzzylogik	10,933	4232	3801	1212,11	0,0090
dezentral Arithmetisch	15,912	1745	1165	1209,91	0,0132
dezentral Arithmetisch aggr	20,165	1013	362	1091,15	0,0185
dezentral Fuzzylogik	21,715	999	368	1054,71	0,0206
dezentral Fuzzylogik aggr	14,396	1890	1194	1248,88	0,0115

Tab. B.9.: Gemittelte statistische Kenngrößen für Szenario XG

	σ	Start	Stopp	VM/h	Quotient
zentral Arithmetisch	12,4	1524,9	1093,4	1125,5	0,011002
zentral Fuzzylogik	12,2	5488,1	5069,4	1181,7	0,010322
dezentral Arithmetisch	17,6	1627,4	1067,6	1149,2	0,015299
dezentral Arithmetisch aggr	21,5	947,3	314,9	1048,4	0,020498
dezentral Fuzzylogik	20,5	1003,7	386,0	1073,8	0,019052
dezentral Fuzzylogik aggr	17,6	1767,1	1249,3	1166,86	0,015105

Tab. B.10.: Gemittelte statistische Kenngrößen für die Szenarien XA - XG

Literaturverzeichnis

[1] Allrutz, Ralf ; Cap, Clemens ; Eilers, Stefan ; Fey, Dietmar ; Haase, Helmut ; Hochberger, Christian ; Karl, Wolfgang ; Kolpatzik, Bernd ; Schmeck, Hartmut ; Ungerer, Theo u. a.: Organic Computing - Computer- und Systemarchitektur im Jahr 2010 / VDE/ITG/GI-Positionspapier. Version: 2003. `http://www.informatikperspektiven.de/fileadmin/redaktion/Presse/VDE-ITG-GI-Positionspapier_20Organic_20Computing.pdf`. 2003. – Forschungsbericht

[2] Amano, Akio ; Aritsuka, Toshiyuki ; Hataoka, Nobuo ; Ichikawa, Akira: On the Use of Neural Networks and Fuzzy Logic in Speech Recognition. In: *Proceedings of the International Joint Conference on Neural Networks* IEEE, 1989 (IJCNN 1989), S. 301–305

[3] Armbrust, Michael ; Fox, Armando ; Griffith, Rean ; Joseph, Anthony D. ; Katz, Randy H. ; Konwinski, Andrew ; Lee, Gunho ; Patterson, David A. ; Rabkin, Ariel ; Stoica, Ion ; Zaharia, Matei: Above the Clouds: A Berkeley View of Cloud Computing / Dept. Electrical Eng. and Comput. Science, University of California, Berkeley. Version: Feb 2009. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html`. 2009 (UCB/EECS-2009-28). – Forschungsbericht

[4] Bacchus, Fahiem: AIPS 2000 Planning Competition: The Fifth International Conference on Artificial Intelligence Planning and Scheduling Systems. In: *AI Magazine* 22 (2001), Nr. 3, S. 47. – AAAI

[5] Barroso, L.A. ; Hölzle, U.: The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. In: *Synthesis Lectures on Computer Architecture* 4 (2009), Nr. 1, S. 1–108. – Morgan & Claypool Publishers

[6] Barroso, Luiz A. ; Hölzle, Urs: The Case for Energy-Proportional

Computing. In: *IEEE Computer* 40 (2007), Nr. 12, S. 33–37

[7] Bojadziev, George ; Bojadziev, Maria: *Fuzzy Logic for Business, Finance, and Management*. World Scientific Publishing Co., Inc., 2007

[8] Branco, Miguel: *Topology-aware Gossip Dissemination for Large-scale Datacenters*, Instituto Superior Tecnico, Universidade Tecnica de Lisboa, Diplomarbeit, Oktober 2012

[9] Branco, Miguel ; Leitão, Joao ; Rodrigues, Luís: Bounded Gossip: A Gossip Protocol for Large-Scale Datacenters. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal : ACM, 2013 (SAC 2013). – ISBN 978–1–4503–1656–9, 591–596

[10] Brinkschulte, Uwe ; Pacher, Mathias ; Von Renteln, Alexander: Towards an Artificial Hormone System for Self-Organizing Real-Time Task Allocation. In: *Software Technologies for Embedded and Ubiquitous Systems*. Springer, 2007, S. 339–347

[11] Brockmann, Werner ; Maehle, Erik ; Grosspietsch, Karl-Erwin ; Rosemann, Nils ; Jakimovski, Bojan: ORCA: An Organic Robot Control Architecture. In: *Organic Computing—A Paradigm Shift for Complex Systems*. Springer, 2011, S. 385–398

[12] Castro, Miguel ; Druschel, Peter ; Hu, Y C. ; Rowstron, Antony: Exploiting network proximity in peer-to-peer overlay networks / Microsoft Research. 2002 (MSR-TR-2002-82). – Forschungsbericht

[13] Castro, Miguel ; Druschel, Peter ; Kermarrec, Anne-Marie ; Nandi, Animesh ; Rowstron, Antony ; Singh, Atul: SplitStream: High-Bandwidth Multicast in Cooperative Environments. In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles* ACM, 2003 (SOSP '03), S. 298–313

[14] Castro, Miguel ; Druschel, Peter ; Kermarrec, Anne-Marie ; Rowstron, Antony: SCRIBE: A Large-Scale and Decentralized Application-Level Multicast Infrastructure. In: *IEEE Journal on Selected Areas in Communications* 20 (2002), Nr. 8, S. 1489–1499. – IEEE

[15] Chieu, Trieu C. ; Mohindra, Ajay ; Karve, Alexei A. ; Segal, Alla: Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In: *IEEE International Conference on e-Business Engineering*, 2009 (ICEBE '09), S. 281–286

[16] Dean, Jeffrey ; Ghemawat, Sanjay: MapReduce: Simplified Data Processing on Large Clusters. In: *Communications of the ACM* 51 (2008), Nr. 1, S. 107–113. – ACM

[17] Di Nitto, Elisabetta ; Dubois, Daniel J. ; Mirandola, Raffaela ; Saffre, Fabrice ; Tateson, Richard: Applying Self-Aggregation to Load Balancing: Experimental RSesults. In: *Proceedings of the 3rd International Conference on Bio-Inspired Models of Network, Information and Computing Sytems* Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), 2008 (BIONETICS 2008), S. 14

[18] Di Nitto, Elisabetta ; Dubois, Daniele J. ; Mirandola, Raffaela: Self-Aggregation Algorithms for Autonomic Systems. In: *Proceedings of the 2nd International Conference on Bio-Inspired Models of Network, Information and Computing Systems* IEEE, 2007 (BIONETICS 2007), S. 120–128

[19] Dorigo, Marco ; Maniezzo, Vittorio ; Colorni, Alberto: Ant System: Optimization by a Ccolony of Cooperating Agents. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 26 (1996), Nr. 1, S. 29–41. – IEEE

[20] El-Ansary, Sameh ; Alima, Luc ; Brand, Per ; Haridi, Seif: Efficient broadcast in structured P2P networks. In: *Peer-to-Peer Systems II* 2735 (2003), S. 304–314. – Springer

[21] Fan, Xiaobo ; Weber, Wolf-Dietrich ; Barroso, Luiz A.: Power Provisioning for a Warehouse-sized Computer. In: *Proceedings of the 34th Annual International Symposium on Computer Architecture*. New York, NY, USA : ACM, 2007 (ISCA '07), 13–23

[22] Feng, Hong ; Minglu, Li ; Minyou, Wu ; Jiadi, YU: PChord: Improvement on Chord to Achieve Better Routing Efficiency by Exploiting Proximity. In: *IEICE Transactions on Information and Systems* 89 (2006), Nr. 2, S. 546–554. – The Institute of Electronics, Information and Communication Engineers (IEICE)

[23] Ghanbari, H. ; Simmons, B. ; Litoiu, M. ; Iszlai, G.: Exploring Alternative Approaches to Implement an Elasticity Policy. In: *IEEE International Conference on Cloud Computing*, 2011 (CLOUD 2011), S. 716–723

[24] Haas, Zygmunt J. ; Halpern, Joseph Y. ; Li, Li: Gossip-Based Ad Hoc Routing. In: *IEEE/ACM Transactions on Networking* 14 (2006), Nr. 3, S. 479–491. – IEEE Press

[25] Hamilton, James: *Cost of Power in Large-Scale Data Centers*. `http://perspectives.mvdirona.com/2008/11/28/CostOfPowerInLargeScaleDataCenters.aspx`, November 2008

[26] Hamilton, James: Internet-scale service efficiency. In: *Proceedings of 2nd Workshop on Large-Scale Distributed Systems and Middleware*, ACM, 2008 (LADIS 2008)

[27] Hedetniemi, Sandra M. ; Hedetniemi, Stephen T. ; Liestman, Arthur L.: A Survey of Gossiping and Broadcasting in Communication Networks. In: *Networks* 18 (1988), Nr. 4, S. 319–349. – Wiley Online Library

[28] Henderson, Thomas R. ; Roy, Sumit ; Floyd, Sally ; Riley, George F.: ns-3 Project goals. In: *Proceeding from the 2006 Workshop on ns-2: The IP Network Simulator* ACM, 2006 (WNS2 '06), S. 13

[29] Hoffmann, Jörg: FF: The Fast-Forward Planning System. In: *AI Magazine* 22 (2001), Nr. 3, S. 57. – AAAI

[30] Hoffmann, Martin ; Wittke, Michael ; Hahner, Joörg ; Müller-Schloer, Christian: Spatial Partitioning in Self-Organizing Smart Camera Systems. In: *IEEE Journal of Selected Topics in Signal Processing* 2 (2008), Nr. 4, S. 480–492. – IEEE

[31] Horn, Paul: Autonomic Computing: IBM's Perspective on the State of Information Technology. In: *Computing Systems* 15 (2001), Nr. Jan, S. 1–40. – IBM

[32] Jakimovski, Bojan ; Litza, Marek ; Mösch, Florian ; Auf, Adam El S.: Development of an Organic Computing Architecture for Robot Control. In: *GI Jahrestagung (1)*, 2006, S. 145–152

[33] Jänen, Uwe ; Spiegelberg, Henning ; Sommer, Lars ; Mammen, Sebastian von ; Brehm, Jürgen ; Hähner, Jörg: Object Tracking as Job-Scheduling Problem. In: *Proceedings of the 7th ACM/IEEE International Conference on Distributed Smart Cameras*, 2013 (ICDSC 2013)

[34] Kephart, J.O. ; Chess, D.M.: The vision of Autonomic Computing. In: *Computer* 36 (2003), Nr. 1, S. 41–50. – IEEE

[35] Kluge, Florian: *Autonomic- und Organic-Computing-Techniken für eingebettete Echtzeitsysteme*, Universität Augsburg, Diss., 2011

[36] Kluge, Florian ; Uhrig, Sascha ; Mische, Jörg ; Ungerer, Theo: A two-layered Management Architecture for Building Adaptive Real-Time Systems. In: *Software Technologies for Embedded and Ubiquitous Systems*. Springer, 2008, S. 126–137

[37] Kyasanur, Pradeep ; Choudhury, Romit R. ; Gupta, Indranil: Smart Gossip: An Adaptive Gossip-Based Broadcasting Service for Sensor Networks. In: *Proccedings of the International Conference on Mobile Adhoc and Sensor Systems* IEEE, 2006 (MASS 2006), S. 91–100

[38] Levis, Philip A. ; Patel, Neil ; Culler, David ; Shenker, Scott: *Trickle: A self regulating algorithm for code propagation and maintenance in wireless sensor networks*. Computer Science Division, University of California, 2003

[39] Maekawa, Mamoru: A N Algorithm for Mutual Exclusion in Decentralized Systems. In: *ACM Transactions on Compututer Systems* 3 (1985), Mai, Nr. 2, S. 145–159. – ACM

[40] Mamei, Marco ; Zambonelli, Franco: Spatial Computing: The TOTA Approach. In: *Self-star Properties in Complex Information Systems*. Springer, 2005, S. 307–324

[41] Marinos, Alexandros ; Briscoe, Gerard: Community Cloud Computing. In: *Cloud Computing* Bd. 5931. Springer, 2009, S. 472–484

[42] Mell, Peter ; Grance, Tim: The NIST Definition of Cloud Computing / National Institute of Standards and Technology, Information Technology Laboratory. Version: 26, Juli 2009. `http://www.csrc.nist.gov/groups/SNS/cloud-computing/`. 2009. – Forschungsbericht

[43] Mösch, Florian ; Litza, Marek ; Auf, Adam El S. ; Maehle, Erik ; Grosspietsch, Karl E. ; Brockmann, Werner: ORCA–Towards an Organic Robotic Control Architecture. In: *Self-Organizing Systems*. Springer, 2006, S. 251–253

[44] Muller-Schloer, Christian: Organic computing-on the feasibility of controlled emergence. In: *International Conference on Hardware/Software Codesign and System Synthesis* IEEE, 2004 (CODES+ISSS 2004), S. 2–5

[45] Nakrani, Sunil ; Tovey, Craig: On Honey Bees and Dynamic Server

Allocation in Internet Hosting Centers. In: *Adaptive Behavior* 12 (2004), Nr. 3-4, S. 223–240. – SAGE Publications

[46] Nickschas, Manuel ; Brinkschulte, Uwe: CARISMA - A Service-Oriented, Real-Time Organic Middleware Architecture. In: *Journal of Software* 4 (2009), Nr. 7, S. 654–663. – Academy Publisher

[47] Plaxton, C.G. ; Rajaraman, R. ; Richa, A.W.: Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In: *Theory of Computing Systems* 32 (1999), Nr. 3, S. 241–280. – Springer

[48] Prothmann, Holger ; Rochner, Fabian ; Tomforde, Sven ; Branke, Jörgen ; Müller-Schloer, Christian ; Schmeck, Hartmut: Organic Control of Traffic Lights. In: *Procceding of the 5th International Conference on Autonomic and Trusted Computing*, Springer, 2008 (ATC 2008), S. 219–233

[49] Rada-Vilela, Juan: fuzzylite: A Fuzzy Logic Control Library in C++. In: *Proccedings of the Open Source Developer Conference*, http://www.fuzzylite.com, 2013

[50] Randles, M. ; Lamb, D. ; Taleb-Bendiab, A: A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing. In: *Proccedings of the 24th International Workshop on Advanced Information Networking and Applications*, IEEE, April 2010 (WAINA 2010), S. 551–556

[51] Randles, M. ; Taleb-Bendiab, A ; Lamb, D.: Scalable Self-Governance Using Service Communities as Ambients. In: *Proccedings of the World Conference on Services - I*, 2009 (SERVICES 2009), S. 813–820

[52] Ratnasamy, Sylvia ; Francis, P. ; Handley, Mark ; Karp, Richard ; Shenker, Scott: A Scalable Content-Addressable Network. In: *Proceedings of the 2001 ACM Special Interest Group on Data Communication Conference on the Applications, Technologies, Architectures, and Protocols for Computer Communications*, ACM, 2001 (ACM SIGCOMM 2001), S. 161–172

[53] Ratnasamy, Sylvia ; Handley, Mark ; Karp, Richard ; Shenker, Scott: Topologically-Aware Overlay Construction and Server Selection. In: *Proceedings of the 21th Annual Joint Conference of the IEEE Computer and Communications Societies* Bd. 3 IEEE, 2002 (INFOCOM 2002), S.

1190–1199

[54] Richter, Urban ; Mnif, Moez ; Branke, Jürgen ; Müller-Schloer, Christian ; Schmeck, Hartmut: Towards a Generic Observer/Controller Architecture for Organic Computing. In: *GI Jahrestagung (1)* 93 (2006), S. 112–119

[55] Roth, Michael ; Schmitt, Julia ; Kiefhaber, Rolf ; Kluge, Florian ; Ungerer, Theo: Organic Computing Middleware for Ubiquitous Environments. In: *Organic Computing—A Paradigm Shift for Complex Systems*. Springer Basel, 2011, S. 339–351

[56] Roth, Michael ; Schmitt, Julia ; Kluge, Florian ; Ungerer, Theo: Information Dissemination in Distributed Organic Computing Systems with Distributed Hash Tables. In: *Proceddings of the 15th International Conference on Computational Science and Engineering* IEEE, 2012 (CSE 2012), S. 554–561

[57] Roth, Michael ; Schmitt, Julia ; Kluge, Florian ; Ungerer, Theo: DHT Broadcast Optimisation with ID Assignment Rules. In: *Proccedings of the 2013 Workshop on Embedded Self-Organizing Systems*. Berkeley, CA : USENIX, 2013 (ESOS 2013). – https://www.usenix.org/conference/esos13/workshop-program/presentation/Roth

[58] Rowstron, A. ; Druschel, P.: Pastry: Scalable, decentralized object Location, and routing for large-scale peer-to-Peer Systems. In: *Middleware 2001* Springer, 2001, S. 329–350

[59] Rowstron, Antony ; Kermarrec, Anne-Marie ; Castro, Miguel ; Druschel, Peter: SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In: *Networked Group Csommunication*. Springer, 2001, S. 30–43

[60] Saffre, Fabrice ; Tateson, Richard ; Halloy, José ; Shackleton, Mark ; Deneubourg, Jean L.: Aggregation Dynamics in Overlay Networks and Their Implications for Self-Organized Distributed Applications. In: *The Computer Journal* 52 (2009), Nr. 4, S. 397–412. – Oxford University Press on behalf of The British Computer Society

[61] Schmitt, Julia: *Zweistufige, Planer-basierte Organic Computing Middleware*, Universität Augsburg, Diss., 2013

[62] Schmitt, Julia ; Roth, Michael ; Kiefhaber, Rolf ; Kluge, Florian ; Ungerer, Theo: Concept of a Reflex Manager to Enhance the Planner Component of an Autonomic/Organic System. In: *Proccedings of the 8th International Conference on Autonomic and Trusted Computing*, 2011 (ATC 2011), S. 19–30. – Springer

[63] Sehgal, Anuj: Introduction to OpenStack. In: *Running a Cloud Computing Infrastructure with OpenStack* (2012). – University of Luxembourg

[64] Stoica, I. ; Morris, R. ; Karger, D. ; Kaashoek, M.F. ; Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: *Proceedings of the 2001 ACM Special Interest Group on Data Communication Conference on the Applications, Technologies, Architectures, and Protocols for Computer Communications* Bd. 31, 2001 (ACM SIGCOMM 2001 4), S. 149–160. – ACM

[65] Takagi, Hideyuki: Application of Neural Networks and Fuzzy Logic to Consumer Products. In: *Proceedings of the 22nd International Conference of Industrial Electronics Society* IEEE, 1992 (IECON 1996), S. 1629–1633

[66] Tomforde, Sven ; Prothmann, Holger ; Branke, Jörgen ; Hähner, J ; Müller-Schloer, C ; Schmeck, Hartmut: Possibilities and Limitations of Decentralised Traffic Control Systems. In: *Proccedings of the 2010 International Joint Conference on Neural Networks* IEEE, 2010 (IJCNN 2010), S. 1–9

[67] Tomforde, Sven ; Prothmann, Holger ; Rochner, Fabian ; Branke, Jörgen ; Hahner, J ; Müller-Schloer, C ; Schmeck, Hartmut: Decentralised Progressive Signal Systems for Organic Traffic Control. In: *Proccedings of the 2nd International Conference on Self-Adaptive and Self-Organizing Systems* IEEE, 2008 (SASO 2008), S. 413–422

[68] Vogels, WA: Head in the Clouds - The Power of Infrastructure as a Service. In: *First workshop on Cloud Computing and in Applications*, 2008 (CCA 2008)

[69] Xiong, Jiping ; Zhang, Youwei ; Hong, Peilin ; Li, Jinsheng: Chord6: IPv6 Based Topology-Aware Chord. In: *Proccedings of the Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services* IEEE, 2005 (ICAS-ICNS 2005), S. 4–4

[70] Xu, Zhichen ; Tang, Chunqiang ; Zhang, Zheng: Building topology-aware overlays using global soft-state. In: *Proceedings of the 23rd International Conference on Distributed Computing Systems* IEEE, 2003 (ICDCS 2003), S. 500–508

[71] Xu, Zhichen ; Zhang, Zheng: Building Low-Maintenance Expressways for P2P Systems / Hewlett-Packard Labs, Palo Alto, CA. 2002 (HPL-2002-41). – Forschungsbericht

[72] Yang, Hailong ; Breslow, Alex ; Mars, Jason ; Tang, Lingjia: Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ACM, 2013 (ISCA 2013)

[73] Zadeh, Lotfi A.: Fuzzy Sets. In: *Information and Control* 8 (1965), Nr. 3, S. 338–353. – Elsevier

[74] Zambonelli, Franco ; Mamei, Marco: Spatial computing: An Emerging Paradigm for Autonomic Computing and Communication. In: *Autonomic Communication* Bd. 3457. Springer, 2005, S. 44–57

[75] Zhao, Ben Y. ; Kubiatowicz, John ; Joseph, Anthony D.: Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing / University of California at Berkeley. 2001 (UCB/CSD-01-1141). – Forschungsbericht

[76] Zhuang, S.Q. ; Zhao, B.Y. ; Joseph, A.D. ; Katz, R.H. ; Kubiatowicz, J.D.: Bayeux: An Architecture for Scalable and Fault-Tolerant Wide-Area Data Dissemination. In: *Proceedings of the 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video* ACM, 2001 (NOSSDAV 2001), S. 11–20

Abbildungsverzeichnis

Tabellenverzeichnis