2. August 2021

FH MÜNSTER
University of Applied Sciences

Bachelorarbeit zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.) im Studiengang Informatik, dual

Erweiterung des CAP-Theorems zur Charakterisierung von (Micro-)Service-Architekturen

am Beispiel einer Monitoring-Anwendung

Leon Alexander Kraß

Matrikel-Nr.: 947743

Erstprüfer:
Prof. Dr. Patrick Stalljohann

Zweitprüfer:
Marc Bisping (Brillux GmbH & Co. KG)

Vorwort

Gender-Hinweis

Aus Gründen der besseren Lesbarkeit wird bei personenbezogenen Bezeichnungen in dieser Arbeit vornehmlich die maskuline Form angeführt. Entsprechende Begriffe sind im Sinne der Gleichberechtigung aller Geschlechter zu verstehen und sollen keinesfalls eine Diskriminierung darstellen.

Danksagung

Ich möchte mich an dieser Stelle bei allen Parteien bedanken, welche mich während der Ausarbeitung dieser Bachelorarbeit unterstützt und motiviert haben.

Ein besonderer Dank gilt Herrn Prof. Dr. Stalljohann, der diese Arbeit seitens der Fachhochschule Münster betreut und begutachtet hat. Für die konstruktive Kritik und die bereichernden Hilfestellungen möchte ich mich herzlich bedanken.

Ebenso möchte ich Herrn Marc Bisping für die Betreuung der Arbeit seitens der Brillux GmbH. & Co. KG herzlich danken. Die mir eröffneten Möglichkeiten haben nicht nur diese Arbeit, sondern auch meinen beruflichen Werdegang bereichert. Dieser Dank gilt ebenso der Brillux GmbH & Co. KG, welche mein duales Studium getragen und gefördert hat.

Ein großer Dank gilt allen Personen, welche mich während meines dualen Studiums unterstützt haben. Hier gilt mein Dank vor allem Herrn Marko Simonovic, welcher im Rahmen des ausbildungsintegrierten Studiums mein Ausbilder war. Ebenso möchte ich mich bei allen Arbeitskollegen für die gute Atmosphäre bedanken, die ein ständiger Teil meiner Ausbildung war.

Ich möchte des Weiteren allen Lehrkräften der Fachhochschule Münster und des Ludwig-Erhardt Berufskollegs Münster für die mir ermöglichte Bildung danken. Ohne die gewonnenen Erfahrungen und Erkenntnisse, wäre mir die Ausarbeitung dieser Arbeit nicht möglich gewesen.

Zuletzt möchte ich mich bei all meinen Freunden und besonders bei meiner Familie bedanken, die mir in jeder Situation zur Seite stehen. Der größte Dank gilt hierbei meiner verstorbenen Mutter. Danke für Alles, Mama!

Terminologie

Symbole, Formeln und Einheiten

Die Einheiten von Größen, welche im Rahmen dieser Arbeit genutzt bzw. definiert werden, werden in eckigen Klammern („[...]“) hinter den jeweiligen Werten und Symbolen angegeben.

Für die physikalische Größe der Zeit wird, sofern nicht anders spezifiziert, das Symbol t genutzt. Die Einheit der Zeit kann beispielsweise in Sekunden („[s]“), Minuten („[min]“) oder Millisekunden („[ms]“) angegeben werden.

Wird für ein Symbol oder einen Wert keine konkrete Einheit formuliert, ist anzunehmen, dass diese Größe einheitslos ist. In manchen Fällen sind einheitslose Größen explizit mit „arbitrary unit“ („[a.u.]“) gekennzeichnet.

Das arithmetische Mittel über eine Wertemenge wird mit „$\bar{\cdot}$“ gekennzeichnet.

Mengen

Die Menge der natürlichen Zahlen beinhaltet im Rahmen dieser Arbeit nicht die Zahl Null: $\mathbb{N} = \{1, 2, 3, \dots\}$. Sofern die Null in der Menge enthalten sein soll, wird dies explizit gekennzeichnet: $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$.

Es wird die Schreibweise $\mathbb{R}^{<x} = \{a \in \mathbb{R} \mid a < x\}$ genutzt. Dabei kann „$<$“ belibig durch „$\leq$“, „$\geq$“ oder „$>$“ substituiert werden.

Folgende, im Allgemeinen nicht eindeutig definierte Mengen-Operatoren, wurden in dieser Arbeit verwendet: Sei M eine beliebige Menge, dann ist

- $|M|$ die Mächtigkeit der Menge, also die Anzahl der Elemente der Menge M.
- M^2 das Kreuzprodukt der Menge M mit sich selbst: $M^2 = M \times M$

Intervalle

Intervalle werden in dieser Arbeit wie folgt angegeben:

Seien $a, b \in \mathbb{R}$, dann gilt

- $[a; b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$
- $[a; b) = \{x \in \mathbb{R} \mid a \leq x < b\}$
- $(a; b] = \{x \in \mathbb{R} \mid a < x \leq b\}$
- $(a; b) = \{x \in \mathbb{R} \mid a < x < b\}$

Unkonventionelle Schreibweisen

In manchen Formeln wurden Ausdrücke wie $a = \text{"A"}$ oder $cmp(\text{"A"}, \text{"B"})$ genutzt. Die in Anführungszeichen dargestellten Buchstaben drücken dabei Elemente der Menge des lateinischen Alphabetes aus. Sie besitzen keinerlei mathematische Eigenschaften, werden jedoch für Vergleiche verwendet. Hierbei werden weitere Eigenschaften der Symbole im Kontext definiert.

„Master-Slave“-Terminologie

Da die Terme „Master“ und „Slave“ historisch bedingt sehr umstritten sind, wurde in dieser Arbeit davon abgesehen diese Ausdrücke zu verwenden, auch wenn in Quellen diese Formulierungen verwendet wurden. Stattdessen werden die neutralen Begriffe „Primary“ und „Secondary“ genutzt.

Inhaltsverzeichnis

Tabellen

Abbildungen

Diagramme

Formeln

1. Kurzfassung

Softwareentwickler und -architekten werden vor immer größere Herausforderungen gestellt. Sie müssen in der Lage sein, wichtige (Entwurfs-)Entscheidungen zu treffen. Diese Entscheidungen müssen oft ohne Detailwissen über mögliche Softwaresysteme getroffen werden. Daher ist es wichtig, dass für Softwareentwickler und -architekten bestimmte Metriken und Kennwerte über Systeme bereitgestellt werden, die ihnen dabei helfen, spezifische Eigenschaften schnell zu identifizieren.

Als Grundlage für den Entwurf vieler Softwaresysteme hat sich das CAP-Theorem etabliert. Es besagt, dass ein System maximal zwei der drei Eigenschaften Verfügbarkeit, Konsistenz und Partitionstoleranz erfüllen kann. Jedoch steht das CAP-Theorem in der Kritik, was als Anreiz gelten soll, diese Kritik aufzugreifen und das CAP-Theorem zu erweitern bzw. umzuformulieren.

In dieser Arbeit wird die Kritik am CAP-Theorem genutzt, um ein neues Werkzeug, die CAP-Charakteristik, zu schaffen, mit dem Softwaresysteme bewertet und verglichen werden können. Diese Bewertung erfolgt anhand quantitativer Maße, welche an den Systemen gemessen werden und in Relation zueinander gestellt werden. Durch verschiedene Blickwinkel auf die ermittelten Messwerte können tiefere Analysen angestellt oder aber auch Fingerabdrücke von Systemen erstellt werden.

Anhand eines Versuches wird die hier aufgestellte Definition der CAP-Charakteristik überprüft und es wird reflektiert, ob die Charakteristik in der dargelegten Form ein geeignetes Mittel zur Bewertung von (Micro-)Service-Architekturen darstellt. Verbesserungspotenzial, welches sich aus dem Versuch ergibt, wird angeführt und ein Ausblick gegeben, wie die Charakteristik in Zukunft genutzt werden könnte.

Neben der CAP-Charakteristik wird in dieser Arbeit ein Überblick über bestimmte Begriffe der Softwarearchitektur sowie ein Exkurs zu Docker und Kubernetes gegeben, da diese Themenfelder eng mit dem Kontext der CAP-Charakteristik und dem durchgeführten Versuch zusammenhängen.

2. Einleitung

Motivation

In Zeiten, in denen Softwareentwickler und -architekten mit immer schneller werdenden Entwicklungszyklen und sich ständig ändernden Anforderungen auseinandersetzen müssen, ist es kaum überraschend, dass viele Softwareprojekte in der Praxis scheitern. Dies zeigt z. B. der „CHAOS Report“ der Standish Group [1]. Demnach haben im Jahr 2015 bloß 44% der von der Standish Group überwachten Softwareprojekte ihr Budget eingehalten. Gerade einmal 40% der überwachten Projekte sind in der geplanten Zeitvorgabe fertiggestellt worden.

Dies zeigt, dass es notwendig ist, Softwareprojekte gleichzeitig strukturiert, aber auch flexibel zu gestalten. Hierzu können in der Praxis viele Hilfsmittel und Prinzipien verfolgt werden, die einen guten Verlauf eines Projekts unterstützen. Das Festlegen zu verfolgender Prinzipien für den Lebenszyklus von Software ist typischerweise die Aufgabe eines Softwarearchitekten [2]. Damit ein Softwarearchitekt jedoch entscheiden kann, welche Softwarekomponenten er auswählen soll, benötigt er gewisse Anhaltspunkte, um eine ganzheitliche, in sich bündige Architektur zu entwerfen.

Beispiel: Entwurf einer Monitoring-Plattform

Für das Monitoring-System Grafana stehen verschiedene Datenquellen bereit [3]. Grafana selbst bietet hauptsächlich eine grafische Oberfläche für die Darstellung von Daten aus Datenquellen an. Somit ist es erforderlich, neben Grafana auch Datenquellen als Komponenten einer Monitoring-Anwendung bereitzustellen. Zwei mögliche Datenquellen sind hierbei die NoSQL Datenbank Redis und die relationale Datenbank PostgreSQL [4].

Ein Softwarearchitekt, der nun die Monitoring-Plattform entwerfen soll, muss entscheiden für welchen Anwendungsfall welche Datenquelle besser geeignet ist. Hat der Architekt keine Vorerfahrung mit den beiden Systemen, so muss er bestimmte Eigenschaften und Kennziffern der Systeme in Erfahrung bringen, um entsprechende Entwurfsentscheidungen zu treffen.

Entscheidungsgrundlagen

Einen wichtigen Anhaltspunkt, der bei dem Entwurf von Softwaresystemen bisher von großer Bedeutung gewesen ist, stellt das CAP-Theorem nach Gilbert und Lynch [5] dar. Es besagt, dass ein System maximal zwei der drei Eigenschaften Verfügbarkeit, Konsistenz und Partitionstoleranz erfüllen kann. Diese Einschränkung gilt in vielen Kreisen, wie z. B. in der

NoSQL-Bewegung, als Grundlage der Entscheidung über das Design von Softwarearchitekturen [6].

Doch es gilt zu hinterfragen, ob es sinnvoll ist, das CAP-Theorem in der von Gilbert und Lynch formulierten Form als Entscheidungsgrundlage zu nutzen. Viele Kritiken stufen gewisse Aspekte und Nuancen des CAP-Theorems als bedenklich bzw. ungenau ein.

Die CAP-Charakteristik

Die Kritiken am CAP-Theorem werden aufgegriffen und genutzt, um ein neues Werkzeug zu schaffen, das den Entwurfs- und Entwicklungsprozess von (Micro-)Service-Architekturen unterstützt. Ein solches Werkzeug muss in der Lage sein, einen Softwarearchitekten bei seiner Entscheidungsfindung zu unterstützen.

Um dieses Ziel zu erreichen, wird die sogenannte CAP-Charakteristik eingeführt. Sie dient der Bewertung und dem Vergleich verschiedener Architekturen. Anhand eines Versuches wird die Nutzung der CAP-Charakteristik demonstriert und evaluiert.

Bevor die CAP-Charakteristik aufgestellt werden kann, ist es zunächst jedoch nötig den theoretischen Hintergrund des CAP-Theorems aufzuarbeiten und in Bezug auf die CAP-Charakteristik weitere Grundlagen und Definitionen einzuführen.

3. Theoretische Grundlagen

3.1 (Micro-)Service-Architekturen

Ziel dieser Arbeit ist es, (Micro-)Service-Architekturen mit Hilfe einer Erweiterung des CAP-Theorems zu bewerten. Bevor jedoch erläutert wird, wie diese Bewertung durchgeführt wird, wird in diesem Abschnitt erklärt, was der eigentliche Gegenstand der Bewertung ist. Dies hat den Hintergrund, dass die Festlegung von Maßstäben in der Regel sehr kontextbezogen ist.

Zunächst gilt es zu verstehen, was sich hinter dem Begriff (Micro-)Service-Architekturen verbirgt. Der Begriff setzt sich zusammen aus den Teilen (Micro-)Service und Architekturen. Dabei stellt der Begriff Architekturen eine Kurzform des Wortes Softwarearchitekturen dar. Diese beiden Teilbegriffe werden nun erläutert:

Softwarearchitekturen

Der Begriff der Softwarearchitektur wird in verschiedenen Quellen sehr unterschiedlich definiert. Angelehnt an die SO/IEC/IEEE-Standards 42010:2011 wird die Softwarearchitektur in [2] von Starke et al. wie folgt beschrieben:

> *„Die Softwarearchitektur definiert die grundlegenden Prinzipien und Regeln für die Organisation eines Systems sowie dessen Strukturierung in Bausteinen und Schnittstellen und deren Beziehungen zueinander wie auch zur Umgebung. Dadurch legt sie Richtlinien für den gesamten Systemlebenszyklus, angefangen bei Analyse über Entwurf und Implementierung bis zu Betrieb und Weiterentwicklung, wie auch für die Entwicklungs- und Betriebsorganisation fest.“*

Daraus ergibt sich, dass der Begriff zwei zentrale Gesichtspunkte berücksichtigt: Die Softwarearchitektur beschäftigt sich auf der einen Seite mit der Organisation der Prinzipien, welche für die Entwicklungsarbeit und den Lebenszyklus von Softwaresystemen gelten. Auf der anderen Seite umfasst die Softwarearchitektur aber auch die Beschreibung des Aufbaus von Softwaresystemen und deren Zerlegungen in verschiedene Komponenten, welche über Schnittstellen miteinander kommunizieren [2]. Im Rahmen dieser Arbeit ist vor allem der letzte Punkt, also der grundlegende Aufbau von Softwaresystemen, von Bedeutung.

Softwaresysteme, Komponenten und Schnittstellen

Nach Ludewig und Lichter [7] lassen sich Softwaresysteme beschreiben, als

> *„eine Menge von Software-Einheiten und ihren Beziehungen, wenn sie gemeinsam einem bestimmten Zweck dienen.“*

Die hier genannten Software-Einheiten lassen sich mit dem Synonym „Komponente“ bezeichnen. Komponenten bilden Teileinheiten des Softwaresystems ab, welche bestimmte Funktionalitäten bereitstellen und mit anderen Komponenten kommunizieren. Dies entspricht der Definition der (System-) Komponenten nach Grone et al. in [8]:

> *„eine Komponente ist ein aktiver Teil des (abstrakten) Systems [...]; eine Komponente stellt eine definierte Funktion bereit und kommuniziert mit anderen Teilen des Systems. Alle Teile des Systems können einen eigenen Zustand haben.“*
> *(Übersetzung des Autors)*

Die Kommunikation der Komponenten erfolgt über Schnittstellen, was die Definition einer Schnittstelle von Ludewig und Lichter [7] widerspiegelt:

> *„Die Schnittstelle einer Komponente stellt die Leistungen der Komponente für ihre Umgebung zur Verfügung und/oder fordert Leistungen, die sie aus Ihrer Umgebung benötigt.“*

Beispiel: Komponentendiagramm

Das in Abbildung 1 dargestellte Komponentendiagramm nach UML-Standard [9] bildet eine mögliche Architektur des Monitoring-Systems Grafana ab, welche den Zusammenhang der Begriffe Softwaresystem, Komponente und Schnittstelle verdeutlicht.

Nicht alle Funktionen des Systems werden über Grafana selbst bereitgestellt. Die Datenquellen und die Services, welche den Nutzer letztlich benachrichtigen oder die Authentifizierung bereitstellen, stellen wieder eigenständige Komponenten dar.

Es ist wichtig zu beachten, dass die hier genutzte Definition des Softwaresystems es erlaubt, die Komponenten wiederum als Systeme anzusehen. Die Modellierung per UML ermöglicht dabei, diese Komponenten in einem höheren Detailgrad weiter aufzuschlüsseln.

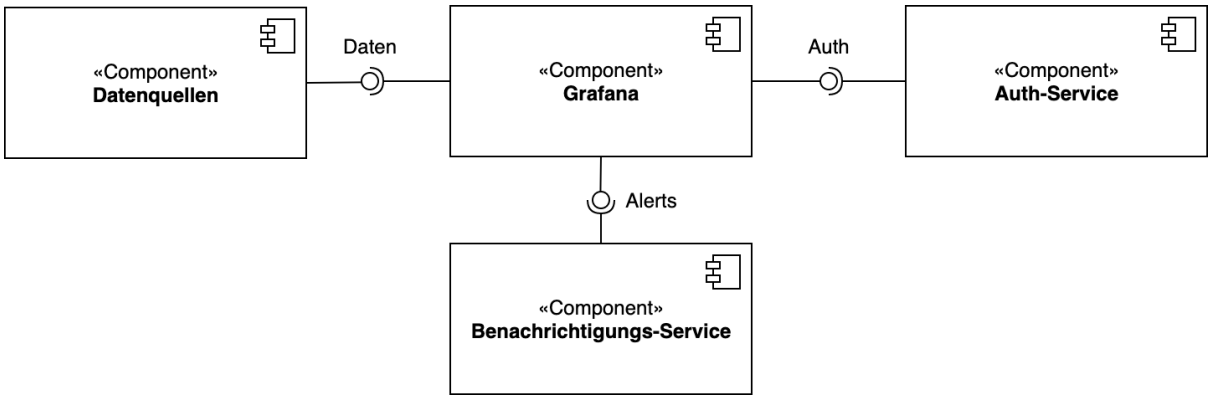

Abbildung 1: Beispiel – Komponentendiagramm Grafana

Mögliche Datenquellen für Grafana können z. B. Redis oder PostgreSQL sein [3], [4]. Diese Systeme können wiederum mit verschiedenen Architekturen modelliert werden.

(Micro-)Service

Service

Eine mögliche Form der Softwarearchitektur ist die Serviceorientierte Architektur, kurz SOA. Wie der Name bereits sagt, handelt es sich hierbei um eine Architektur, welche sich am Service als architektonische Einheit orientiert. Als Teil einer genaueren Definition von SOA, hat The Open Group [10] den Begriff Service wie folgt beschrieben:

> *"Ein Service:*
>
> - *Ist eine logische Repräsentation eines wiederholbaren Geschäftsprozesses, der einem spezifischen Zweck dient [...]*
> - *Ist in sich geschlossen*
> - *Darf aus anderen Services bestehen*
> - *Stellt für Konsumenten des Service eine ‚Black-Box' dar"*
>
> *(Übersetzung des Autors)*

Es fällt auf, dass diese Definition der vorherigen Definition der Komponente sehr ähnlich ist. Sowohl die Komponente, als auch der Service dienen einem bestimmten Geschäfts- bzw. Anwendungsfall. Die Funktionalität der Komponente wird über eine Schnittstelle bekanntgegeben. Dies impliziert, dass eine Komponente ebenfalls in sich geschlossen ist und nur die per Schnittstelle beschriebenen Funktionen für den Nutzer bereitstehen. Eine Komponente kann analog zum Service selbst aus kleineren Komponenten bestehen.

Microservice

Ein Microservice stellt eine spezialisierte Form des Service dar. Der Begriff sagt hierbei aus, dass ein Microservice ein in seiner Größe beschränkter Service ist. Dabei kann die Größe wiederum durch verschiedene Faktoren beeinflusst werden, wie z. B. durch die Größe des Teams, welches einen solchen Service implementiert, oder durch den Grad der Modularisierung eines Services [11].

Für den Kontext dieser Arbeit zeichnet sich ein Microservice vor allem dadurch aus, dass er erstens einen starken fachlichen Zusammenhang innerhalb seiner Grenzen aufweist und zweitens von anderen Services möglichst stark entkoppelt ist, also unabhängig von anderen Services operieren kann. Diese beiden Prinzipien werden erstens als hohe Kohäsion und zweitens als lose Kopplung bezeichnet. Nach [12] ist die Kohäsion

> *„ein Maß für die Stärke der funktionalen Bindung von Aktivitäten oder Sachverhalten in einer Betrachtungseinheit."*

Die Kopplung wird nach [2] definiert als die

> *„Beziehung zwischen den Bausteinen sowie die Stärke dieser Beziehung und der daraus resultierenden Abhängigkeit [...]."*

Je höher die Kohäsion eines Service ist und je loser er von anderen Services entkoppelt ist, desto eher kann er als Microservice betrachtet werden. Die Vorteile dieser Prinzipien für eine Softwarearchitektur werden in [2] näher erläutert.

Begriffssynthese

Da nun die Begriffe (Micro-)Service und Softwarearchitektur definiert sind, kann der zusammengesetzte Begriff (Micro-)Service-Architektur weiter ausgeführt werden:

Eine (Micro-)Service-Architektur ist eine Softwarearchitektur, welche ihre Komponenten in Form von (Micro-)Services umsetzt. Diese Komponenten folgen dabei den Prinzipien der hohen Kohäsion und losen Kopplung. Je stärker diese Konzepte ausgeprägt sind, desto eher stellt ein Service einen Microservice dar.

Verschiedene (Micro-)Service-Architekturen werden in Kapitel 5 dargestellt und anhand ihrer CAP-Eigenschaften charakterisiert. Dabei dienen die hier beschriebenen Begriffe und Konzepte als Grundlage weiterer Beschreibungen.

Zusätzlich wird im Folgenden der Begriff „Knoten eines Systems“ verwendet. Ein Knoten eines Systems stellt eine Einheit eines Systems dar, welche zur Laufzeit eine Funktionalität des Systems abdeckt. Dabei ist nicht festgelegt, ob ein Knoten nur einen Teil der Funktionalität des Systems bereitstellt oder autark das ganze System repräsentieren kann. Im Unterschied zu einer Komponente, welche eine abstrakte Vorlage für einen Knoten darstellt, existiert ein Knoten zur Laufzeit des Systems. Für eine Komponente eines Systems können also mehrere Knoten existieren, während das System ausgeführt wird.

3.2 Das CAP-Theorem in seiner ursprünglichen Form

Eric Brewer hat im Jahr 2000 erstmals auf dem „Symposium on Principles of Distributed Computing“ die Aussage formuliert, dass ein System höchstens zwei von drei Eigenschaften erfüllen könne. Diese Eigenschaften hat er als Konsistenz (engl. „**c**onsistency“), Verfügbarkeit (engl. „**a**vailability“) und Partitionstoleranz (engl. „**p**artition tolerance“) benannt [13].

Die Interpretation der drei Eigenschaften wurde von Brewer im ursprünglichen Ansatz nur grob umschrieben, jedoch haben Seth Gilbert und Nancy Lynch diese in Form des CAP-Theorems zwei Jahre später weiter spezifiziert [5]. Die Abkürzung CAP ergibt sich hierbei aus den (engl.) Anfangsbuchstaben der drei o. g. Eigenschaften, welche nach Gilbert und Lynch wie folgt definiert sind:

Verfügbarkeit

Ein System gilt als verfügbar, sofern es auf eine Anfrage in einer bestimmten Zeit antwortet. Hierbei wird keine obere Grenze für die Antwortzeit angegeben. Wichtig ist jedoch, dass auch Fehler mit einer entsprechenden Antwort signalisiert werden.

Beispiel: HTTP-Server

Überträgt man diese Definition zur Anschauung auf einen HTTP-Server nach RFC 1945 [14], so erfüllt dieser das Kriterium der Verfügbarkeit, solange er auf eine HTTP-Anfrage stets mit einer HTTP-Antwort antwortet. Diese Antwort kann dabei z. B. den Status-Code 200 (OK) oder aber 500 (Internal Server Error) enthalten. Auch letztere Statusausprägung erfüllt per Definition das Kriterium der Verfügbarkeit, obwohl man dies als Nutzer des Servers intuitiv gegenteilig interpretieren könnte.

Konsistenz

Gilbert und Lynchs Definition der Konsistenz beruht darauf, dass ein System als eine Datenquelle betrachtet wird, die mit Operationen ausgelesen oder verändert werden kann. Um die Eigenschaft der Konsistenz zu erfüllen, muss eine Datenquelle atomar sein:

1. Eine Datenquelle ist dann atomar, wenn die ihr zugrundeliegenden Operationen atomar, also in sich geschlossen, sind [15]. Zwei Operationen, welche dasselbe Datenobjekt als Ziel haben, dürfen nicht gleichzeitig ausgeführt werden.
2. Darüber hinaus ist es wichtig, dass mehrere (zusammenhängende) Operationen so verarbeitet werden, dass diese Verarbeitung als ein linearer Prozess verstanden werden kann, der zu einem schlussendlichen Zustand der Datenquelle führt.
3. Eine schreibende Operation gilt erst dann als abgeschlossen, sofern sie alle Knoten des Systems erreicht hat.

Beispiel: Bankkonto

Als Beispiel soll hier ein Bankkonto mit einem Kontostand von 50€ dienen. Der Inhaber des Kontos bekommt zum gleichen Zeitpunkt, in dem er eine Rechnung über 100€ bezahlt, eine Einzahlung von 50€ überwiesen. Die Einzahlung und die Bezahlung der Rechnung stellen hier die atomaren Operationen dar.

Die Operationen werden zur Verarbeitung in eine lineare Reihenfolge gebracht: Entweder wird zunächst die Einzahlung oder zunächst die Bezahlung ausgeführt. Die Verarbeitung resultiert in beiden Fällen in einem Kontostand von 0€.

Die Operation der Bezahlung bzw. Einzahlung gilt erst dann als abgeschlossen, wenn alle Knoten des Bankensystems diese verarbeitet haben, also z. B. sowohl ein Rechenzentrum in Münster als auch eines in Frankfurt den Kontostand des Inhabers angeglichen haben.

Partitionstoleranz

Für das Verständnis der Partitionstoleranz ist es zunächst wichtig zu verstehen, wie Gilbert und Lynch eine Partition definieren. Eine Partition entsteht dann, wenn die Kommunikation von Knoten in einem Teil eines Netzwerkes zu einem anderen Teil des Netzwerkes unterbrochen wird. Ein System gilt dann als partitionstolerant, wenn es trotz der Partition noch in der Lage ist, eine Antwort in einer bestimmten Zeit (vgl. Verfügbarkeit) oder eine atomare (vgl. Konsistenz) Antwort zu liefern.

Beispiel: Redis Sentinel

Die Key-Value-basierte Datenbank Redis implementiert eine Primary-Secondary-Architektur, welche es vorsieht mehrere Knoten einer Datenbank auszuführen. Die Daten werden hierbei unter den Knoten dupliziert bzw. synchronisiert. Neben dem Redis-Service wird auf jedem Knoten ein Sentinel-Service ausgeführt, der die Knoten überwacht.

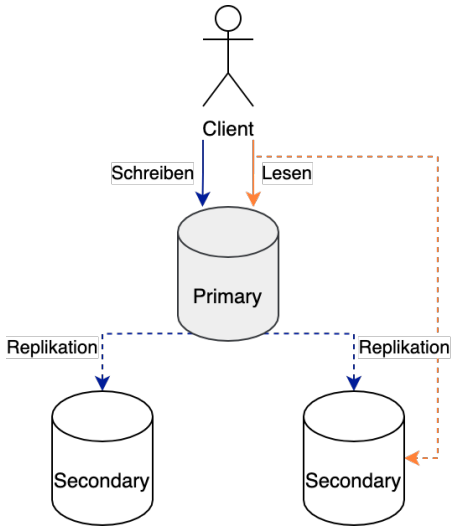

Abbildung 2: Redis Sentinel – Architektur

Schreibanfragen werden ausschließlich vom Primary verarbeitet und von diesem auf die Secondaries gespiegelt. Leseanfragen werden in der Regel auch zunächst vom Primary beantwortet, können jedoch auch von den Secondaries beantwortet werden.

Tritt nun eine Partition auf, wird durch den Sentinel-Service in dem Teil des Netzwerkes, welcher die Mehrheit der Knoten hält, ein neuer Primary gewählt:

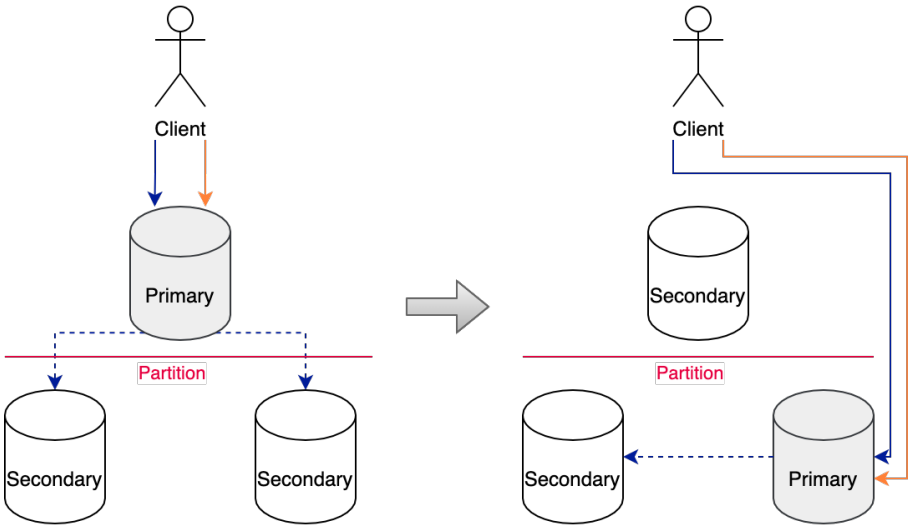

Abbildung 3: Redis – Primary Auswahl

Ist die Partition wieder aufgehoben, werden die Daten vom neuen Primary auf die Knoten des anderen Netzwerkteils gespiegelt. So bleibt das System auch im Falle einer Partition weiter funktionsfähig und ist somit partitionstolerant [16]–[18]:

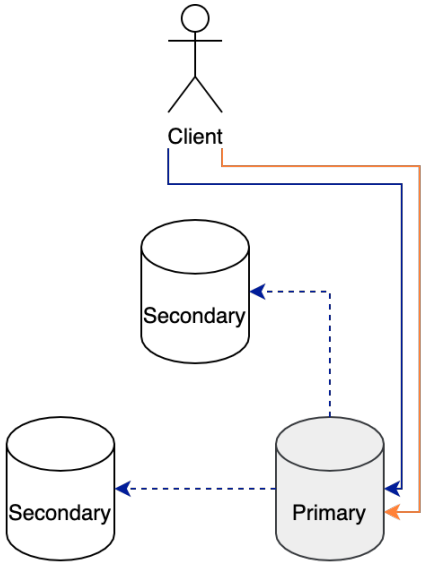

Abbildung 4: Redis – Recovery

Schlussbemerkung

Gilbert und Lynch haben auf Grundlage der hier beschriebenen Definitionen einen formalen Beweis ausgeführt, welcher zeigt, dass ein System nur zwei der drei zuvor genannten Eigenschaften erfüllen kann [5]. Dies hat in den vergangenen Jahren einen großen Einfluss auf das Design von verschiedenen Softwaresystemen und -architekturen ausgeübt, jedoch auch für viele Diskussionen gesorgt [6], [19], [20]. Trotz des formalen Beweises hat sich in der Praxis gezeigt, dass Entwickler in der Lage waren, die Einschränkungen des CAP-Theorems zu umgehen [20], [21]. Es gilt zu hinterfragen, ob die Formalisierung von Brewers Aussage durch Gilbert und Lynch dem entspricht, was Brewer statuieren wollte.

Eine andere Auslegung der ursprünglichen Definitionen ermöglicht es, Systeme zu beschreiben, die alle drei Eigenschaften des Theorems erfüllen können, wenn auch nicht simultan. Dabei spielt es z. B. eine Rolle, wie sich die Zustände eines Systems über einen gewissen Zeitraum verändern. Darüber hinaus besteht die Frage, ob es Sinn ergibt, die Eigenschaften des Theorems, wie Gilbert und Lynch dies getan haben, qualitativ zu messen oder aber dies quantitativ zu tun [6], [19].

Diese und weitere Aspekte sind in diversen Kritiken und Reviews des CAP-Theorems aufgegriffen worden, die im Folgenden erörtert werden.

3.3 Kritiken am CAP-Theorem

Definitionen der Eigenschaften

Wie in [13] gezeigt wird, hat Brewer bei seinen ursprünglichen Ausführungen viele Punkte offen gelassen, welche dann von Gilbert und Lynch [5] für ihren formalen Beweis weiter definiert wurden. Doch es stellt sich die Frage, ob diese Definitionen auch sinnvoll ausgeführt worden sind. Martin Kleppmann führt in seinem Beitrag „A Critique of the CAP Theorem“ [6] diverse Kritikpunkte zu Gilbert und Lynchs Definitionen der CAP-Eigenschaften an:

Verfügbarkeit

Die Definition von Gilbert und Lynch sieht vor, dass jeder Knoten des Systems eine Antwort liefern können muss, um als verfügbar zu gelten. Es würde also ein System, in welchem einige Knoten ausgewählte Operationen bewusst nicht zulassen, als nicht verfügbar eingestuft werden [6]. So wären im Sinne der Definition von Gilbert und Lynch die nur lesbaren Redis-Secondaries aus o. g. Beispiel nicht verfügbar, obwohl sie ihre spezifizierte Funktionalität liefern können.

Konsistenz

Die Definition der Konsistenz nach Gilbert und Lynch wird oftmals als starke Konsistenz beschrieben [6]: Erst wenn eine schreibende Operation vollständig auf allen Knoten eines Systems durchgeführt wurde, dürfen diese Knoten ausgelesen werden [5].

Dem entgegen steht die eventuelle Konsistenz. Dieser Ausdruck besagt, dass die Daten von Spiegelungen jederzeit ausgelesen werden dürfen, unter Berücksichtigung der Tatsache, dass die Daten ggf. noch nicht aktuell sind [22]. Auch diese Form der Konsistenz ist für bestimmte Anwendung nützlich, wird jedoch von Gilbert und Lynch nicht berücksichtigt [6].

Partitionstoleranz

Eine Partition tritt nach Gilbert und Lynchs Definition bereits dann auf, wenn einzelne Nachrichten eines Knotens den Zielknoten nicht erreichen können, also im Netzwerk verloren gehen. Die Partitionstoleranz ist dann gegeben, wenn eine willkürliche Anzahl von Nachrichten verloren gehen kann und das System dennoch funktionsfähig bleibt [5]. Die Definition lässt an dieser Stelle jedoch offen, ob es sich hierbei um eine abzählbare Anzahl an Nachrichten handelt oder nicht. Zusätzlich wird nicht definiert, ob eine Partition von begrenzter Dauer oder von unendlicher Dauer sein kann.

Qualitative Bewertung der Eigenschaften

Die Eigenschaften des CAP-Theorems werden von Gilbert und Lynch als rein qualitative Eigenschaften definiert. Sie sind in diesem Sinne binär, werden also von einem System erfüllt oder nicht erfüllt. Jedoch bestehen auch diverse Ansätze zur quantitativen Messung der Eigenschaften. Brewer selbst hat zwölf Jahre nach seiner ursprünglichen Ausführung [13] erläutert, dass die Eigenschaften des CAP-Theorems flexibler interpretiert werden sollten, als von Gilbert und Lynch ausgeführt [19]. Martin Kleppmann hat in seiner Kritik ebenfalls eine mögliche quantitative Bewertung der Eigenschaften Verfügbarkeit und Konsistenz beschrieben [6].

Verfügbarkeit

Nach Gilbert und Lynch ist es unbedeutend, wie lange es dauert, bis eine Antwort empfangen wird. Es ist entscheidend, ob eine Antwort empfangen wird [6]. Dies entspricht nur bedingt dem intuitiven Verständnis der Verfügbarkeit aus der Sicht eines Nutzers. Dieser möchte innerhalb einer angemessenen Zeit eine Antwort auf eine Anfrage erhalten.

Das Verständnis von „angemessener Zeit“ mag hier im Auge des Betrachters und des Anwendungsfalles liegen, es ist jedoch möglich, die Antwortlatenz, also die Dauer bis ein System antwortet, zu messen. Anhand dieser Latenz ist wiederum die Verfügbarkeit quantitativ bewertbar [6].

Konsistenz

Auch die Konsistenz wird von Gilbert und Lynch als binäre Eigenschaft definiert. Es ist jedoch möglich, die Konsistenz als stochastisches Maß quantitativ zu messen [6]. Ein Beispiel hierfür ist unter [23] angegeben.

Neben dem komplexeren Beispiel aus [23] ist es auch möglich, simplere Metriken zur quantitativen Messung der Konsistenz zu verwenden. Brewer und Fox schlagen vor, die Vollständigkeit einer Antwort auf eine Anfrage zu bestimmen [24]. Unter Kapitel 4 wird zusätzlich ein eigener Vorschlag zur quantitativen Messung der Konsistenz erarbeitet.

Partitionstoleranz

In den hier genutzten Quellen ist zwar keine Möglichkeit zur quantitativen Messung der Partitionstoleranz aufgeführt worden, allerdings wird in Kapitel 4 auch hierzu eine Metrik vorgeschlagen.

Latenzbasierte Modelle

Das CAP-Theorem nach Gilbert und Lynch ignoriert, wie im vorherigen Kritikpunkt bereits zum Teil beschrieben, jegliche Form der Latenz [19]. Vor allem die Definition der Verfügbarkeit ist jedoch sehr abhängig von einer Latenz. Sofern ein System in einer beliebigen Zeit antwortet, gilt die Verfügbarkeit als gegeben [5]. Somit ist nicht ausgeschlossen, dass sich die Dauer einer Antwort solange verzögert, bis z. B. eine durch Partition verursachte Inkonsistenz beseitigt wurde. Gilbert und Lynch schließen dieses Verhalten in ihrem Beweis des CAP-Theorems aus, indem die Partitionen in ihrem Modell von unendlicher Dauer sind. Eine Teilung des Netzwerks könnte also nicht wieder aufgehoben werden [6].

In der Praxis ist dies jedoch anders: Eine Partition kann bereits dann auftreten, wenn einzelne Datenpakete im Netzwerk verloren gehen [5], aber auch dann, wenn physische Netzwerkverbindungen ausfallen. Der Verlust einzelner Pakete kann zufällig sein [19], sodass bereits eine erneute Übertragung des verlorenen Paketes Abhilfe schafft. Physische Netzwerkverbindungen können mit vergleichsweise größerem Aufwand repariert werden.

Dieses Beispiel soll verdeutlichen, dass eine Partition nicht zwingend permanent sein muss, sondern auch von begrenzter Dauer sein kann. Dies sorgt wiederum für Latenz in der Kommunikation zwischen den Knoten eines Systems. Je nach Art der Partition fällt diese Latenz unterschiedlich groß aus und wirkt sich unter Umständen auch auf die Antwortzeiten des Systems, also die Verfügbarkeit, aus.

Brewer beschreibt, dass in der Handhabung der partitionsbedingten Latenz der Kern der CAP-Eigenschaften liegt [19]. Ein System muss anhand dieser Latenz entscheiden, ob eine Operation auf Kosten der Verfügbarkeit abgebrochen oder aber auf Kosten der Konsistenz fortgeführt wird. Es muss ein entsprechender Mechanismus vorhanden sein, welcher diese Entscheidung für die jeweilige Operation trifft. Hierzu gehören laut Brewer drei Schritte:

1. Das System bzw. einzelne Knoten des Systems **erkennen** eine Partition.
2. Das System bzw. einzelne Knoten des Systems werden in einen **Partitionsmodus** versetzt, welcher ggf. bestimmte Operationen einschränkt.
3. Nach Aufhebung der Partition wird eine **Wiederherstellung** des Systems bzw. der Knoten ausgelöst.

Beispiel: Partitionsbehandlung

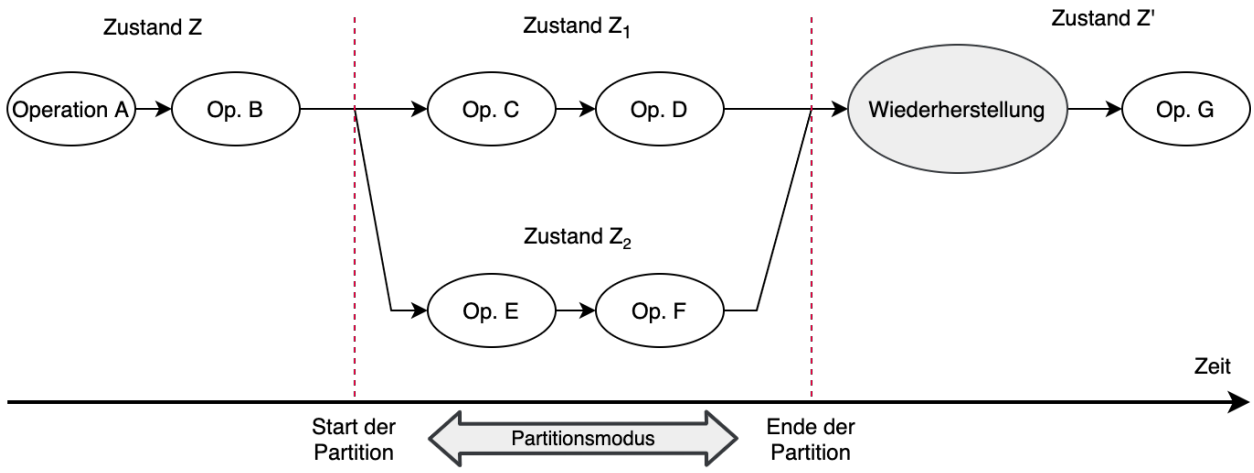

Abbildung 5: Beispiel – Partitionsbehandlung

Das Beispielsystem startet in einem konsistenten Zustand Z, bis die Partition entsteht. Beide Seiten der Partition werden in den Partitionsmodus versetzt, um weiter verfügbar zu bleiben. Dadurch entsteht eine temporäre Inkonsistenz. Mit Ende der Partition werden die beiden Seiten des Systems wieder zu einem konsistenten Zustand Z' zusammengeführt.

Dieses Beispiel zeigt, dass ein System durchaus in der Lage sein kann, alle drei CAP-Eigenschaften zu erfüllen, wenn auch nicht zum selben Zeitpunkt. Im Partitionsmodus ist zu entscheiden, ob die Operationen eingeschränkt werden, was die Wiederherstellung der Konsistenz erleichtert, aber die Verfügbarkeit einschränkt. Oder aber, ob alle Operationen zugelassen werden, was wiederum die Verfügbarkeit begünstigt und die Wiederherstellung der Konsistenz erschwert [19].

Zusammenfassung

Durch die zuvor vorgestellte Kritik stellt sich heraus, dass Gilbert und Lynchs CAP-Theorem zwar als anerkanntes Mittel für Entscheidungen der Softwarearchitektur gilt, jedoch für den Entwurf von Softwaresystemen zu ggf. unflexibel ist. Zum Zweck des formalen Beweises haben Gilbert und Lynch die CAP-Eigenschaften qualitativ definiert. In der Praxis hingegen ist es sinnvoller, die Eigenschaften als quantitative Maße zu interpretieren und an konkreten Systemen zu messen.

So kann basierend auf den CAP-Eigenschaften ein Modell erstellt werden, mit dessen Hilfe sich Softwaresysteme charakterisieren und untereinander vergleichen lassen. Dieses Modell wird mit dem Ziel definiert, für die praktische Anwendung von Nutzen zu sein.

4. Die CAP-Charakteristik

4.1 Die Eigenschaften der CAP-Charakteristik

Da die Kritikpunkte am CAP-Theorem von Gilbert und Lynch [5] nun deutlich sind, soll ein Modell entworfen werden, welches geeigneter ist, um Softwaresysteme in der Praxis einzuordnen. Das Modell wird folgend als CAP-Charakteristik bezeichnet. Es hebt sich im Vergleich zu Gilbert und Lynch vor allem dadurch ab, dass die CAP-Eigenschaften, wie von Brewer empfohlen [19], als quantitative Eigenschaften definiert werden:

Verfügbarkeit

Ähnlich wie bei Gilbert und Lynch zeichnet sich die Verfügbarkeit in der CAP-Charakteristik dadurch aus, dass ein System in einer bestimmten Zeit eine Antwort liefert. Es wird dabei nicht bewertet, ob eine Anfrage an das System überhaupt eine Antwort liefert, sondern wie schnell dies geschieht. Folglich wird zunächst die absolute Dauer gemessen, welche ein System benötigt, um eine Anfrage zu beantworten. Diese Dauer wird als absolute Antwortlatenz $a_{abs} \in \mathbb{R}^{\geq 0}$ bezeichnet.

An dieser Stelle wird noch nicht festgelegt, in welchen Größenordnungen sich die Antwortlatenz eines Systems bewegen darf, damit dieses als verfügbar gilt. Das hat den Hintergrund, dass Systeme abhängig von ihrer Funktion unterschiedlich lange benötigen, um zu antworten. Stattdessen legt der Anwendungskontext eine obere Grenze, $a_{max} \in \mathbb{R}^{>0}$ fest, nach welcher die Antwortlatenz eines Systems als akzeptabel gilt. Eine solche Grenze kann für mehrere Systeme mit ähnlicher Funktion gemeinsam festgelegt werden, sodass die Systeme untereinander verglichen werden können.

Die Antwortlatenz muss der Einheitlichkeit halber direkt an dem von einem System vorgegebenen Endpunkt, z. B. einem Reverse-Proxy [25], abgenommen werden, um Messungenauigkeiten durch beispielsweise Packet Loss [26] in einem Netzwerk zu vermeiden.

Die Verfügbarkeit wird gemessen, indem die absolute Antwortlatenz a_{abs} relativ zur oberen Grenze a_{max} dargestellt wird:

$$a = 1 - \frac{\min(a_{abs}, a_{max})}{a_{max}}$$

Formel 1: Maß der Verfügbarkeit

Konsistenz

Zur Messung der Konsistenz wird ein Maß verwendet, welches der starken Konsistenz nahe ist. Hierzu sei folgendes Modell gegeben:

Ein System S, welches eine Datenquelle darstellt, besitzt $n \in \mathbb{N}$ Knoten, bezeichnet mit K_i, $i = 1, \dots, n$. Ein Knoten speichert die Datensätze $d_{i,k}$. Der Index $k \in K$ stellt hierbei einen eindeutigen Identen eines Datensatzes dar, wobei K die Menge aller Schlüssel des Systems S ist. Die Funktion $id: \{ x \in \mathbb{N} \mid 1 \leq x \leq |K| \} \to K$ nummeriert die Schlüssel in K durch, sodass diese mit einer ganzen Zahl referenziert werden können.

Die Menge der Schlüssel K wird in diesem Kontext nicht weiter definiert, da sich die Schlüssel in der Praxis auf verschiedene Arten ausdrücken lassen können. Mögliche Beispiele hierfür sind zusammengesetzte Primärschlüssel in relationalen Datenbanken [27] oder Keys in Key-Value-basierten Datenbanken [28]. Die formale Definition von $d_{i,k}$ ist ebenfalls wieder abhängig vom Anwendungsfall: In relationalen Datenbanken kann z. B. $d_{i,k}$ einem Tupel einer Relation entsprechen [29], hingegen in einer Key-Value-basierten Datenbank dem Wert zu einem Schlüssel [28].

Die Konsistenz $c \in [0; 1]$ eines Systems wird nun bestimmt, indem die Datensätze $d_{i,k}$ zwischen den Knoten zu einem gemeinsamen Zeitpunkt untereinander verglichen werden. Es werden im Folgenden drei Fälle betrachtet mit der Vergleichsfunktion

$$cmp(d_1, d_2) = \begin{cases} 0, & \text{wenn } \mathrm{d}_1 \text{ und } \mathrm{d}_2 \text{ gleich} \\ 1, & \text{wenn } \mathrm{d}_1 \text{und } \mathrm{d}_2 \text{ ungleich} \end{cases}.$$

Die Gleichheit von zwei Datensätzen muss wiederum für den konkreten Anwendungsfall genauer definiert werden.

Für $n = 1$ ist die Konsistenz immer gegeben, da es keine Möglichkeit eines Vergleiches gibt, wenn nur ein Knoten existiert. Somit ist in diesem trivialen Fall $c = 1$.

Für $n = 2$ gilt, dass alle Datensätze von Knoten K_1 mit den schlüsselgleichen Datensätzen von Knoten K_2 verglichen werden und hierbei der Mittelwert der Vergleichsergebnisse gebildet wird:

$$c = \frac{1}{|K|} \sum_{j=1}^{|K|} cmp\left(d_{1,id(j)}, d_{2,id(j)}\right)$$

Formel 2: Maß der Konsistenz für n = 2

Für $n > 2$ gilt, dass jeder Knoten mit jedem anderen Knoten, außer mit sich selbst, verglichen werden muss, um die Konsistenz des gesamten Systems zu berechnen. So wird

$$c_{a,b} = \frac{1}{|K|} \sum_{j=1}^{|K|} cmp\left(d_{a,id(j)}, d_{b,id(j)}\right)$$

Formel 3: Maß der Konsistenz für n > 2

definiert als Konsistenz zwischen zwei Knoten eines Systems. Die Konsistenz eines gesamten Systems wird gebildet aus dem Mittelwert der Ergebnisse von $c_{a,b}$ über alle möglichen Vergleichstupel $(a, b) \in \{(x, y) \in \mathbb{N}^2 \mid x < y \land x \leq n \land y \leq n\}$, die eine Kombination von zwei Knoten des Systems repräsentieren.

Beispiel: Menge der Vergleichstupel

Für $n = 4$ kann die Menge der Vergleichstupel wie folgt dargestellt werden:

$$\{(x, y) \in \mathbb{N}^2 \mid x < y \land x \leq 4 \land y \leq 4\}$$

$\downarrow x \quad y \rightarrow$	1	2	3	4	⋯
1	$(1,1)$	$(1,2)$	$(1,3)$	$(1,4)$	⋯
2	$(2,1)$	$(2,2)$	$(2,3)$	$(2,4)$	⋯
3	$(3,1)$	$(3,2)$	$(3,3)$	$(3,4)$	⋯
4	$(4,1)$	$(4,2)$	$(4,3)$	$(4,4)$	⋯
⋮	⋮	⋮	⋮	⋮	⋱

n

Tabelle 1: Vergleichstupel für n = 4

Beispiel: Messung der Konsistenz

Das unten abgebildete System stellt einen Key-Value-Speicher dar. Zu jedem numerischen Schlüssel wird ein alphanumerischer Wert abgelegt. Die Abbildung stellt eine Momentaufnahme aller Knoten zu einem festen Zeitpunkt dar. Folgende Werte lassen sich aus der Abbildung entnehmen:

Knoten K_1, K_2, K_3, Anzahl der Knoten $n = 3$, Schlüsselmenge $K = \{1, 2, 3\}$,

Datensätze $d_{1,1} = \text{"A"}, \ldots, d_{3,3} = \text{"F"}$

Die Funktion $id: \{ x \in \mathbb{N} \mid 1 \leq x \leq |K| \} \to K$ sei definiert als $id(x) = x$. Durch den gegebenen Kontext gilt die Gleichheit der Funktion $cmp(d_1, d_2)$ als erfüllt, wenn die zu einem Schlüssel gelieferten Werte und die Schlüssel der Werte selbst übereinstimmen. Die Werte gelten dabei als gleich, sofern sie den gleichen Buchstaben des lateinischen Alphabetes darstellen. Die Schlüssel gelten als gleich, sofern sie den gleichen Zahlenwert haben.

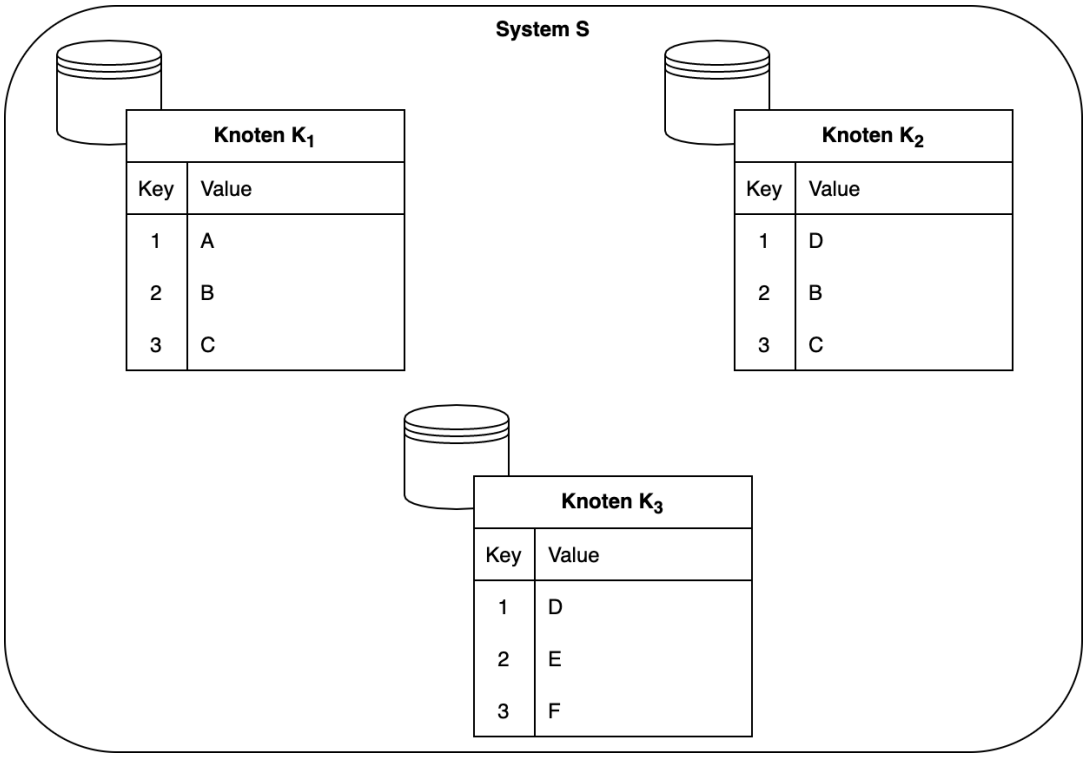

Abbildung 6: Key-Value-System mit drei Knoten

Da $n = 3$ ist, ergibt sich $\{(1,2), (1,3), (2,3)\}$ als Menge der möglichen Vergleichstupel. Exemplarisch wird die Berechnung von $c_{1,2}$ dargestellt:

$$
\begin{aligned}
c_{1,2} &= \frac{1}{|K|} \sum_{j=1}^{|K|} cmp\left(d_{1,id(j)}, d_{2,id(j)}\right) \\
&= \frac{1}{3}\left(cmp\left(d_{1,1}, d_{2,1}\right) + cmp\left(d_{1,2}, d_{2,2}\right) + cmp\left(d_{1,3}, d_{2,3}\right)\right) \\
&= \frac{1}{3}\left(cmp(\text{"A"}, \text{"D"}) + cmp(\text{"B"}, \text{"B"}) + cmp(\text{"C"}, \text{"C"})\right) \\
&= \frac{1}{3}(0 + 1 + 1) \quad = \quad \frac{2}{3}
\end{aligned}
$$

Für die übrigen Vergleichstupel ergeben sich $c_{1,3} = 0$ und $c_{2,3} = \frac{1}{3}$. Der Mittelwert und somit das Maß der Konsistenz beträgt in diesem Beispiel $c = \frac{1}{3} \cdot \left(\frac{2}{3} + 0 + \frac{1}{3}\right) = \frac{1}{3}$.

Partitionstoleranz

Das wesentliche Merkmal der Partitionstoleranz ist, dass ein Knoten eines Systems, unabhängig von den anderen Knoten, konsistent die volle Funktionalität des Systems bereitstellen kann. Im Falle der Partition muss jeder Knoten eigenständig agieren können, um weiterhin verfügbar zu sein. Bei Wiederherstellung der Verbindung zwischen den Knoten müssen diese untereinander regeln, wie mögliche Inkonsistenzen, die während der Partition entstanden sind, beseitigt werden.

Das hier beschriebene Verhalten orientiert sich am o. g. Beispiel der Partitionsbehandlung. Es wird also angenommen, dass ein System einen Algorithmus implementiert, der es ermöglicht, dass sich die Knoten untereinander synchronisieren, um die Konsistenz unter den Knoten (wieder-)herzustellen [19].

Die Partitionstoleranz kann gemessen werden, indem zum einen die Eigenständigkeit eines Knotens und zum anderen die Integrität des Systems bewertet wird. Ein mögliches Maß hierzu stellt die Dauer dar, welche ein System benötigt, um die Daten zwischen den Knoten zu synchronisieren. Die Synchronisation bezieht sich dabei sowohl auf das bloße Replizieren von Daten zwischen Knoten, als auch die Beseitigung möglicher Inkonsistenzen nach einer Partition.

Bei Systemen, die einen solchen Synchronisations-Algorithmus implementieren, wird also zunächst die Dauer $p_{abs} \in \mathbb{R}^{\geq 0}$ gemessen, welche der Algorithmus benötigt, um alle Knoten in den gleichen Zustand zu versetzen. Systeme, die keine Synchronisation nutzen, werden als vollkommen partitionstolerant eingestuft, sofern es der Integrität des Systems genügt, dass alle Knoten inkonsistent sind. Setzt die Integrität des Systems jedoch eine Synchronisation voraus, die nicht implementiert worden ist, wird das System als vollkommen intolerant gegenüber Partitionen angesehen:

Falls ein System nur **einen Knoten** besitzt oder für die Integrität **keine Synchronisation benötigt**, wird $p_{abs} = 0$ gesetzt.

Für Systeme **mit Synchronisation** und **mehr als einem Knoten** berechnet sich p_{abs} als die mittlere Dauer, die ein Synchronisations-Algorithmus benötigt, um alle Knoten von einem Zustand Z in einen Zustand Z' zu überführen bzw. inkonsistente Zustände Z_i zu einem gemeinsamen Zustand Z' zu vereinen. Der Index $i = 1, \dots, n$ adressiert hierbei die Zustände, wobei $n \in \mathbb{N}$ die Anzahl der sich unterscheidenden Zustände angibt.

Sofern ein System **keine Synchronisation** implementiert, diese aber **für die Integrität benötigt** wird, so ist $p_{abs} = p_{max}$.

Analog zur Verfügbarkeit ist je nach Anwendungskontext eine obere Grenze, $p_{max} \in \mathbb{R}^{>0}$, festzulegen, die angibt, welche Ausführungszeit eines Synchronisations-Algorithmus tolerierbar ist. Das Maß für die Partitionstoleranz ergibt sich wiederum, wenn p_{abs} in Relation zu p_{max} gesetzt wird:

$$p = 1 - \frac{\min(p_{abs}, p_{max})}{p_{max}}$$

Formel 4: Maß der Partitionstoleranz

Erneut ist es wichtig zu beachten, dass die Integrität eines Systems sehr anwendungsspezifisch anzusehen ist. So ist es für die Integrität einer Datenbank, welche dem ACID-Prinzip folgt, erforderlich, dass alle Knoten möglichst immer konsistent sind [30]. Es kann jedoch auch Systeme geben, welche ihre Integrität auch dann wahren, wenn alle Knoten vollkommen inkonsistent sind.

Beispiel: Messung der Partitionstoleranz

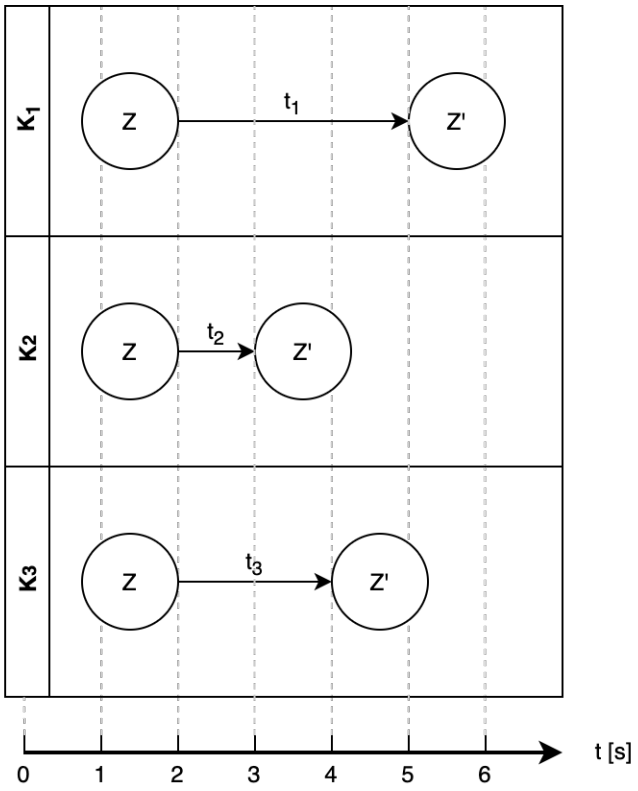

Abbildung 7: Beispiel – Sync.-Algo. 1

Die dargestellte Abbildung stellt ein aus drei Knoten K_1, K_2 und K_3 bestehendes System dar. Alle drei Knoten starten im gemeinsamen Initialzustand Z. Das System implementiert einen Synchronisations-Algorithmus, der die Zustände der Knoten im Einklang hält. Für den Zustandswechsel ergeben sich die Zeiten $t_1 = 3s,\ t_2 = 1s,\ t_3 = 2s$. Der Mittelwert beträgt in diesem Fall $p_{abs} = \frac{1}{3} \cdot (3s + 1s + 2s) = 2s$.

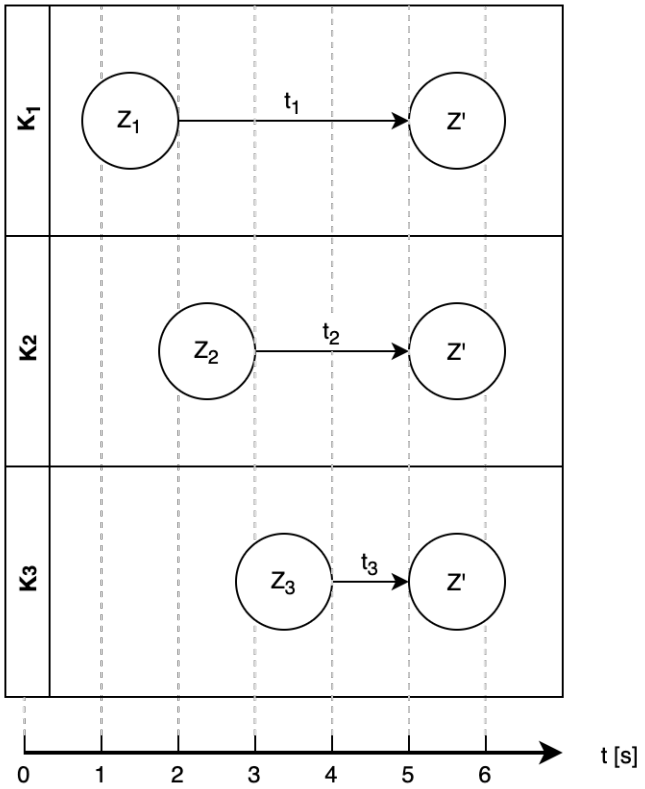

Abbildung 8: Beispiel – Sync.-Algo. 2

Analog zum vorherigen Beispiel ist es auch möglich, dass die Knoten des Systems zu Beginn der Synchronisation unterschiedliche (inkonsistente) Zustände Z_1, Z_2 und Z_3 haben, welche in einen gemeinsamen Zustand überführt werden. Für dieses Beispiel mit $t_1 = 3s,\ t_2 = 2s$ und $t_3 = 1s$ beträgt der Mittelwert ebenfalls $p_{abs} = \frac{1}{3} \cdot (3s + 2s + 1s) = 2s$.

4.2 Nutzung der Maße

Da nun die Eigenschaften der CAP-Charakteristik definiert worden sind, wird nun in diesem Abschnitt diskutiert, wie Softwaresysteme anhand der CAP-Charakteristik analysiert werden können. Hierzu werden die Eigenschaften c, a und p auf verschiedene Weisen dargestellt.

Darstellung als Graph

Die Eigenschaften der CAP-Charakteristik bilden eine Momentaufnahme eines Systems ab. Die Betrachtung latenzbasierter Modelle hat jedoch gezeigt, dass sich die Verfügbarkeit, Konsistenz und Partitionstoleranz eines Systems über einen Zeitraum verändern können. Somit besteht die Annahme, dass die Werte von c, a und p je nach Zeitpunkt der Messung variieren.

Daher werden c, a und p als Funktionen in Abhängigkeit von mehreren Zeitpunkten t_i, $i = 1, \dots, n$ mit $n \in \mathbb{N}$, beschrieben und mit $c(t), a(t)$ und $p(t)$ bezeichnet. Die Eigenschaften der CAP-Charakteristik werden also über einen bestimmten Zeitraum an einem System gemessen, um Messreihen zu erhalten. Anhand der Messreihen können Graphen aufgestellt werden, die das Verhalten eines Systems darstellen. Die Graphen können mathematisch darüber hinaus analysiert werden, z. B. darauf, ob sich Trends in der Entwicklung der Eigenschaften zeigen oder wie stark die Messwerte über den Zeitraum streuen.

Beispiel: Fiktive Darstellung von c(t)

Das folgende Beispiel stellt eine fiktive Messreihe der Konsistenz eines Systems dar.

t [s]	0	1	2	3	4	5
c [a.u.]	0,7	0,65	0,8	0,75	0,8	0,7

Tabelle 2: Fiktive Messreihe für c(t)

Aus dieser Messreihe ergibt sich das folgende Diagramm:

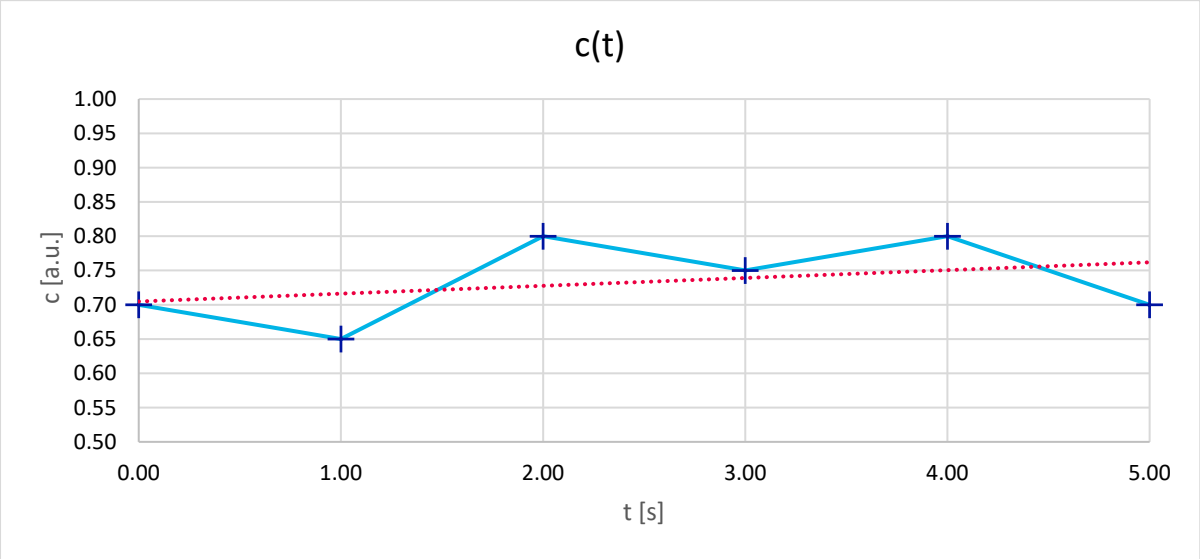

Diagramm 1: Darstellung von c(t)

Darstellung als Dreieck

Eine weitere Art der Darstellung bietet das Anordnen der Eigenschaften in einem Dreieck [20]. Es wäre ungünstig, hierzu a_{abs} und p_{abs} zu verwenden, da diese mit $\mathbb{R}^{\geq 0}$ als Wertebereich im Vergleich zu $c \in [0;1]$ über einen großen Zahlenbereich streuen können. Für die gewählte Darstellung ist es erforderlich, dass sich die Werte aller Eigenschaften sich im gleichen Zahlenbereich bewegen.

Bei genauerer Betrachtung fällt auf, dass $c(t), a(t)$ und $p(t)$ alle den Wertebereich $[0;1]$ aufweisen. Dies kann auch bewiesen werden:

Beweis für c

$$c = \frac{1}{|K|}\sum_{j=1}^{|K|} f\left(d_{1,id(j)}, d_{2,id(j)}\right)$$

$$\Leftrightarrow\ c \cdot |K| = \sum_{j=1}^{|K|} f\left(d_{1,id(j)}, d_{2,id(j)}\right)$$

Wegen $f(d_1, d_2) \in \{0,1\}$: $\quad c \cdot |K| \leq |K| \quad \Leftrightarrow \quad c \geq 1$

$$\wedge\ \ c \cdot |K| \geq\ 0 \quad \Leftrightarrow \quad c \leq 0$$ **q.e.d.**

($c_{a,b}$ analog)

Beweis für a, p

$$a = 1 - \frac{\min(a_{abs},\ a_{max})}{a_{max}}$$

I. Falls $a_{abs} > a_{max}$:

$$a = 1 - \frac{a_{max}}{a_{max}} = 1 - 1 = 0$$

II. Falls $a_{abs} \leq a_{max}$:

$$a = 1 - \frac{a_{abs}}{a_{max}}$$

Wegen $a_{abs} \in \mathbb{R}^{\geq 0}$: $\quad a \leq 1 - \frac{0}{a_{max}} = 1$

Und wegen $a_{abs} = a_{max}$: $\quad a\ \geq 1 - \frac{a_{max}}{a_{max}} = 0$

Aus I. und II. folgt: $\quad 0 \leq a \leq 1$ **q.e.d.**

(p analog)

Die gewählten Definitionen der Eigenschaften bieten mehrere Vorteile: Das Festlegen von a_{max} und p_{max} erlaubt es, eine obere Grenze zu deklarieren, ab welcher ein System als nicht mehr verfügbar bzw. partitionstolerant gilt. Dadurch wird ein starker Bezug zu einem möglichen Anwendungskontext hergestellt.

Zudem kann mit der oberen Grenze das Ausmaß festgelegt werden, in welchem a und p skalieren. Bei kleinerem a_{max} bzw. p_{max} ist a bzw. p empfindlicher gegen eine geringe Veränderung von a_{abs} bzw. p_{abs}. Bei größerem a_{max} bzw. p_{max} müssen a_{abs} bzw. p_{abs} sich stärker verändern, um a und p signifikant zu beeinflussen.

Ein weiterer Vorteil ist, dass sich die drei Eigenschaften auf Grund ihres Zahlenbereiches in einer homogenen Darstellung in Relation zueinander setzen lassen. Die Werte der Eigenschaften beschreiben dabei den Abstand von Eckpunkten eines Dreiecks zum Mittelpunkt. So kann für jedes System eine Art Fingerabdruck erzeugt werden.

Diese Art der Darstellung erlaubt nur die Repräsentation eines Wertes je Eigenschaft. Die Maße werden analog zur Darstellung als Graph über mehrere Zeitpunkte gemessen. Um die Schwankungen der Werte über den Zeitraum zu egalisieren, werden Mittelwerte $\bar{c}, \bar{a}$ und $\bar{p} \in [0; 1]$ berechnet. Diese Werte werden dann genutzt, um die Eckpunkte des Dreiecks zu modellieren.

Beispiel: Darstellung als Dreieck

Die folgende Tabelle stellt fiktive Messreihen für c, a und p bereit:

t [s]	0	1	2	3	4	5
$c(t)$ [a.u.]	0,7	0,65	0,8	0,75	0,8	0,7
$a(t)$ [a.u.]	1,0	0,95	0,9	0,98	0,91	0,96
$p(t)$ [a.u.]	0,5	0,56	0,57	0,6	0,4	0,5

Tabelle 3: Fiktive Messreihe für die Darstellung als Dreieck

Aus den Messreihen ergeben sich die Mittelwerte, welche die Eckpunkte des Dreiecks bestimmen:

$$\bar{c} \approx 0{,}73 \qquad \bar{a} = 0{,}95 \qquad \bar{p} \approx 0{,}52$$

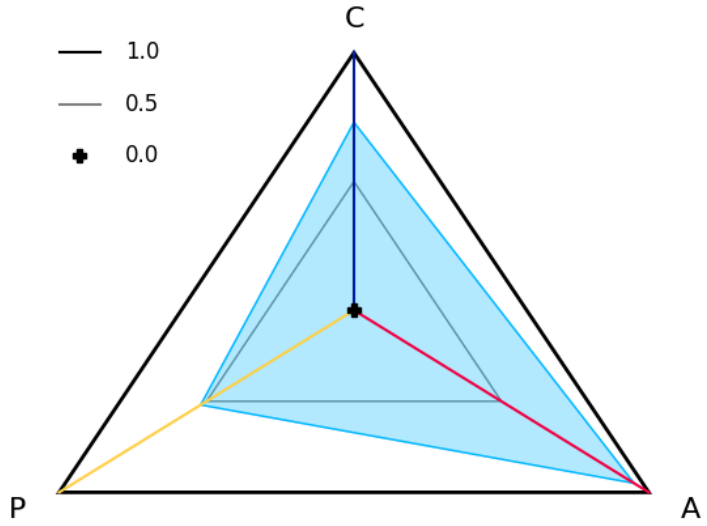

Abbildung 9: Beispieldarstellung CAP-Dreieck

Diese Abbildung ist von einem Python-Programm mit Hilfe der Bibliothek Matplotlib [31] erzeugt worden. Das zugehörige Programm ist in Anhang A hinterlegt.

4.3 Zusammenfassung

Mit Hilfe der als quantitativ definierten CAP-Eigenschaften ist es möglich, Softwaresysteme bzw. deren Architekturen flexibel im Kontext der Konsistenz, Verfügbarkeit und Partitionstoleranz einzuordnen. Dabei gelten diese Eigenschaften nicht als erfüllbare Kriterien, sondern sie können unterschiedlich stark ausgeprägt sein.

Durch die unterschiedlichen Darstellungsweisen werden die CAP-Eigenschaften hierbei aus verschiedenen Blickwinkeln betrachtet. Die Observierung der Eigenschaften über einen Zeitraum hinweg ermöglicht es, Untersuchungen zu der Entwicklung der Eigenschaften anzustellen. Mittelwerte über Zeitserien bieten eine gute Approximation, um Aussagen über die allgemeine Konsistenz, Verfügbarkeit oder Partitionstoleranz eines Systems zu treffen.

Dadurch, dass die Eigenschaften der CAP-Charakteristik anwendungsspezifische Parameter, wie z. B. a_{max} bzw. p_{max} berücksichtigen, lassen sich die Definitionen für den jeweiligen Kontext anpassen. So kann die CAP-Charakteristik als unterstützendes Mittel zur Planung von Softwarearchitekturen genutzt werden. Die CAP-Charakteristik kann darüber hinaus die Vergleichbarkeit zwischen Softwaresystemen herstellen, die einem ähnlichen Zweck dienen.

Im folgenden Kapitel soll die CAP-Charakteristik genutzt werden, um verschiedene Datenbanksysteme miteinander zu vergleichen.

5. Anwendung der CAP-Charakteristik

Um zu prüfen, ob die CAP-Charakteristik in der hier beschriebenen Form sinnvoll verwendet werden kann, wird der folgende Versuch angestellt. Hierbei wird die CAP-Charakteristik für die Key-Value-basierte Datenbank Redis [32] und die relationale Datenbank PostgreSQL [33] aufgestellt, um diese miteinander zu vergleichen und deren Architekturen zu bewerten.

Für beide Systeme wird in diesem Versuch eine containerbasierte Architektur verwendet. Eine Voraussetzung für diese Architekturen sind die Container-Plattform Docker [34] und die Orchestrierungs-Plattform Kubernetes [35]. Um ein tieferes Verständnis für die verwendeten Architekturen zu gewinnen, sollen Docker und Kubernetes zunächst vorgestellt werden. Darüber hinaus wird Helm als Werkzeug zur Bereitstellung von Systemen in Kubernetes eingeführt.

5.1 Exkurs: Docker, Kubernetes und Helm

Docker

Die folgenden Ausführungen orientieren sich an dem einleitenden Kapitel aus [36].

Die Container-Plattform Docker ermöglicht es Softwaresysteme in kompakte Einheiten zu verpacken, die Container genannt werden. Als Vorlage für einen Container wird ein sogenanntes Image erzeugt, das alle Abhängigkeiten, welche zur Ausführung eines Systems gebraucht werden, enthält. Hierzu gehören unter anderem der Programmcode bzw. ausführbare Programmdateien, Laufzeitumgebungen oder auch weitere Tools und Bibliotheken, von denen ein System abhängig ist. Container können als kleine virtuelle Maschinen betrachtet werden, deren Betriebssystem durch das Image vorgegeben wird. Die Definition der Images erfolgt über Dockerfiles. Ein Beispiel einer Dockerfile steht in Anhang B bereit.

Docker-Container werden ausgeführt, indem die sogenannte Docker-Engine Container aus den Images erzeugt. Es können auch mehrere Container eines Images zur gleichen Zeit ausgeführt werden. Ein besonderer Aspekt von Docker im Vergleich zu anderen Virtualisierungsumgebungen ist, dass sich die Docker-Container mit dem Host-Betriebssystem den Betriebssystem-Kernel teilen. Docker-Container können in der Regel flexibel in vielen

unterschiedlichen Umgebungen, wie beispielsweise auf PCs, in Datencentern oder in der Cloud, ausgeführt werden. Hierbei stellt die Docker-Engine die Kompatibilität zum Host her.

Docker ermöglicht es, dass Dateien zwischen Host und den Containern über gemeinsame Laufwerke (engl. „volumes“) ausgetauscht werden. Darüber hinaus können gemeinsame Umgebungsvariablen in verschiedenen Containern gesetzt werden und Container können über das lokale Netzwerk untereinander und mit dem Host kommunizieren.

Im Hinblick auf die vorherigen Ausführungen dieser Arbeit, eröffnet dies viele Möglichkeiten: (Micro-)Services können als Images so definiert werden, dass diese eine hohe Kohäsion aufweisen. Die Flexibilität der Images erlaubt es, die Systeme auf fast jeden Host zu portieren. Softwaresysteme bzw. -komponenten können über mehrere Container verteilt und modularisiert werden.

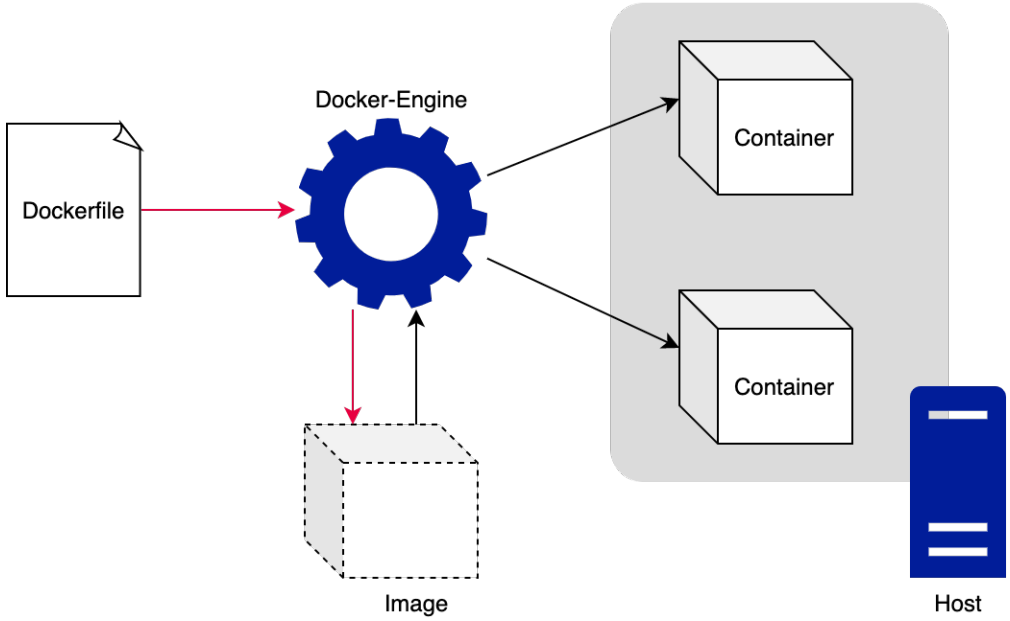

Abbildung 10: Docker-Umgebung

Müssen viele Container gemeinsam verwaltet werden, stößt Docker jedoch an seine Grenzen. Hier kommt die Orchestrierungs-Plattform Kubernetes ins Spiel.

Kubernetes

Das folgende Zitat aus [37] soll den Begriff der Orchestrierung näher umschreiben:

> *„Der Begriff Orchestrierung wurde aus dem Bereich der Musik entlehnt und bedeutet flexibles Kombinieren mehrerer Services oder Dienste zu einer sinnvollen Konzeption (Komposition), die einen Geschäftsprozess beschreibt.“*

Ein essentielles Werkzeug für die Orchestrierung von Softwaresystemen stellt Kubernetes dar. Wie in [38] beschrieben, ermöglicht Kubernetes eine erleichterte Bereitstellung von zuverlässigen, skalierbaren und verteilten Systemen.

Kubernetes bedient sich der deklarativen Konfiguration seiner Komponenten. Dies bedeutet, dass der gewünschte Zustand der Komponenten nicht durch die Ausführung mehrerer Kommandos, wie es bei einer imperativen Konfiguration üblich ist, erreicht wird. Stattdessen wird der gewünschte Zustand in Form einer deklarativen Konfiguration(-sdatei) definiert, die auf Kubernetes angewandt wird. Diese Art der Konfiguration ist weniger fehleranfällig und ermöglicht darüber hinaus den Einsatz von Versionsverwaltungssystemen, wie Git [39]. Die Infrastruktur wird so als Code definiert (IaC), was einem häufig verwendeten Design-Prinzip entspricht [40]. Beispiele solcher Konfigurationsdateien finden sich in Anhang B.

Es stellt sich die Frage, wie Kubernetes bei der Orchestrierung von Systemen hilfreich sein kann und dabei unter anderem eine lose Kopplung verfolgt. Hierzu ist zunächst wichtig zu verstehen, wie Systeme in Kubernetes bereitgestellt werden, in welcher Umgebung die Systeme ausgeführt werden und welche Artefakte Kubernetes dabei verwendet. Die folgende Abbildung soll eine Übersicht über den wesentlichen Aufbau von Kubernetes geben.

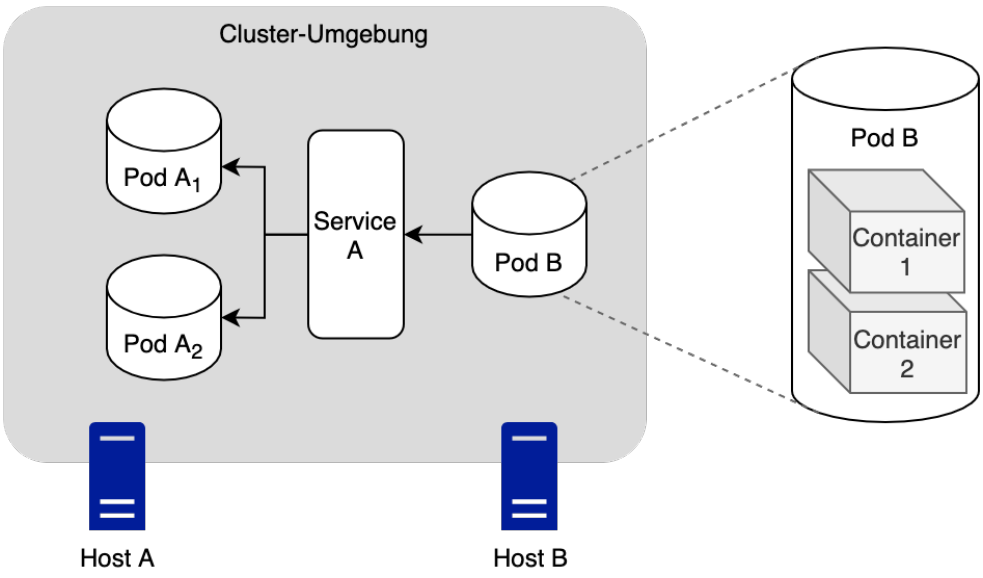

Abbildung 11: Kubernetes – Übersicht

Cluster-Umgebung

Kubernetes ermöglicht es mehrere physische Hosts in eine gemeinsame Cluster-Umgebung einzubinden. Die Hosts erscheinen dem Nutzer als eine große Umgebung, in welchem Softwaresysteme ausgeführt werden können. Dabei ist es in der Regel nicht wichtig zu wissen,

auf welchem Host der Knoten eines Systems konkret ausgeführt wird. Sofern ein Knoten dennoch auf einem spezifischen Host ausführt werden soll, kann dies entsprechend konfiguriert werden.

Dem Cluster steht ein eigenes Netzwerk zur Verfügung, in welchem den Knoten eines Systems virtuelle IPs zugewiesen werden. Zur Namensauflösung der URLs von Systemen steht dem Cluster ein eigener DNS-Service zur Verfügung.

Pods

Pods sind das kleinste deploybare Artefakt in Kubernetes und bestehen aus einem oder mehreren Docker-Containern. Die Container eines Pods werden immer auf demselben Host ausgeführt und haben die gleiche IP-Adresse und den gleichen Port-Space. Darüber hinaus können die Container eines Pods über native Interprozess-Kommunikationskanäle, wie Pipes oder Shared-Memory miteinander kommunizieren.

Bei der Konfiguration eines Pods wird unter anderem festgelegt, welche Docker-Images in einem Pod zur Erzeugung eines Containers genutzt werden und welche Ports der Container freigegeben werden. Es kann festgelegt werden, welche Volumes in die Container gemountet werden sollen und welche Ressourcen (RAM, CPU, ...) den Containern zur Verfügung stehen. Das Einrichten bestimmter Kriterien, nach denen ein Container als bereit (Readiness-Probe) bzw. verfügbar (Liveness-Probe) gilt, ermöglicht es Kubernetes automatisiert feststellen zu lassen, ob ein Pod den gewünschten Zustand eingenommen hat oder nicht.

Services

Um den Zugriff auf ein System zu ermöglichen, wird in Kubernetes das Artefakt des Service verwendet. Ein Service fasst die offenen Ports mehrerer Pods in einer gemeinsamen Schnittstelle zusammen und definiert, wie auf diese Pods bzw. die in den Pods enthaltenen Container zugegriffen werden soll.

Hierzu werden DNS-Namen für die Pods vergeben, welche auf die IPs der Pods aufgelöst werden. Das Verhalten der Namensauflösung kann dabei konfiguriert werden. Der Zugriff auf Systeme über einen Service als abstrakte Schnittstelle, ermöglicht eine sehr lose Kopplung mehrerer Systeme.

Die Konfiguration eines Service sieht im Wesentlichen vor, dass festgelegt wird, welche Pods zu einem Service gehören und welche Ports der Pods über den Service bereitgestellt werden sollen. Es stehen verschiedene Service-Typen zur Verfügung, wie z. B. ClusterIP oder

NodePort. Der Typ ClusterIP stellt den Service im lokalen Cluster-Netzwerk zur Verfügung, während der Typ NodePort hingegen den Service auf den Ports des Hosts bereitstellt.

Services bieten neben den hier erwähnten Funktionalitäten noch viele weitere Möglichkeiten, wie z. B. Load-Balancing über mehrere Pods. Im Fokus soll jedoch stehen, dass Services ein Hilfsmittel darstellen, welches es erlaubt mehrere Pods in einem gemeinsamen Zugriffspunkt zu abstrahieren. Dies ermöglicht einen übersichtlichen, lose gekoppelten Zugriff auf ein über mehrere Knoten (bzw. Pods) verteiltes System in einer definierten Schnittstelle [41].

Zusammenfassung

Die wichtigsten Kubernetes Artefakte, die für das Verständnis dieser Arbeit relevant sind, stellen Pods und Services dar. Die Knoten eines Systems werden durch Pods abgebildet, während die Services die Schnittstellen eines Systems repräsentieren. Für ein tieferes Verständnis von Kubernetes empfehlen sich das Buch „Kubernetes – eine Kompakte Einführung“ [38], sowie die offizielle Kubernetes Dokumentation [42].

Helm

Das Programm Helm ermöglicht eine einfache Installation von Systemen in Kubernetes über sogenannte Helm-Charts. Eine Helm-Chart umfasst im Allgemeinen die Konfigurationsdateien mehrerer Kubernetes-Artefakte. Hierbei können in den Konfigurationen verschiedene Felder als Variablen definiert werden, welche in einer globalen Datei (häufig values.yaml) ausformuliert werden. So kann eine Helm-Chart über die Value-File schnell und flexibel für bestimme Einsatzgebiete, wie Produktiv- und Testumgebungen, angepasst werden.

Helm-Charts können in Repositiories versioniert und mit anderen Nutzern geteilt werden. Weiterhin wird Helm in der Regel über die Kommandozeile genutzt, um ein System in Kubernetes zu installieren oder auch zu updaten. Die Updates können im Fehlerfall auch wieder zurückgedreht werden [43].

5.2 Versuchsaufbau

Nun, da die Werkzeuge bekannt sind, welche für den Versuch benötigt werden, sollen in diesem Abschnitt der grundlegende Aufbau des Versuches und die Architekturen der zu testenden Systeme beschrieben werden.

Allgemein

Im Zentrum des Aufbaus steht das zu testende System. Das zu testende System wird durch einen Datenerzeuger in einem festen Intervall kontinuierlich mit Daten beschrieben. Unabhängig von dem zu testenden System stellen die Daten hierbei immer eine Kombination aus einem Zahlenwert $k \in \mathbb{N}_0$ als Identen und einem Buchstaben als Wert dar. Der Wert lässt sich dabei aus dem Identen bestimmen und entspricht dem $(k \bmod 26 + 1)$-ten Buchstaben des lateinischen Alphabets.

Abbildung 12: Anwendung – Versuchsaufbau

Messung der Verfügbarkeit

Zur Messung der Verfügbarkeit werden in diesem Versuch sowohl die Dauer der schreibenden, als auch der lesenden Operationen gemessen. Beim Lesen wird vom ersten Datensatz ($k = 0$) an immer der Datensatz mit dem nächstgrößeren Identen ($k = 1, k = 2, \ldots$) gelesen. Da somit immer nur ein Datensatz gelesen wird, ist die Menge der zu lesenden Daten mit jedem Aufruf einheitlich.

Um einen gemeinsamen Wert für die Verfügbarkeit anzugeben, wird das arithmetische Mittel über die Dauer einer lesenden und einer schreibenden Operation gebildet, die den Datensatz des gleichen Identen behandeln. Da die lesenden Operationen mit den schreibenden Operationen korrespondieren, erfolgt die Messung beider Operationen im gleichen Intervall.

Messung der Konsistenz

Die Messung der Konsistenz erfolgt, indem parallel an allen einzelnen Knoten des zu testenden Systems die gesamte Datenmenge abgerufen wird und anhand dieser Daten per Definition der CAP-Charakteristik das Maß der Konsistenz berechnet wird. Es gelten zwei Datensätze als gleich, wenn die Kombinationen aus Identen und Buchstaben zweier Sätze übereinstimmen.

Da im späteren Verlauf der Mittelwert über einen Betrachtungszeitraum gebildet wird, wird an dieser Stelle vernachlässigt, dass die Abfrageergebnisse nicht garantiert von exakt demselben Zeitpunkt aller Knoten stammen. Die Messung der Konsistenz erfolgt im gleichen Intervall wie die Generierung der Daten.

Messung der Partitionstoleranz

Für PostgreSQL kann die Synchronisationsdauer aus dem Feld „replay_lag“ der Systemtabelle „pg_stat_replication“ ausgelesen werden und ist somit leicht zu bestimmen [44].

Die Bestimmung der Synchronisationsdauer stellt sich für Redis als schwieriger heraus, denn Redis bietet hierzu keine native Möglichkeit. Zur Messung werden stattdessen die in Redis implementierten Keyspace Notifications genutzt, die z. B. bei der Anlage eines Datensatzes erzeugt werden [45]. Für jeden Redis-Knoten wird ein Handler eingerichtet, der auf die Datenanlage reagiert und anhand der Datensatz-ID den Anlagezeitpunkt je Knoten sichert.

So kann die Zeitdifferenz zwischen den Anlagezeitpunkten und somit die Synchronisationsdauer berechnet werden. Da diese Messung nicht direkt auf den Knoten selbst erfolgt, ist davon auszugehen, dass hier größere Messungenauigkeiten möglich sind.

Auch die Messung der Partitionstoleranz erfolgt im gleichen Intervall wie die Generierung der Daten.

Architekturen der zu testenden Systeme

Redis Sentinel Architektur

Die für Redis verwendete Architektur orientiert sich an der in [46] spezifizierten Helm-Chart. Die bei dem Versuch verwendete Konfiguration (values-redis.yaml) ist in Anhang B dokumentiert.

Das System ist analog aufgebaut zu der im o. g. Beispiel zur Partitionstoleranz beschriebenen Architektur: Jeder der drei Knoten besteht aus einem Redis-Service und einem Sentinel-Service. Der Redis-Service stellt die grundlegenden Funktionen des Knotens bereit. Der Sentinel-Service überwacht alle Knoten, um im Falle einer Partition entsprechende Maßnahmen einzuleiten. Zusätzlich ist ein Proxy konfiguriert, welcher Anfragen automatisch an den aktuellen Primary weiterleitet. Sollte der Primary wechseln, wird der Proxy hierüber informiert, sodass die Anfragen weiterhin den korrekten Knoten erreichen [17].

Der Replikationsmechanismus ist für diesen Versuch so konfiguriert, dass der Primary nicht darauf wartet, bis mögliche Zustandsänderungen von den Secondaries verarbeitet sind. Darüber hinaus ist das System so konfiguriert, dass die Secondaries auf lesende Anfragen auch dann antworten dürfen, wenn sie noch nicht den aktuellsten Stand der Daten vom Primary empfangen haben [18].

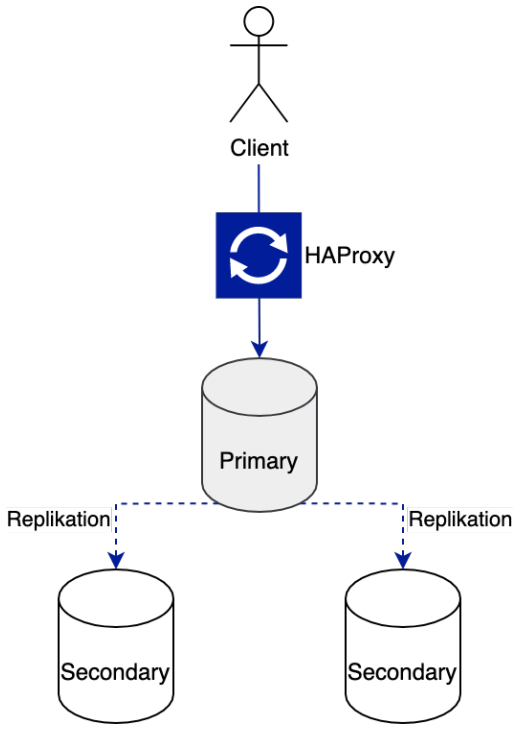

Abbildung 13: Redis Sentinel Architektur

Demnach implementiert diese Architektur ein System, welches einen hohen Wert auf die Verfügbarkeit und Partitionstoleranz legt und dabei hinnimmt, dass die Konsistenz nicht immer vollständig gegeben ist. Dies entspricht dem Verhalten, welches vom BASE-Prinzip definiert wird [47].

PostgreSQL-HA Architektur

Die hier verwendete Architektur orientiert sich an der in [48] spezifizierten Helm-Chart. Die bei dem Versuch verwendete Konfiguration (values-postgres.yaml) ist in Anhang B dokumentiert.

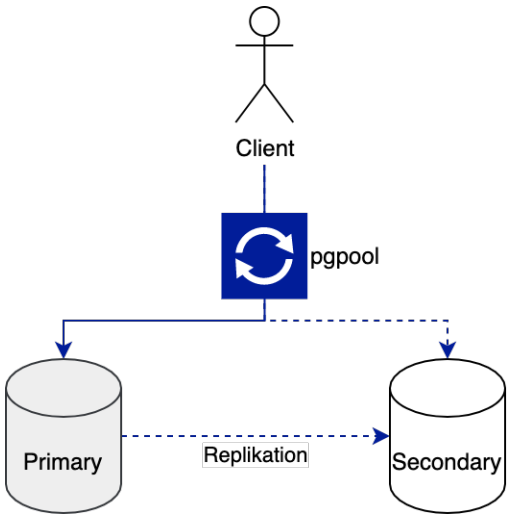

Abbildung 14: PostgreSQL-HA Architektur

Im Allgemeinen ist die PostgreSQL-HA Architektur der Redis Sentinel Architektur relativ ähnlich. In der hier genutzten Konfiguration besteht das System aus zwei Knoten. Von diesen beiden Knoten stellt einer den Primary und der andere den Secondary dar. Die Daten werden dabei kontinuierlich auf den Secondary gespiegelt.

Der Client verbindet sich zu dem System über den pgpool. Der pgpool überwacht Primary und Secondary und ist somit dafür verantwortlich, die Verbindung des Clients, wie ein Proxy, an den korrekten Endpunkt weiterzuleiten. Dabei werden die Verbindungen teilweise abhängig vom jeweiligen SQL-Command entweder an den Primary oder einen Secondary weitergeleitet. Sofern jedoch der Zustand des Secondaries zu stark vom Primary abweicht, werden alle Anfragen an den Primary weitergeleitet [49].

Verliert der pgpool z. B. wegen einer Partition die Verbindung zu einem der Knoten, so werden alle aktiven Sitzungen zu diesem Knoten unterbrochen. Sollte die Verbindung zum Primary abbrechen, wählt der pgpool einen neuen Primary aus den Secondaries aus. Verliert der pgpool die Verbindung zu allen Knoten, so kann keine Verbindung mehr für den Client bereitgestellt werden [50].

Das hier beschriebene System scheint also großen Wert auf die Konsistenz zu legen. Die Verfügbarkeit wird teilweise zu Gunsten der Konsistenz reduziert, wenn z. B. die Replikation des Secondaries zu langsam ist. Die Partitionstoleranz ist solange gegeben, bis der pgpool von allen Knoten isoliert wird. Es handelt sich hierbei um ein System, welches im Wesentlichen dem ACID-Prinzip folgt [30].

5.3 Versuchsdurchführung

Technische Daten

Der Versuch wird auf einem MacBook Pro (13 Zoll, Mitte 2012) mit einem 2,5 GHz Dual-Core Intel Core i5 Prozessor und einem 16 GB 1600 MHz DDR3 Arbeitsspeicher durchgeführt. Verwendet wird Docker Desktop 3.5.1 mit der Docker-Engine 20.10.7. Der Docker-Engine werden dabei 3 CPU Kerne und 8 GB RAM, sowie 3 GB Swap-Speicher zugewiesen. Für Kubernetes wird die in Docker Desktop integrierte Implementierung (v1.21.1) genutzt.

Es werden zur Installation von Redis und PostgreSQL in Kubernetes die Helm-Charts unter [46] (Chart-Version 4.12.15) bzw. [48] (Chart-Version 7.7.1) mit den Value-Files values-redis.yaml bzw. values-pg.yaml (siehe Anhang B) verwendet. Hierbei werden alle Pods, entgegen der Empfehlung der Charts, auf demselben Host ausgeführt, weil es sich bei dem in Docker Desktop integrierten Cluster um ein Cluster mit nur einem Host handelt. Da somit eine Kommunikation über Wide-Area-Networks ausgeschlossen ist, ist von einer geringen Latenz zwischen den Knoten bzw. Pods in diesem Versuchsaufbau auszugehen.

Zur Messung der CAP-Eigenschaften werden Python-Skripte [51] (siehe Anhang A) in einem Container gebündelt und als Pods in Kubernetes ausgeführt. Die Dockerfile und Pod-Definitionen sind in Anhang B hinterlegt. Das Basis-Image der Dockerfile „python:3“ hat zum Zeitpunkt des Versuches die Python-Version 3.9.6 beinhaltet.

Durchführung

Die beiden Systeme werden isoliert voneinander getestet, d. h. sie werden nicht zur gleichen Zeit in Kubernetes ausgeführt. Zunächst wird per

```
helm install redis dandydev/redis-ha -f values-redis.yaml
```

Redis bzw. per

```
helm install postgres bitnami/postgresql-ha -f values-postgres.yaml
```

PostgreSQL installiert.

Während die Pods der Anwendung starten, werden zusätzliche Services angelegt, welche nicht in den Helm-Charts enthalten sind, jedoch nötig sind, um auf die einzelnen Knoten direkt zuzugreifen:

```
kubectl expose pod redis-redis-ha-server-0
kubectl expose pod redis-redis-ha-server-1
kubectl expose pod redis-redis-ha-server-2
```

bzw.

```
kubectl expose pod postgres-postgresql-ha-postgresql-0
kubectl expose pod postgres-postgresql-ha-postgresql-1
```

Die Definitionen der Pods für die Messung der CAP-Eigenschaften (siehe Anhang B) sind in gemeinsamen Ordnern „redis“ bzw. „pg“ abgelegt, sodass diese mit

```
kubectl apply -f ./redis
```

bzw.

```
kubectl apply -f ./pg
```

simultan gestartet werden konnten. Mit Hilfe des Befehls

```
kubectl logs -f <POD-NAME>
```

werden die Messwerte der Pods in der Kommandozeile ausgegeben und für die Auswertung verarbeitet.

Die Systeme werden über den Zeitraum von einer Minute betrachtet. Die Datengeneratoren generieren im Intervall von einer Sekunde Datensätze. Die Analyse-Pods für die Konsistenz und Partitionstoleranz bilden ebenfalls im Sekundentakt ihre Messwerte.

Zu bemerken ist an dieser Stelle, dass es schwierig ist, die Pods so zu synchronisieren, dass Sie ihre Messwerte zum gleichen Zeitpunkt abgreifen. Da die Pods jedoch annähernd zum gleichen Zeitpunkt gestartet werden, wird diese Ungenauigkeit im Versuchsergebnis vernachlässigt.

5.4 Versuchsauswertung

Bei der Messung hat sich herausgestellt, dass sich in diesem Versuch $a_{max} = 500\text{ms}$ und $p_{max} = 500\text{ms}$ als geeignete obere Grenzen eignen. Im Folgenden werden die Messwerte, wie in Abschnitt 4.2 diskutiert, jeweils als Graph und Dreieck dargestellt. Die vollständige, tabellarische Auflistung aller Messwerte ist in Anhang C zu finden.

Darstellung als Graph

Verfügbarkeit

Die Diagramme weisen ähnliche Ergebnisse für die Verfügbarkeit beider Systeme auf. Bis auf einige Abweichungen, streut die Mehrzahl der Messergebnisse zwischen 1,0 und 0,8. Dies stellt ein gutes Ergebnis dar. Die Trendlinie zeigt in beiden Fällen, dass sich die Verfügbarkeit mit der Zeit im Schnitt leicht verbessert hat.

Auffällig ist, dass die Streuung der Messwerte für Redis stärker ausfällt als für PostgreSQL. Dies macht sich vor allem dadurch bemerkbar, dass einige Messwerte stark nach unten abweichen und die Anzahl der ausschweifenden Messwerte größer ist.

Diese Beobachtung kann als Grundlage weiterer Analysen dienen, die z. B. untersuchen, ob PostgreSQL mehr Wert auf eine konsistente Antwortlatenz legt als Redis.

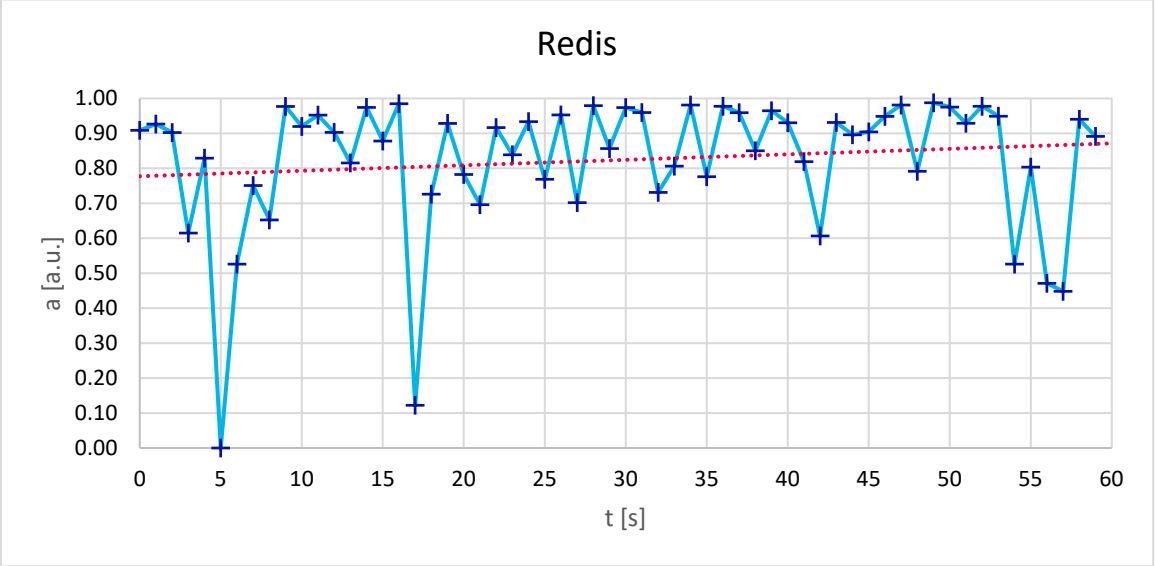

Diagramm 2: Versuchsauswertung – Verfügbarkeit von Redis

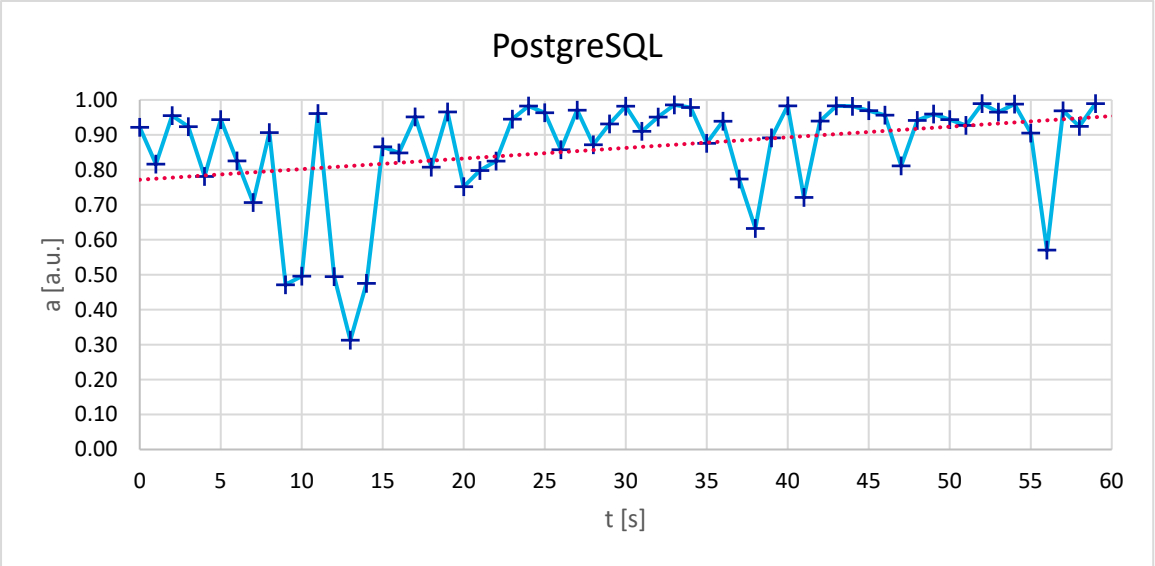

Diagramm 3: Versuchsauswertung – Verfügbarkeit von PostgreSQL

Konsistenz

Im Verlauf der gesamten Messung weist PostgreSQL keine einzige Inkonsistenz auf. Dies spricht dafür, dass das System über alle seine Knoten hinweg eine stetige Konsistenz anstrebt, was dem ACID-Prinzip [30] entspricht.

Redis hingegen zeigt kleinere Inkonsistenzen während der Messung auf. Das beobachtete Verhalten weist wiederum darauf hin, dass bei Redis das BASE-Prinzip verfolgt wird. Dem BASE Prinzip genügt im Gegensatz zum ACID-Prinzip die eventuelle Konsistenz [47].

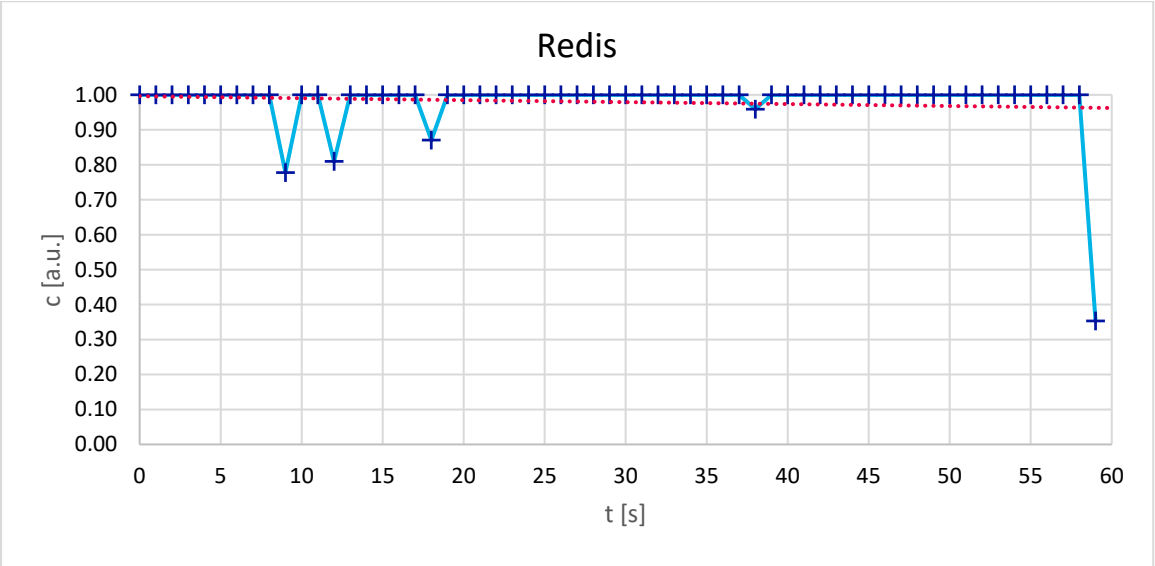

Diagramm 4: Versuchsauswertung – Konsistenz von Redis

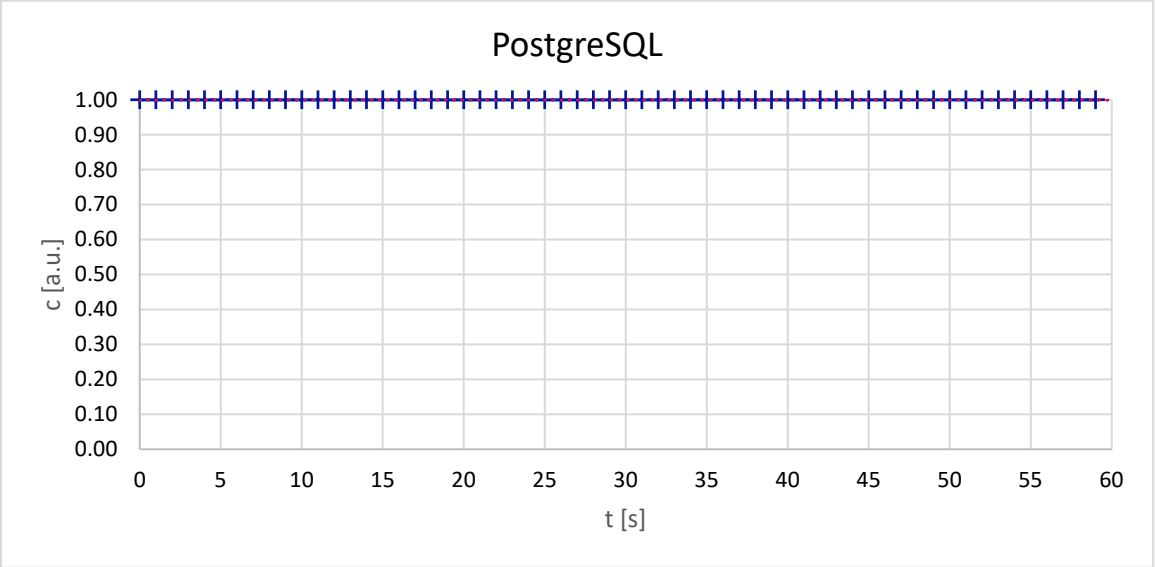

Diagramm 5: Versuchsauswertung – Konsistenz von PostgreSQL

Partitionstoleranz

Bei der Partitionstoleranz lässt sich der größte Unterschied zwischen den beiden Systemen feststellen. Während PostgreSQL bei der gesamten Messung gute Werte aufzeigt, streuen die Messwerte für Redis über den gesamten Wertebereich. Im Mittel betrachtet sind die Werte für Redis zwar noch akzeptabel (Mittelwert = 0,7), sie sind jedoch deutlich schlechter als die Werte für PostgreSQL.

Da jedoch, wie in Abschnitt 5.2 beschrieben, das Messverfahren für Redis ein größeres Risiko darstellt, die Ergebnisse zu verfälschen, sind die Messwerte für Redis an dieser Stelle kritisch zu betrachten. Dieses Problem wird in Kapitel 6 weiter diskutiert.

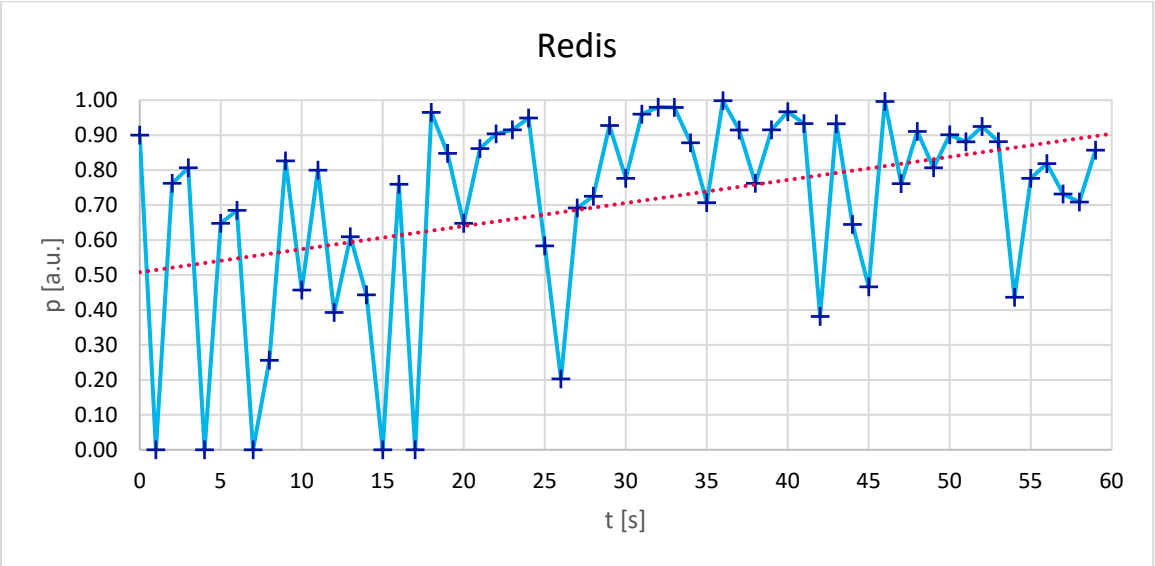

Diagramm 6: Versuchsauswertung – Partitionstoleranz von Redis

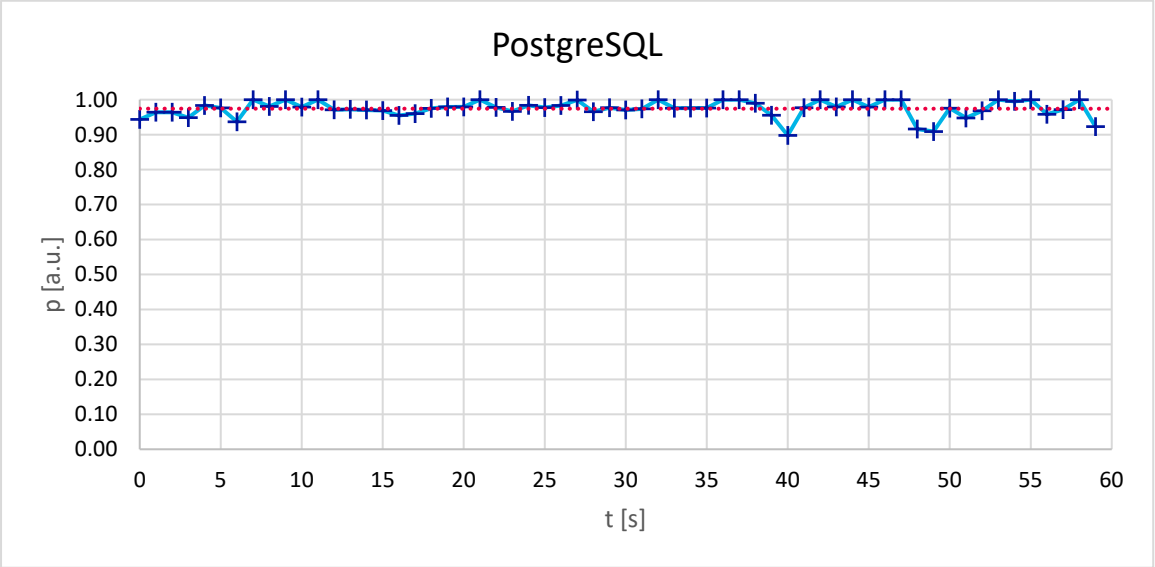

Diagramm 7: Versuchsauswertung – Partitionstoleranz von PostgreSQL

Darstellung als Dreieck

Im Folgenden werden die Mittelwerte der einzelnen CAP-Eigenschaften je System wie unter Abschnitt 4.2 beschrieben als Dreieck dargestellt. Aus den Messreihen ergeben sich folgende Mittelwerte:

	$\bar{c}$ [a.u.]	$\bar{a}$ [a.u.]	$\bar{p}$ [a.u.]
Redis	1,00	0,86	0,97
PostgreSQL	0,98	0,82	0,70

Tabelle 4: Mittelwerte der Messreihen

Aus der Tabelle ergeben sich wiederum folgende Dreiecke:

Redis

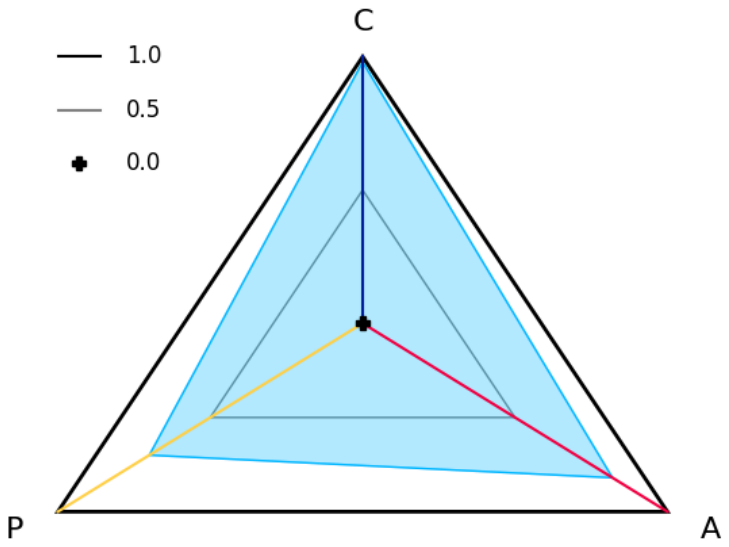

Abbildung 15: Redis – Darstellung als Dreieck

Redis zeigt ein Defizit in der Partitionstoleranz auf, während die Konsistenz und Verfügbarkeit fast vollständig erfüllt sind.

PostgreSQL

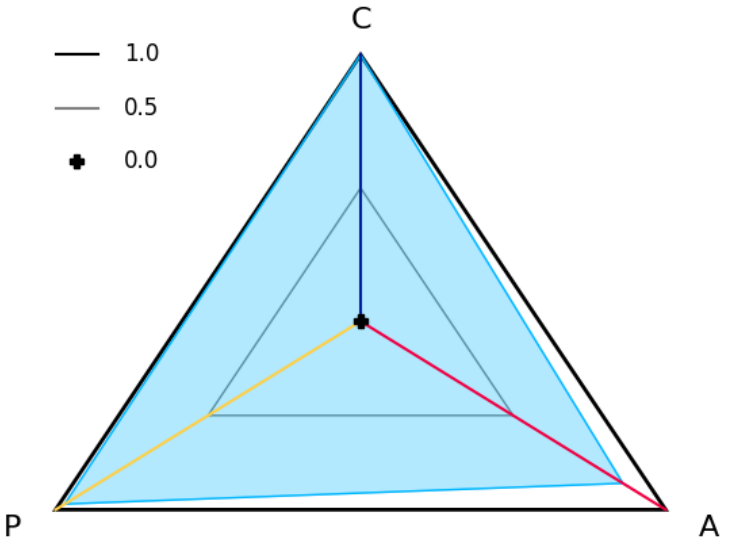

Abbildung 16: PostgreSQL – Darstellung als Dreieck

PostgreSQL füllt das Dreieck fast vollständig aus und zeigt somit keine Defizite.

5.5 Zusammenfassung

Vergleicht man die Messreihen und Dreiecke der beiden Systeme, so fällt auf, dass zwar beide Systeme die CAP-Eigenschaften fast vollständig erfüllen, jedoch Redis in diesem Versuch PostgreSQL unterlegen ist. Vor allem in der Partitionstoleranz schneidet Redis schlechter als PostgreSQL ab, in der Verfügbarkeit und Konsistenz unterscheiden sich die beiden Systeme kaum. Dieses Ergebnis ist zum Teil überraschend, sofern man betrachtet, dass Redis dem BASE-Prinzip und PostgreSQL dem ACID-Prinzip folgt.

Da das BASE-Prinzip im Gegensatz zum ACID-Prinzip keine starke Konsistenz voraussetzt, wäre es plausibel gewesen, dass Redis in der Konsistenz schlechter abschneidet als PostgreSQL [30], [47]. Aus gleichem Grunde wäre eine geringere Verfügbarkeit bei PostgreSQL im Vergleich zu Redis denkbar gewesen, da das ACID-Prinzip abgeschlossene Transaktionen voraussetzt. So hätte eine lesende Anfrage auf einen Datensatz beispielsweise solange verzögert werden können, bis eine schreibende Transaktion auf diesen abgeschlossen wurde.

Im Hinblick auf die verwendeten Architekturen wurden jedoch auch bestimmte Mechanismen implementiert, die auf das hier erzielte Ergebnis hinarbeiten. So werden beispielsweise bei PostgreSQL lesende Anfragen zwar zwischen den beiden Knoten verteilt, jedoch werden die lesenden Anfragen ab einer bestimmten Synchronisations-Differenz zwischen den beiden Knoten nur noch an den Primary geleitet, um durch Transaktionen bedingte Verzögerungen zu vermeiden [49].

Letztlich dienen die für die Systeme verwendeten Architekturen dem Zweck, die hier getesteten Eigenschaften möglichst stark zu erfüllen. Daher ist es ggf. nötig, die Systeme z. B. durch spezifischere Testszenarien und höhere Auslastungen ausgiebiger zu testen, um eindeutigere Ergebnisse zu erlangen. Dieser und weitere Punkte werden im folgenden Kapitel weiter erörtert.

6. Evaluierung

In diesem Kapitel wird diskutiert, welche Aspekte des Versuches gut abgelaufen sind und welche Aspekte weiter verbessert werden können. Aus dieser Diskussion wird ein Rückschluss darauf gezogen, ob die CAP-Charakteristik ein geeignetes Mittel zu Charakterisierung von Softwaresystemen darstellt oder aber die Charakteristik noch weiter geschärft werden muss.

6.1 Reflektion des Versuches

Bewertung des Szenarios

Der Versuch hat gezeigt, dass beide Systeme alle CAP-Eigenschaften fast vollständig erfüllen. Es liegt die Vermutung nahe, dass die Auslastung, welche während des Versuches erzeugt worden ist, eine vergleichsweise geringe Belastung für die Systeme dargestellt hat. Dadurch könnten die Messwerte verfälscht worden sein. Damit der Versuch zuverlässigere Resultate liefert, wäre es sinnvoll, die Systeme an ihre Leistungsgrenze mit den zur Verfügung gestellten Systemressourcen zu bringen. Möglichkeiten, dies umzusetzen bestehen beispielsweise darin, die Anzahl der Clients zu erhöhen oder das Intervall der Datengeneratoren und Messungen zu verkleinern.

Das Szenario welches für diesen Versuch gewählt wurde, wurde der Nachvollziehbarkeit halber möglichst einfach gehalten. Es wurden, wie man den Helm Value-Files entnehmen kann, kaum Einstellungen der Systeme verändert. In den Einstellungen von Kubernetes wurden keine Veränderungen vorgenommen. Somit konnte eine leicht herstellbare, normierte Versuchsumgebung geschaffen werden.

Eine Option, um spezifischere Ergebnisse zu erzielen, bringt die Variation weiterer Parameter der normierten Versuchsumgebung. Solche Parameter können z. B. die Netzwerk- oder Systemauslastung sein. Das Simulieren von zufälligen Partitionen durch Packet Loss [26] würde ein aussagekräftigeres Ergebnis hervorbringen.

Messintervalle und -genauigkeit

Messintervall

Für diesen Versuch ist das Intervall der Messung der CAP-Eigenschaften mit dem Intervall der Datengeneration gleichgesetzt worden. Es gilt nun zu prüfen, ob dies sinnvoll ist.

Würde man bei der Messung ein geringeres Intervall wählen als das Intervall der Datengeneration, so ist die Wahrscheinlichkeit höher, dass bei der Messung der Konsistenz temporär bestehende Inkonsistenzen erkannt werden. Dabei muss berücksichtigt werden, dass das Abrufen der Daten eine bestimmte Zeit braucht und häufige Anfragen gleichzeitig die Systemlast erhöhen.

Ähnlich verhält sich ein geringeres Abfrageintervall für die Messung der Partitionstoleranz bei PostgreSQL. Auch hier könnten durch ein geringeres Intervall ggf. höhere Synchronisationsdifferenzen erkannt werden, da die Wahrscheinlichkeit verringert wird, die Abfrage eines temporär bestehenden Wertes zu „verpassen“.

Messungenauigkeiten

Im Rahmen des Versuches sind einige Quellen für Messungenauigkeiten toleriert worden, welche noch eliminiert werden könnten. Hierzu gehört unter anderem, dass das verwendete Cluster nur aus einem Host bestanden hat. Die Spezifikationen der Systeme empfehlen es jedoch, dass die einzelnen Knoten auf jeweils einem eigenen Host ausführt werden. Dies könnte Messungenauigkeiten verursacht haben.

Ebenfalls ist nicht darauf geachtet worden, dass die Messungen der Eigenschaften synchron erfolgt sind. Dadurch, dass die Container, welche die Messungen durchgeführt haben, unterschiedlich schnell gestartet sind, können die Messungen mit einer kleinen zeitlichen Versetzung erfolgt sein. Ob dies von wesentlicher weiterer Bedeutung für die Messergebnisse ist, bzw. diese kleine Ungenauigkeit nicht tolerierbar ist, gilt es zu hinterfragen.

Messung der Partitionstoleranz

Es stellt sich die Frage, ob das für die Partitionstoleranz verwendete Maß auch dafür geeignet ist, um die Partitionstoleranz zu messen. Das hier genutzte Maß zielt primär darauf ab, zu ermitteln, ob ein Knoten im Falle einer Partition eigenständig agieren kann. Dieses Kriterium wird als gegeben angesehen, sofern alle Knoten untereinander keine Inkonsistenzen aufweisen.

Auf den ersten Blick erscheint dieses Maß keines zu sein, welches die Partitionstoleranz, sondern eher die Konsistenz misst. Bei genauerer Betrachtung fällt jedoch auf, dass nicht die Konsistenz selbst gemessen wird, sondern die Dauer, bis die Konsistenz zwischen den Knoten hergestellt ist.

Sind alle Knoten konsistent, so könnten diese eigenständig im Falle der Partition agieren. Letztlich existieren aber auch Systeme, in denen einige Knoten gar nicht die volle Funktionalität des Systems bereitstellen müssen. Dieses Kriterium wird derzeit nicht berücksichtigt.

Ebenfalls wäre es möglich die Synchronisations-Differenz nicht als zeitliches Maß zu messen, sondern als Menge der Bytes, die noch nicht zwischen den Knoten ausgetauscht worden sind. Die Menge der noch nicht synchronisierten Bytes lässt sich nativ aus Redis und PostgreSQL auslesen. Das native Auslesen hätte zusätzlich den Vorteil, dass die Werte der beiden Systeme vergleichbarer wären als im aktuellen Versuchsaufbau.

6.2 Rückschluss auf die CAP-Charakteristik

Unter Berücksichtigung der unter 6.1 genannten Kritikpunkte, wird nun diskutiert, ob die CAP-Charakteristik in der hier definierten Form zur Bewertung von (Micro-)Service-Architekturen geeignet ist.

Weitere Versuche

Der in dieser Arbeit durchgeführte Versuch deckt nur eines von vielen möglichen Szenarien ab, unter denen die Systeme getestet werden können. Da das hier gewählte Szenario ein möglichst einfaches Szenario darstellt, ist davon auszugehen, dass die Versuchsergebnisse als gute Grundlage für weiterführende Szenarien gelten. Auf der anderen Seite ist auf Grund der geringen Belastung, welche in dem hier durchgeführten Versuch erzeugt worden ist, das Versuchsergebnis nicht zwingend repräsentativ.

Daher sollten weitere Versuchsszenarien durchgeführt werden, die anhand ihres Aufbaus bestimmte Ergebnisse provozieren. So kann z. B. ein Versuch konstruiert werden, der darauf abzielt die Verfügbarkeit der Systeme stark zu fordern bzw. einzuschränken. Das Ergebnis des Versuches dürfte schlechtere Werte für die Verfügbarkeit liefern. Mit dieser Art des Versuchsszenarios können die Definitionen der einzelnen Eigenschaften der CAP-Charakteristik gezielt stärker validiert werden.

Quantitative Messung

Anhand der Versuchsauswertung ist deutlich geworden, dass die quantitative Messung der CAP-Eigenschaften eine wesentlich tiefere Analyse von (Micro-)Service-Architekturen ermöglicht, als es qualitative Eigenschaften erlauben würden. Mit der Hilfe von Graphen können die Ergebnisse der Messreihen leicht visualisiert werden und es können vertiefende mathematische Analysen der Ergebnisse angestellt werden. Dies wäre mit qualitativen Eigenschaften nur begrenzt möglich.

Die Darstellung der Eigenschaften als Dreieck ermöglicht es, für eine Vielzahl von Systemen einen Fingerabdruck zu erstellen. Dieser Fingerabdruck kann z. B. in den Git-Repositories der Systeme hinterlegt werden, um einem Nutzer auf einen Blick zu vermitteln, wie das System charakterisiert werden kann.

Die quantitative Interpretation der CAP-Eigenschaften entspricht eher Brewers Interpretation der Eigenschaften. Darüber hinaus berücksichtigt die CAP-Charakteristik die zeitliche Entwicklung der Eigenschaften eines Systems. Somit wird auch die von Brewer aufgeführte Behandlung von Partitionen in der Messung der Eigenschaften berücksichtigt [19].

Alternative Maße für Partitionstoleranz

Die Messung der Partitionstoleranz hat sich als eine größere Herausforderung herausgestellt. In der durchgeführten Versuchsreihe ist zwar eine Lösung für die Messung der Partitionstoleranz konstruiert worden, jedoch ist auf Grund der unterschiedlichen Arten, wie die Messwerte an den Systemen festgestellt worden sind, zu erwarten, dass diese nicht gut miteinander vergleichbar sind.

Aus diesem Grund sollte für zukünftige Versuche entweder ein anderes Maß für die Partitionstoleranz definiert oder ein einheitlicher Weg gesucht werden, die Messwerte zu ermitteln.

An dieser Stelle ist es von Vorteil, dass die einzelnen Eigenschaften der CAP-Charakteristik unabhängig voneinander bestehen und somit austauschbar sind. Die CAP-Charakteristik umfasst nicht nur die Definition der Eigenschaften, auch die Art der Auswertung, die angestellt worden ist, stellt einen fundamentalen Teil der Charakteristik dar.

7. Zusammenfassung

Brewers ursprüngliche Aussage, dass ein System maximal zwei der drei Eigenschaften, Verfügbarkeit, Konsistenz und Partitionstoleranz erfüllen kann, hat bis heute einen großen Einfluss auf die Entwicklung von Software und deren Architekturen gehabt. So ist basierend auf Brewers Aussage von Gilbert und Lynch das CAP-Theorem formuliert worden und in der Theorie bewiesen worden. Die Idee, zwei der drei Eigenschaften auf das höchstmögliche Niveau zu bringen und dabei eine der Eigenschaften zu vernachlässigen, ist zum Teil eine Folge des Theorems gewesen.

Mit der Zeit sind jedoch diverse Kritiken in Bezug auf das CAP-Theorem geäußert worden. Insgesamt haben viele dieser Kritiken gemein, dass die Definitionen der Verfügbarkeit, Konsistenz und Partitionstoleranz nach Gilbert und Lynch für die Praxis untauglich und im Allgemeinen ungenau sind. Die binäre, qualitative Betrachtungsweise der CAP-Eigenschaften ist laut Kritiken zu starr und bietet nur eine eingeschränkte Approximation der Realität. Dadurch, dass das CAP-Theorem keine latenzbasierten Ansätze berücksichtigt, bleiben viele Möglichkeiten unbeachtet.

Aus diesem Grunde sind im Rahmen dieser Arbeit die CAP-Eigenschaften als quantitative Maße neu definiert worden und sind über einen Zeitraum an konkreten Systemen gemessen worden. Die quantitative Formulierung der Eigenschaften bietet mehr Flexibilität und bildet die Realität präziser ab. Die Messung über einen Zeitraum berücksichtigt dabei, dass die CAP-Eigenschaften eines Systems sich mit dem Lauf der Zeit verändern können. Durch die Bildung von Mittelwerten werden die Veränderungen ausgeglichen und es ist sogar möglich, dass ein System alle drei Eigenschaften erfüllt, wenn auch nicht zum gleichen Zeitpunkt.

Auch ist festgestellt worden, dass die Messung der quantitativen CAP-Eigenschaften an konkreten Systemen keine leichte Aufgabe ist. Es gilt, bessere Messverfahren zu finden, um die CAP-Eigenschaften eines Systems festzustellen. Dabei können konkrete Versuchsszenarien umgesetzt werden, um bestimmte Eigenschaften eines Systems zu testen. Die Versuchsauswertung dieser Arbeit hat gezeigt, dass es lohnenswert ist, Systeme nach der CAP-Charakteristik auszumessen.

8. Ausblick

Zuverlässigkeit der Messverfahren

Wie sich bereits abgezeichnet hat, gilt es weiterhin, die Messverfahren der CAP-Eigenschaften zu verbessern. Die zuverlässige Messung der CAP-Eigenschaften stellt eine größere Herausforderung dar, als ursprünglich angenommen.

Am wichtigsten ist es, dass die Abnahme der Messwerte vieler Systemen auf die gleiche Weise erfolgen kann, um die Vergleichbarkeit dieser sicherzustellen. Ist dieses Kriterium erfüllt, so kann die CAP-Charakteristik als Mittel genutzt werden, um schnell für bestimmte Anwendungsfälle das richtige System auszuwählen.

Weitere Anwendungsfälle

In dieser Arbeit wurden zwei Systeme als Testobjekte verwendet, welche Datenbanken sind. Es ist bisher ungeklärt, wie man mit der CAP-Charakteristik beispielsweise Web-Services analysiert, die das HTTP-Protokoll nutzen.

Für diese und weitere Systeme muss festgelegt werden, wie sie mit der CAP-Charakteristik analysiert werden können. Die Charakteristik soll für möglichst viele Anwendungsfälle nutzbar gemacht werden.

Anbindung an Grafana

Zur praxisnahen Analyse von Systemen kann in Betracht gezogen werden, die Messwerte der Analyse-Container über Schnittstellen bereitzustellen, welche die Daten in einem für Prometheus kompatiblem Format darstellen. Prometheus [52] ist eine mögliche Grafana-Datenquelle [3], [4], welche dann die Maße der CAP-Eigenschaften von den Analyse-Containern abgreifen kann.

Somit wird die Einbindung der CAP-Charakteristik in das Monitoring-System ermöglicht. Darüber hinaus kann ein Grafana-Plugin [4] entwickelt werden, welches dann die CAP-Eigenschaften der Systeme in der Dreiecks-Darstellung in Echtzeit wiedergibt.

Anhang

Um zu vermeiden, diese schriftliche Ausführung mit Quellcode usw. zu überladen, wird ein Teil des Anhangs als digitale Kopie ausgeliefert. Die Dateien können zusätzlich unter https://ba.leonkrass.de abgerufen werden. Sollte der Link nicht funktionieren, kann jederzeit eine neue digitale Kopie beim Autor angefordert werden.

A: Quellcode

Im Folgenden werden die Quellcode-Dateien aufgelistet und deren Zweck beschrieben:

```
./caplot/caplot.py
```

Dieses Skript dient dem Plotten der Dreiecksdarstellung der CAP-Charakteristik.

```
./capalyzer/common.py
```

Diese Datei enthält gemeinsame Variablen aller übrigen Skripte im Ordner „capalyzer“.

```
./caplot/requirements.txt

./capalyzer/requirements.txt
```

In diesen Dateien sind alle externen Bibliotheken aufgelistet [31], [53]–[55].

```
./capalyzer/redis-datagen.py

./capalyzer/pg-datagen.py
```

Diese Skripte implementieren die Datengeneration für Redis und PostgreSQL.

```
./capalyzer/redis-availability.py

./capalyzer/pg-datagen.py
```

In diesen Dateien findet sich der Quellcode für die Analyse der Verfügbarkeit der Systeme. Analog existieren auch Skripte für die Konsistenz und die Partitionstoleranz.

B: Konfigurationsdateien

Helm Value-Files

Die folgenden Value-Files wurden verwendet, um Redis bzw. PostgreSQL mit den Helm-Charts [46] bzw. [48] zu installieren:

values-redis.yaml

```yaml
haproxy:
  enabled: true
  replicas: 1
  hardAntiAffinity: false

hardAntiAffinity: false
```

values-postgres.yaml

```yaml
global:
  postgresql:
    username: admin
    password: admin
    repmgrUsername: admin
    repmgrPassword: admin
  pgpool:
    adminUsername: admin
    adminPassword: admin
```

Dockerfile für die Analyse-Container

Um den Quellcode aus Anhang A in Docker-Container zu verpacken, wurde folgende Dockerfile genutzt:

```
FROM python:3

WORKDIR /capalyzer

COPY <Dateipfad> .

RUN rm -rf ./venv

RUN pip install --no-cache-dir -r requirements.txt

CMD echo "Use 'python -u' with one of the following as container-command:" \

      $(grep --include=\*.py --exclude-dir=venv -rnwl '.' -e "__name__ == \

      '__main__'" | tr ' ' ', ')
```

Der Platzhalter „<Dateipfad>“ muss dabei mit dem Pfad des „capalizer“-Ordners ersetzt werden.

Kubernetes-Artefakt für Analyse-Pods

Die folgende Konfiguration stellt einen Pod für Kubernetes dar, der die Analyse-Container nutzt. Hierbei muss der Platzhalter „<Script>“ mit dem entsprechenden Python-Skript ersetzt werden.

```
apiVersion: v1

kind: Pod

metadata:

  name: redis-availability

  labels:

    name: redis-availability

spec:

  containers:

  - name: redis-availability

    image: capalyzer:latest

    command: ["python", "-u", "<Script>"]

    imagePullPolicy: Never
```

C: Messwerte

Hier werden alle Messwerte des durchgeführten Versuches aufgeführt:

Redis

t [s]	a_{abs} [ms]		p_{abs} [ms]	c [a.u.]	a [a.u.]	p [a.u]
	lesen	schreiben				
0	67,52	23,57	50,07	1,00	0,91	0,90
1	7,33	65,94	627,68	1,00	0,93	0,00
2	24,14	73,17	118,95	1,00	0,90	0,76
3	33,26	351,86	96,74	1,00	0,61	0,81
4	10,85	159,86	830,29	1,00	0,83	0,00
5	92,76	1194,07	176,09	1,00	0,00	0,65
6	9,30	464,34	157,66	1,00	0,53	0,68
7	148,27	100,53	991,96	1,00	0,75	0,00
8	150,03	197,24	372,00	1,00	0,65	0,26
9	14,49	8,40	86,78	0,78	0,98	0,83
10	36,44	43,55	271,61	1,00	0,92	0,46
11	4,17	43,47	100,14	1,00	0,95	0,80
12	18,38	78,73	303,70	0,81	0,90	0,39
13	22,21	162,39	195,26	1,00	0,82	0,61
14	16,53	8,86	278,45	1,00	0,97	0,44
15	91,04	30,65	649,93	1,00	0,88	0,00
16	6,33	8,80	120,17	1,00	0,98	0,76
17	173,83	703,68	569,69	1,00	0,12	0,00
18	92,20	181,68	17,46	0,87	0,73	0,97
19	21,62	49,88	75,97	1,00	0,93	0,85
20	155,04	62,36	176,03	1,00	0,78	0,65
21	39,42	264,61	69,19	1,00	0,70	0,86
22	62,90	20,47	48,16	1,00	0,92	0,90
23	143,90	17,07	42,52	1,00	0,84	0,91
24	34,06	32,41	25,55	1,00	0,93	0,95
25	224,68	6,50	208,48	1,00	0,77	0,58
26	28,79	18,32	398,63	1,00	0,95	0,20
27	291,77	6,09	154,22	1,00	0,70	0,69
28	1,58	18,91	137,35	1,00	0,98	0,73
29	50,36	92,70	36,31	1,00	0,86	0,93
30	21,37	4,60	111,68	1,00	0,97	0,78
31	15,86	24,54	19,89	1,00	0,96	0,96
32	3,92	264,82	10,13	1,00	0,73	0,98
33	48,86	144,94	10,32	1,00	0,81	0,98
34	14,64	4,11	60,93	1,00	0,98	0,88
35	162,57	61,40	146,53	1,00	0,78	0,71
36	19,62	2,67	0,73	1,00	0,98	1,00

37	22,08	18,56	42,76	1,00	0,96	0,91
38	135,01	14,61	118,97	0,96	0,85	0,76
39	9,36	25,64	42,47	1,00	0,97	0,92
40	50,28	19,21	16,48	1,00	0,93	0,97
41	27,26	153,43	33,59	1,00	0,82	0,93
42	79,91	313,40	309,29	1,00	0,61	0,38
43	53,38	15,18	33,65	1,00	0,93	0,93
44	34,50	69,44	177,72	1,00	0,90	0,64
45	86,92	8,59	266,90	1,00	0,90	0,47
46	28,58	22,47	1,86	1,00	0,95	1,00
47	1,58	17,02	119,45	1,00	0,98	0,76
48	177,42	31,00	44,82	1,00	0,79	0,91
49	6,19	6,24	96,78	1,00	0,99	0,81
50	19,64	4,89	49,57	1,00	0,98	0,90
51	55,68	15,45	59,50	1,00	0,93	0,88
52	17,01	5,34	37,60	1,00	0,98	0,92
53	25,16	25,48	59,34	1,00	0,95	0,88
54	78,97	394,86	281,81	1,00	0,53	0,44
55	158,11	38,43	111,70	1,00	0,80	0,78
56	313,26	215,58	90,65	1,00	0,47	0,82
57	407,08	144,35	134,24	1,00	0,45	0,73
58	7,65	51,84	145,61	1,00	0,94	0,71
59	87,23	21,29	71,48	0,35	0,89	0,86
Mittelwert	70,71	110,49	168,22	0,98	0,82	0,70
Maximum	407,08	1194,07	991,96	1,00	0,99	1,00
Minimum	1,58	2,67	0,73	0,35	0,00	0,00

PostgreSQL

t [s]	a_{abs} [ms]		p_{abs} [ms]	c [a.u.]	a [a.u.]	p [a.u]
	lesen	schreiben				
0	54,55	23,77	28,28	1,00	0,92	0,94
1	2,63	180,95	17,98	1,00	0,82	0,96
2	1,95	42,92	17,98	1,00	0,96	0,96
3	35,24	41,11	25,77	1,00	0,92	0,95
4	16,85	201,72	8,22	1,00	0,78	0,98
5	2,41	53,77	11,76	1,00	0,94	0,98
6	27,55	146,73	31,63	1,00	0,83	0,94
7	31,30	262,03	0,00	1,00	0,71	1,00
8	27,90	65,39	9,66	1,00	0,91	0,98
9	2,90	525,75	0,00	1,00	0,47	1,00
10	6,44	497,77	10,57	1,00	0,50	0,98
11	1,64	37,34	0,00	1,00	0,96	1,00

12	8,85	496,66	14,46	1,00	0,49	0,97
13	4,92	682,31	13,87	1,00	0,31	0,97
14	36,73	487,89	14,78	1,00	0,48	0,97
15	7,58	126,38	15,53	1,00	0,87	0,97
16	88,34	63,25	22,54	1,00	0,85	0,95
17	31,83	16,87	19,93	1,00	0,95	0,96
18	2,21	190,04	12,36	1,00	0,81	0,98
19	17,84	16,47	10,44	1,00	0,97	0,98
20	5,69	242,45	10,33	1,00	0,75	0,98
21	3,21	198,63	0,00	1,00	0,80	1,00
22	43,14	131,52	11,12	1,00	0,83	0,98
23	5,62	49,24	16,76	1,00	0,95	0,97
24	3,53	13,96	8,10	1,00	0,98	0,98
25	11,01	25,75	11,21	1,00	0,96	0,98
26	118,10	24,29	8,39	1,00	0,86	0,98
27	13,07	16,07	0,51	1,00	0,97	1,00
28	1,78	126,29	17,19	1,00	0,87	0,97
29	2,02	66,62	11,54	1,00	0,93	0,98
30	1,26	16,66	14,66	1,00	0,98	0,97
31	72,69	17,05	13,13	1,00	0,91	0,97
32	1,80	47,05	0,00	1,00	0,95	1,00
33	1,32	12,92	12,18	1,00	0,99	0,98
34	1,80	19,81	12,06	1,00	0,98	0,98
35	3,50	120,51	11,96	1,00	0,88	0,98
36	6,81	54,00	0,00	1,00	0,94	1,00
37	62,92	163,39	0,00	1,00	0,77	1,00
38	1,84	365,60	5,14	1,00	0,63	0,99
39	3,66	104,70	22,31	1,00	0,89	0,96
40	4,33	12,46	51,09	1,00	0,98	0,90
41	67,02	211,84	11,61	1,00	0,72	0,98
42	50,66	9,94	0,00	1,00	0,94	1,00
43	1,98	15,00	10,30	1,00	0,98	0,98
44	2,29	15,85	0,00	1,00	0,98	1,00
45	12,97	17,73	10,58	1,00	0,97	0,98
46	1,65	41,96	0,00	1,00	0,96	1,00
47	1,63	186,93	0,00	1,00	0,81	1,00
48	2,43	56,11	42,00	1,00	0,94	0,92
49	6,87	33,47	45,55	1,00	0,96	0,91
50	8,65	47,66	12,20	1,00	0,94	0,98
51	3,22	69,70	26,16	1,00	0,93	0,95
52	2,30	8,22	15,94	1,00	0,99	0,97
53	17,56	17,28	0,00	1,00	0,97	1,00
54	3,84	8,06	2,39	1,00	0,99	1,00
55	4,06	90,79	0,00	1,00	0,91	1,00

56	19,94	409,68	20,87	1,00	0,57	0,96
57	9,54	21,48	14,24	1,00	0,97	0,97
58	3,83	71,88	0,00	1,00	0,92	1,00
59	1,93	8,51	38,59	1,00	0,99	0,92
Mittelwert	16,69	122,17	12,90	1,00	0,86	0,97
Maximum	118,10	682,31	51,09	1,00	0,99	1,00
Minimum	1,26	8,06	0,00	1,00	0,31	0,90

Literaturverzeichnis

[1] The Standish Group International, Inc., „CHAOS Report 2015“, 2015. Zugegriffen: Juli 26, 2021. [Online]. Verfügbar unter: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

[2] G. Starke, M. Gharbi, A. Koschel, und A. Rausch, *Basiswissen für Softwarearchitekten*, 4., Überarbeitete und Aktualisierte Auflage. Heidelberg: dpunkt.verlag, 2020.

[3] Grafana Labs, „Data Sources“, *Grafana documentation*. https://grafana.com/docs/grafana/latest/datasources/ (zugegriffen Juni 10, 2021).

[4] Grafana Labs, „Grafana Plugins“, *Grafana*. https://grafana.com/grafana/plugins/ (zugegriffen Juni 10, 2021).

[5] S. Gilbert und N. Lynch, „Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services“, *ACM SIGACT News*, Bd. 33, Nr. 2, S. 51–59, Juni 2002, doi: 10.1145/564585.564601.

[6] M. Kleppmann, „A Critique of the CAP Theorem“, *ArXiv150905393 Cs*, Sep. 2015, Zugegriffen: Mai 26, 2021. [Online]. Verfügbar unter: http://arxiv.org/abs/1509.05393

[7] J. Ludewig und H. Lichter, *Software engineering*, 2., Überarb., Aktualisierte und erg. Aufl. Heidelberg: dpunkt.verlag, 2010.

[8] B. Grone, A. Knopfel, und P. Tabeling, „Component vs. Component“, in *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS’05)*, Greenbelt, MD, USA, 2005, S. 550–552. doi: 10.1109/ECBS.2005.26.

[9] Object Management Group, „Unified Modeling Language, v2.5.1“. Dez. 2017. Zugegriffen: Juli 10, 2021. [Online]. Verfügbar unter: https://www.omg.org/spec/UML/2.5.1/About-UML/

[10] The Open Group, „Service-Oriented Architecture – What Is SOA?“ http://www.opengroup.org/soa/source-book/soa/p1.htm#soa_definition (zugegriffen Juni 11, 2021).

[11] E. Wolff, *Microservices*, 2., Aktualisierte Auflage. Heidelberg: dpunkt.verlag, 2018.

[12] VDI-Gemeinschaftsausschuss Industrielle Systemtechnik, *Software-Zuverlässigkeit*. Berlin Heidelberg: Springer-Verlag, 1993. doi: 10.1007/978-3-642-95800-7.

[13] E. Brewer, „Towards Robust Distributed Systems“, gehalten auf der Symposium on Principles of Distributed Computing, Portland, Oregon, Juli 19, 2000. Zugegriffen: Mai 27, 2021. [Online]. Verfügbar unter: https://sites.cs.ucsb.edu/~rich/class/cs293b-cloud/papers/Brewer_podc_keynote_2000.pdf

[14] T. Berners-Lee, R. Fielding, und H. Frystyk, „Hypertext Transfer Protocol -- HTTP/1.0“, RFC Editor, RFC1945, Mai 1996. doi: 10.17487/rfc1945.

[15] L. Lamport, „On interprocess communication“, 1985, Zugegriffen: Mai 27, 2021. [Online]. Verfügbar unter: http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-8.pdf

[16] Sajal Jain und Opstree, „Redis Cluster: Architecture, Replication, Sharding and Failover“, *Medium*. https://medium.com/opstree-technology/redis-cluster-architecture-replication-sharding-and-failover-86871e783ac0 (zugegriffen Juni 29, 2021).

[17] „Redis Sentinel Documentation“, *Redis*. https://redis.io/topics/sentinel (zugegriffen Juni 29, 2021).

[18] „Replication“, *Redis*. https://redis.io/topics/replication (zugegriffen Juni 29, 2021).

[19] E. Brewer, „CAP twelve years later - How the rules have changed“, *Computer*, Bd. 45, Nr. 2, S. 23–29, Feb. 2012, doi: 10.1109/MC.2012.37.

[20] V. Andrikopoulos, S. Strauch, C. Fehling, und F. Leymann, „CAP-Oriented Design for Cloud-Native Applications“, in *Cloud Computing and Services Science*, Bd. 367, I. I. Ivanov, M. van Sinderen, F. Leymann, und T. Shan, Hrsg. Cham: Springer International Publishing, 2013, S. 215–229. doi: 10.1007/978-3-319-04519-1_14.

[21] S. Gilbert und N. Lynch, „Perspectives on the CAP Theorem“, *Lynch Amy Stout*, Feb. 2012, Zugegriffen: Mai 24, 2021. [Online]. Verfügbar unter: https://dspace.mit.edu/handle/1721.1/79112

[22] V. K. Singh, „Eventual Consistency vs Strong Consistency“, *Medium*, März 11, 2019. https://medium.com/system-design-blog/eventual-consistency-vs-strong-consistency-b4de1f92534d (zugegriffen Juni 02, 2021).

[23] G. Zholtkevych, „Metrics for evaluating consistency in distributed datastores“, *Innov. Technol. Sci. Solut. Ind.*, Bd. 0, Nr. 2 (12), S. 40–48, Juni 2020, doi: 10.30837/2522-9818.2020.12.040.

[24] A. Fox und E. A. Brewer, „Harvest, yield, and scalable tolerant systems“, in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, März 1999, S. 174–178. doi: 10.1109/HOTOS.1999.798396.

[25] F5, Inc., „What is a Reverse Proxy vs. Load Balancer?“, *NGINX*. https://www.nginx.com/resources/glossary/reverse-proxy-vs-load-balancer/ (zugegriffen Juni 18, 2021).

[26] Forcepoint, „What is Packet Loss?“ https://www.forcepoint.com/cyber-edu/packet-loss (zugegriffen Juli 18, 2021).

[27] L. Xie, „Relational Databases — Know your Primary Keys!“, *Medium*, Okt. 24, 2019. https://medium.com/swlh/relational-databases-know-your-primary-keys-3897befe9d2 (zugegriffen Juni 18, 2021).

[28] Amazon Web Services, Inc., „Was ist eine Schlüssel-Werte-Datenbank?“ https://aws.amazon.com/de/nosql/key-value/ (zugegriffen Juni 18, 2021).

[29] Jessy Kösterke, „Was sind eigentlich relationale Datenbanken?“, *t3n Magazin*. https://t3n.de/news/eigentlich-relationale-datenbanken-683688/ (zugegriffen Juni 18, 2021).

[30] MariaDB, „ACID Compliance: What It Means and Why You Should Care“, Juli 29, 2018. https://mariadb.com/resources/blog/acid-compliance-what-it-means-and-why-you-should-care/ (zugegriffen Juni 18, 2021).

[31] John Hunter, Darren Dale, Eric Firing, Michael Droettboom, und Matplotlib development team, „Matplotlib“. https://matplotlib.org/ (zugegriffen Juni 25, 2021).

[32] „Redis“. https://redis.io/ (zugegriffen Juni 28, 2021).

[33] PostgreSQL Global Development Group, „PostgreSQL“, *PostgreSQL*, Juni 28, 2021. https://www.postgresql.org/ (zugegriffen Juni 28, 2021).

[34] „Docker“. https://www.docker.com/ (zugegriffen Juni 28, 2021).

[35] The Linux Foundation, „Kubernetes“. https://kubernetes.io/ (zugegriffen Juni 28, 2021).

[36] D. Vohra, *Pro Docker*. Berkeley, CA: Apress, 2016. doi: 10.1007/978-1-4842-1830-3.

[37] O. Geißler, „Was ist Orchestrierung?“, *Datacenter Insider*, Feb. 06, 2018. https://www.datacenter-insider.de/was-ist-orchestrierung-a-683610/ (zugegriffen Juni 30, 2021).

[38] B. Burns, K. Hightower, und J. Beda, *Kubernetes - Eine kompakte Einführung*. 2020. Zugegriffen: Juni 30, 2021. [Online]. Verfügbar unter: https://content-select-com.ezproxy.fh-muenster.de/de/portal/media/view/5f4a4a42-5278-4349-a717-0695b0dd2d03?forceauth=1

[39] „Git“. https://git-scm.com/ (zugegriffen Juli 10, 2021).

[40] Red Hat, Inc., „What is Infrastructure as Code (IaC)?“ https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac (zugegriffen Juli 23, 2021).

[41] The Linux Foundation, „Service“, *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/ (zugegriffen Juli 12, 2021).

[42] The Linux Foundation, „Kubernetes Documentation“, *Kubernetes*. https://kubernetes.io/docs/home/ (zugegriffen Juli 12, 2021).

[43] Helm Authors, „Helm“. https://helm.sh/ (zugegriffen Juli 12, 2021).

[44] PostgreSQL Global Development Group, „28.2. The Statistics Collector“, *PostgreSQL Documentation*, Mai 13, 2021. https://www.postgresql.org/docs/10/monitoring-stats.html (zugegriffen Juli 21, 2021).

[45] „Redis Keyspace Notifications“, *Redis*. https://redis.io/topics/notifications (zugegriffen Juli 21, 2021).

[46] A. Layfield, *DandyDeveloper/charts/charts/redis-ha*. 2021. Zugegriffen: Juni 28, 2021. [Online]. Verfügbar unter: https://github.com/DandyDeveloper/charts/tree/master/charts/redis-ha

[47] Technische Hochschule Köln / Campus Gummersbach, „BASE“, *Datenbanken Online Lexikon*. https://wikis.gm.fh-koeln.de/Datenbanken/BASE (zugegriffen Juli 22, 2021).

[48] *bitnami/charts/bitnami/postgresql-ha*. Bitnami, 2021. Zugegriffen: Juni 28, 2021. [Online]. Verfügbar unter: https://github.com/bitnami/charts/tree/master/bitnami/postgresql-ha

[49] The Pgpool Global Development Group, „Load Balancing“, *pgpool-II 4.2.3 Documentation*. https://www.pgpool.net/docs/latest/en/html/runtime-config-load-balancing.html (zugegriffen Juni 29, 2021).

[50] The Pgpool Global Development Group, „Failover and Failback“, *pgpool-II 4.2.3 Documentation*. https://www.pgpool.net/docs/latest/en/html/runtime-config-failover.html (zugegriffen Juni 29, 2021).

[51] Python Software Foundation, „Welcome to Python.org“, *Python.org*. https://www.python.org/ (zugegriffen Juli 21, 2021).

[52] Prometheus, „Prometheus - Monitoring system & time series database“, *Prometheus*. https://prometheus.io/ (zugegriffen Juli 26, 2021).

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen und ist noch nicht veröffentlicht worden. Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Münster, den 2. August 2021