Institut für Parallele und Verteilte Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3364

Konzepte und Algorithmen zur Datensynchronisation mit Cloud-Datenzentren

Paul Hummel

Studiengang: Softwaretechnik

Prüfer: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer: Dipl.-Inform. Stephan Schuhmann, Dr. rer. nat. Frank Dürr

Beginn am: 15.06.2012
Beendet am: 15.12.2012

CR-Nummer: D.2.7, E.5, H.2.4, H.3.4

Kurzfassung

Cloud Computing überzeugt immer mehr durch seine Vorteile, wie z. B. Verfügbarkeit, Skalierbarkeit und Kosteneffizienz. Ein aktuelles Forschungsthema ist die Synchronisation der Daten mit Cloud-Rechenzentren. In dieser Arbeit betrachten wir daher Aspekte, die bei diesem Problem im Mittelpunkt stehen. Zuerst werden in dieser Arbeit theoretische Grundlagen und aktuelle Technologien untersucht. Anschließend wird das hier betrachtete Synchronisations-Problem genauer beschrieben und potentielle Lösungsansätze vorgestellt. Das Ziel ist es, die lokalen Datenbanksysteme durch den Einsatz des Cloud Computings überflüssig zu machen und dabei die Vorteile des lokalen Datenbanksystems zu behalten. Es werden eine Architektur und Mechanismen entwickelt, ein Ausschnitt der Lösung implementiert und evaluiert. Abschließend wird die Arbeit zusammengefasst und ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis IV

1 Einleitung 1
1.1 Motivation . . . 1
1.1.1 Datensynchronisierung . . . 1
1.1.2 Cloud Computing . . . 3
1.2 Aufgabenstellung . . . 6
1.2.1 Ziele . . . 6
1.2.2 Abgrenzung zu verwandten Arbeiten . . . 7
1.3 Überblick über das Dokument . . . 8

2 Verwandte Arbeiten 9
2.1 Generelle Lösungsansätze zur Erlangung von Konsistenz . . . 9
2.1.1 Locks . . . 9
2.1.2 ACID-Transaktionen . . . 10
2.1.3 Read-Write-Quorums . . . 10
2.1.4 Last Write Wins . . . 12
2.1.5 Zusammenfassung . . . 12
2.2 Konkrete Lösungen aus der Praxis . . . 12
2.2.1 Rsync . . . 13
2.2.2 Rumor . . . 13
2.2.3 Roam . . . 14
2.2.4 Coda . . . 15
2.2.5 Bayou . . . 16
2.2.6 SyncML . . . 17
2.2.7 BASE . . . 18
2.2.8 IceCube . . . 19
2.2.9 Zusammenfassung . . . 21
2.2.10 Bewertung . . . 22

3 Systemmodell und Problembeschreibung 24
3.1 Infrastruktur und Komponenten . . . 24
3.1.1 Knoten . . . 24
3.1.2 Cloud . . . 26
3.1.3 Netzwerk . . . 26
3.2 Anforderungen an das Zielsystem . . . 27
3.2.1 Beispielanwendung . . . 27
3.2.2 Funktionale Anforderungen . . . 29
3.2.3 Nichtfunktionale Anforderungen . . . 30
3.3 Zusammenfassung . . . 31

4 Entwurf **33**
4.1 Systemarchitektur . . . 33
4.1.1 Cloud . . . 33
4.1.2 Cache . . . 34
4.2 Algorithmen und Mechanismen . . . 35
4.2.1 Grundidee . . . 35
4.2.2 Funktionale Anforderungen . . . 36
4.2.3 Nichtfunktionale Anforderungen . . . 39
4.2.4 Zusammenfassung . . . 47
4.3 Operations-Komponente . . . 49
4.3.1 Datensatz . . . 50
4.3.2 Operation . . . 51
4.3.3 Log . . . 57
4.3.4 Mergefunktion . . . 57
4.3.5 Qualitätsstufen der Operationen . . . 61
4.3.6 Integration der Operations in Architekturmuster . . . 65
4.3.7 Zusammenfassung . . . 67

5 Implementierung **68**
5.1 Architektur . . . 68
5.1.1 System . . . 68
5.1.2 Komponenten . . . 70
5.2 Verwendete Entwurfsmuster . . . 71
5.2.1 Singleton . . . 71
5.2.2 Proxy . . . 71
5.2.3 Abstract-Factory . . . 72
5.2.4 Command . . . 72
5.2.5 Memento . . . 73
5.2.6 Template Method . . . 73
5.2.7 Mediator . . . 73
5.3 Schnittstellen . . . 74
5.3.1 IStorage . . . 74
5.3.2 ICloudSync . . . 74
5.3.3 IPeerSync . . . 74
5.3.4 IMerge . . . 75
5.3.5 IOperation . . . 75
5.3.6 IExecuteOperation . . . 75
5.4 Abläufe . . . 76
5.4.1 Operationsaufruf . . . 76
5.4.2 Synchronisation mit der Cloud . . . 77
5.4.3 Synchronisation über Peer-to-Peer . . . 77
5.4.4 Konfliktauflösung . . . 78

5.5 Beispielanwendung 78
5.5.1 Aufbau 79
5.5.2 Anwendungsfälle 80
5.5.3 Hintergrundinformationen 82
5.6 Zusammenfassung 84

6 Evaluierung 85
6.1 Setup 85
6.1.1 Evaluationsumgebung 85
6.1.2 Evaluationsparameter 86
6.1.3 Finanzielle Kosten 87
6.2 Szenarien 88
6.3 Auswertung 91
6.3.1 Kosten im Szenario 1H 1D 91
6.3.2 Kostensenkung durch die Optimierungsfunktion 92
6.3.3 Kostenersparnis 95
6.3.4 Kosten für den Ausfall der Cloud 99
6.3.5 Fazit der Evaluation 100
6.4 Zusammenfassung 100

7 Fazit 102
7.1 Zusammenfassung 102
7.2 Ausblick auf weitere Arbeiten 104

Literaturverzeichnis VIII

Abbildungsverzeichnis

1 Suchanfragehäufigkeiten für den Begriff Cloud Computing 2
2 Eine Statistik für die Suchanfragen von zwei großen Cloud-Anbietern 2
3 Gegenüberstellung der Kostenverteilung verschiedener Paradigmen . . 5
4 Anschauliche Darstellung des Read-Write-Quorums 11
5 Datenmengenabdeckung bei einer selektiven Synchronisation. 14
6 Darstellung einer möglichen Aufteilung der Server und Clients für das Dateisystem Coda 16
7 Umgebungen und Komponenten des Systems 25
8 Grobarchitektur des Zielsystems 34
9 BASE-Eigenschaften abgebildet auf das CAP-Theorem 36
10 Aufbau der Struktur Anwendung-Operations-Daten 49
11 Parameter einer Operation 52
12 Hierarchie von Operationen 54
13 Ablauf der Konfliktauflösung auf dem Knoten 60
14 Erste Stufe des Peer Merge-Algorithmus 64
15 Eingliederung der Operations in Drei-Schichten und MVVM 66
16 Feinarchitektur des Systems 69
17 Command-Entwurfsmuster 72
18 Bildschirmfoto der Endanwendungen und Demonstrationskonsolen . . 79
19 Demonstration der Implementierung - Anwendungsfall 1 80
20 Demonstration der Implementierung - Anwendungsfall 2 81
21 Demonstration der Implementierung - Anwendungsfall 3 82
22 Demonstration der Implementierung - Anwendungsfall 4 83
23 Zusammengeführter Log nach simultaner Bearbeitung eines Rezeptes 83
24 Darstellung der Evaluationskriterien 89
25 Kosten für die Synchronisation eines Caches ohne Optimierung 90
26 Kostenverteilung der generischen Kosten-Funktion 93
27 Kostenverteilung mit und ohne Optimierung 94
28 Kostenersparnis abhängig von der Lese-/Schreibrate 97
29 Kostenersparnis abhängig von der Datenmenge 98
30 Kostenersparnis abhängig von den Kosten für das Lesen eines veralteten Datensatzes 99

1 Einleitung

Seit 2007 wird das neue Paradigma „Cloud Computing“ immer beliebter. Mit Cloud Computing lassen sich IT-Infrastrukturen dynamisch über das Netzwerk bereitstellen. Die Leistungen können jederzeit an den momentanen Nutzungsbedarf angepasst werden. Auch bei der Verwendung von Cloud Computing müssen verteilte Daten synchronisiert werden, jedoch hat Cloud Computing gewisse Vorteile, sodass man das Problem der Synchronisation unter einem anderen Blickwinkel betrachten kann und somit die Möglichkeit hat, bessere Synchronisierung zu realisieren.

Im nächsten Abschnitt beschäftigen wir uns mit der Datensynchronisierung und dem Cloud Computing - den Themen, die die Motivation dieser Arbeit darstellen. Dabei klären wir den Begriff „Cloud Computing“, diskutieren, welche Vor- und Nachteile das Cloud Computing mit sich bringt und welche Rolle es in dieser Arbeit spielt. Anschließend wird die Aufgabenstellung dieser Arbeit beschrieben. Am Ende des Kapitels folgt ein Überblick über das gesamte Dokument.

1.1 Motivation

Cloud Computing liegt aktuell im Trend für die Bereitstellung der IT-Infrastrukturen. Die Abbildung 1 stellt die Nutzerstatistik der Google-Suchfunktion bezüglich des Begriffs Cloud Computing dar. Man erkennt, dass ab September 2007 das Interesse an Cloud Computing stetig wuchs, bis der Höhepunkt im März 2011 erreicht wurde. In Abbildung 2 lässt sich feststellen, dass das Interesse an Cloud Computing weiterhin wächst. Statistisch äußert es sich jedoch durch Suchanfragen für konkrete Cloud-Anbieter.

Immer mehr Firmen lagern ihre Dienste und Daten in die Cloud aus und oft muss ein Teil dieser Daten auch lokal vorhanden sein. Entsprechend ist Datensynchronisierung ein wichtiges Thema, auf das im Folgenden eingegangen wird.

1.1.1 Datensynchronisierung

Die Datensynchronisierung wird zwangsweise dann benötigt, wenn gleiche Daten über das Netzwerk an mehreren Stellen gleichzeitig verteilt (d. h. repliziert) sind

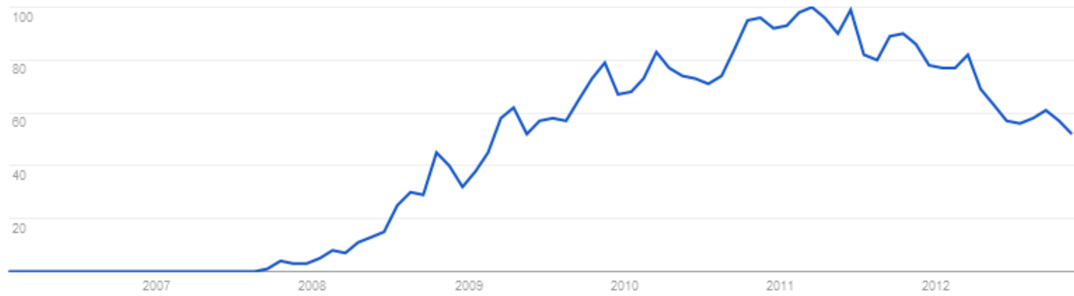

Abbildung 1: Suchanfragehäufigkeiten für den Begriff Cloud Computing. Die Suchmaschine Google erstellt eine Statistik zu einem gegebenen Begriff.

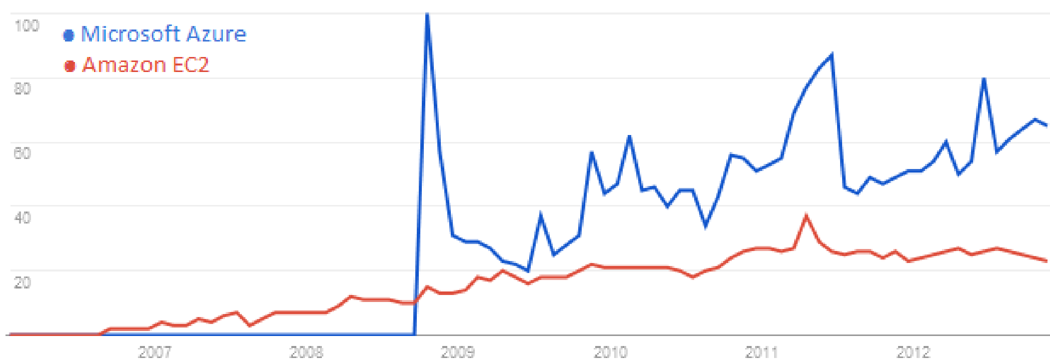

Abbildung 2: Eine Statistik für die Suchanfragen von zwei großen Cloud-Anbietern. Suchanfragehäufigkeiten für Begriffe Microsoft Azure und Amazon EC2.

und an einzelnen Stellen geändert werden, wodurch inkonsistente Datenzustände entstehen können.

> „Leider muss die Replikation von Daten mit der Einschränkung erkauft werden, dass es zu Konsistenzproblemen kommen kann, sobald mehrere Kopien vorhanden sind.“ [TS08]

Wird eine Datenkopie geändert, dann müssen alle anderen Replikas die Änderung durchführen, sonst divergiert die Konsistenz und die darauffolgenden Leseoperationen der verschiedenen Replikas ergeben unterschiedliche Ergebnisse.

> „Daher sollte eine Aktualisierung, die an einer Kopie vorgenommen wurde, an alle Kopien weitergeleitet werden, bevor eine weitere Operation

> stattfinden kann.“ [TS08]

Das Problem der Datensynchronisation besteht schon länger und es wurden einige Algorithmen entwickelt, die das Problem weitgehend einschränken. So existiert das verteilte Dateisystem Coda, bei dem Daten zwischen den verteilten Rechnern synchronisiert werden. IceCube ist eine Entwicklung, die Synchronisationskonflikte automatisch auflöst. Dennoch existiert neben den einzelnen Entwicklungen ein großer Forschungsraum für das Synchronisierungsproblem.

Wie in jedem verteilten System kann es zu Verbindungsausfällen kommen, welche zu der Unerreichbarkeit einiger Knoten führen können. Dieses konkrete Problem der Synchronisierung ist in der Fachliteratur als „Netzwerkpartitionierung“ bekannt. In dieser Arbeit wird ein besonderes Augenmerk darauf gelegt, die Auswirkungen der Netzwerkausfälle einzuschränken.

Bevor die Details der Synchronisierungsmöglichkeiten in Kapitel 2 diskutiert werden, beschäftigen wir uns mit dem Paradigma Cloud Computing und seinen Eigenschaften.

1.1.2 Cloud Computing

Definition

Bislang existieren nur frühe Definitionen für das Cloud Computing. So definieren Wang et al. das Cloud Computing folgendermaßen:

> „A computing Cloud is a set of network enabled services, providing scalable, QoS guaranteed, normally personalized, inexpensive computing platforms on demand, which could be accessed in a simple and pervasive way.“ [WTK+08]

Eine alternative Definition des Cloud Computings wurde von Baun et al. [BKNT10] wie folgt vorgeschlagen:

> „Unter Ausnutzung virtualisierter Rechen- und Speicherressourcen und moderner Web-Technologien stellt Cloud Computing skalierbare, Netzwerk-zentrierte, abstrahierte IT-Infrastrukturen, Plattformen und Anwendungen als on-demand Dienste zur Verfügung. Die Abrechnung dieser Diens-

> te erfolgt nutzungsabhängig.“ [BKNT10]

Foster et al.[FZRL08] definieren den Begriff wie folgt:

> „A large-scale distributed computing paradigm that is driven by economies of scale, in which a pool of abstracted, virtualized, dynamically-scalable, managed computing power, storage, platforms, and services are delivered on demand to external customers over the Internet.“[FZRL08]

Aus den oben genannten Definitionen lassen sich folgende Haupteigenschaften ableiten, die das Cloud Computing auszeichnen:

- Netzwerkbasiert und verteilt
- Skalierbar
- Virtualisierte Software
- Garantierte QoS
- Günstige und leicht mietbare Hardware
- Kosten sind Nutzungsabhängig
- Plattformen, Rechenleistung und Speicher als Vertragsleistung entsprechend SLA

Um das Bild zu vervollständigen, muss auch das Gegenstück zu Cloud Computing erwähnt werden. Es ist die lokale Ausführung der Dienste und wird als „on-premise“ bezeichnet. Dabei ist die Hardware, auf der Dienste ausgeführt werden, vor Ort fest installiert. Die Basis- und laufenden Kosten entstehen hauptsächlich durch die Erstanschaffung, Administrierung und Wartung. Die Quality of Service hängt von der jeweiligen Systemkonfiguration ab, die vom Administrator erstellt wird. Die Software kann virtualisiert werden, jedoch fehlt dem Systembetreiber die Möglichkeit, schnell zusätzliche Ressourcen hinzuzuschalten und somit die Skalierbarkeit zu ermöglichen, um die Vorteile der Virtualisierung auszuschöpfen.

Die Flexibilität ist nur einen Vorteil des Cloud Computings. Weitere Vorzüge dieses Paradigmas sind im Folgenden beschrieben.

Vorteile

Aus den genannten Haupteigenschaften des Cloud Computings lassen sich folgende

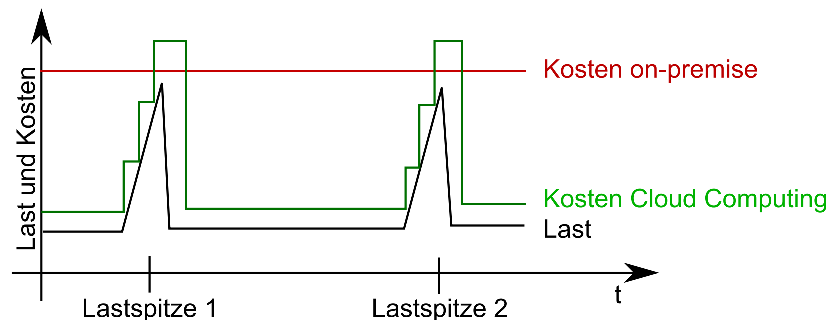

Abbildung 3: Gegenüberstellung der Kostenverteilung verschiedener Paradigmen. Mit Cloud Computing lässt sich eine Kostensenkung erzielen.

Vorteile für den Cloud-Mieter feststellen. Mit Hilfe der Virtualisierung, kann die Änderung der Hardwarekonfiguration auf eine effiziente Weise erfolgen, sodass das System jederzeit flexibel bleibt. Die Infrastruktur und Mechanismen der Cloud unterstützen diese Möglichkeit und somit kann das System entsprechend der Konfigurationsänderung skalieren. Dies ermöglicht einem Cloud-Mieter eine Kostensenkung, z. B. indem die Hardware-Konfiguration abhängig von der Last des Systems geändert wird (Abbildung 3). Allgemein kann ein geringer administrativer Aufwand für den Cloud-Mieter verzeichnet werden, denn das übernimmt der Cloud-Anbieter. Es gibt keine Anschaffungskosten, denn Kosten fallen nur für die Miete an und werden nur für die genutzten Ressourcen berechnet. Das Mieten der Ressourcen erfolgt dabei mit einem Mausklick. Cloud Computing überzeugt durch die hohe Verfügbarkeit, die dem Cloud-Mieter vertraglich zugesichert wird.

Nachteile

Diesen Vorteilen müssen allerdings einige Nachteile entgegengesetzt werden. Das wichtigste Thema ist die Absicherung der Daten vor Zugriffen Dritter. Aus diesem Grund müssen laut [BC+08] und [FZRL08] mit dem Cloud Computing-Anbieter folgende Punkte geklärt werden:

1. Wer den Zugriff auf gespeicherte Daten hat?
2. Wurde der Cloud-Anbieter auf sicheren Umgang mit fremden Daten geprüft?
3. In welchem Land werden Daten gespeichert?
4. Wie stark werden Daten von Daten anderer Kunden getrennt?

5. Wie werden Daten repliziert und im Fehlerfall wiederhergestellt?
6. Welche Unterstützung bietet der Cloud-Anbieter den Ermittlern falls eine illegale Aktivität stattfindet?
7. Welche Auswirkungen hat die Übernahme des Cloud-Anbieters auf die Verfügbarkeit der Daten?

Des Weiteren sollte man bedenken, dass man von jedem beliebigen Internetanschluss aus, alleine mit dem Administratorenpasswort des Cloud-Mieters alle Daten seines Cloudsystems auslesen kann. Aber auch der Aspekt der Datenfreigabe soll berücksichtigt werden. So muss entschieden werden, ob die sensiblen Daten in der Cloud wirklich sicher sind und ausgelagert werden dürfen. Ob die Cloud für sensible Daten geeignet ist, lässt sich generell nicht entscheiden und hängt vom jeweiligen Cloud-Anbieter ab. Eine allgemeine Empfehlung ist es, die Daten in der Cloud mit einem unabhängigen System und Passwort zu verschlüsseln (vgl. [Rei01]) und sensible Daten lokal zu halten.

In diesem Abschnitt wurden die Motivation dieser Arbeit und einige Details zu den hier verwendeten Begriffen erörtert. Im nächsten Abschnitt wird die konkrete Aufgabenstellung dieser Arbeit präsentiert.

1.2 Aufgabenstellung

Die Aufgabenstellung besteht aus bestimmten Zielen, die im Folgenden formuliert sind. Dabei wurden einige Schwerpunkte gesetzt, um die wichtigsten Punkte der Arbeit tiefgehend zu bearbeiten.

1.2.1 Ziele

Das Ziel dieser Arbeit ist es, bestehende Konzepte und Algorithmen zu untersuchen und neue zu entwickeln, sodass der lokale Datenzugriff und die lokale Datenspeicherung ohne Verwendung einer lokalen Datenbank realisiert werden und dabei die Vorteile dieser sichergestellt werden, wie z. B. Zuverlässigkeit, Leistung und Offlinetauglichkeit.

Es sollen eine Architektur und einige Mechanismen entworfen, sowie der Konsis-

tenzbegriff definiert werden. Es sollen optimierte Methoden für den Zugriff, die Bereitstellung und die Synchronisierung der Daten entworfen, implementiert und evaluiert werden. Einerseits sollen Algorithmen die Konsistenz der Daten sichern, andererseits sollen Kosten für das Speichern der Daten in der Cloud und die Kommunikation zwischen der Cloud und on-premise-Komponenten entsprechend üblichen Kostenmodellen minimiert werden.

Dabei sollen folgende Punkte bearbeitet werden:

1. Untersuchen der verwandten Arbeiten aus dem Bereich Datenverwaltung und Cloud Computing
2. Analysieren und Definieren des Konsistenzbegriffs und der Optimierungsziele
3. Entwerfen des Systems
4. Entwerfen der Algorithmen für die Synchronisierung entsprechend dem Konsistenzbegriff und den Optimierungszielen
5. Implementieren und Auswerten des erstellten Ansatzes, bzw. eines gewählten Ausschnitts (Zielumgebung: Windows Azure und Microsoft SQL Server)

Schwerpunkte

Der grundlegende Schwerpunkt dieser Arbeit liegt auf der Entwicklung der Grundidee, sowie des Entwurfs eines Systems, das Daten mit der Cloud synchronisiert, speichert und für lokale Endanwendungen verfügbar macht. Ein weiterer Schwerpunkt ist es, den finanziellen Aufwand für die Datenübertragung zu reduzieren.

Im Folgenden wird das zu erstellende System von anderen abgegrenzt, um die Ziele zu konkretisieren.

1.2.2 Abgrenzung zu verwandten Arbeiten

Viele Ansätze enthalten bereits komplexe Synchronisierungsmechanismen, jedoch nutzen sie die Möglichkeiten des neuen Paradigmas Cloud Computing nicht, das viel Potential für die Optimierung der Ressourcennutzung und der Abläufe bietet.

In dieser Arbeit werden im Gegensatz zu den im Abschnitt 2 vorgestellten Algorithmen besondere Eigenschaften des Cloud Computings genutzt und die Ansätze

der manchen hier beschriebenen Mechanismen so miteinander verzahnt, dass eine bessere Synchronisation der Daten ermöglicht wird. Die beschränkte, aber hohe und durch SLA garantierte Verfügbarkeit der Cloud spielt eine wichtige Rolle bei der Wahl, der Zusammensetzung und der Entwicklung von Synchronisationsmethoden. Zusammen mit dem sicheren Speicher in der Cloud ergibt sich eine zuverlässige zentrale Stelle im System, die Daten für alle Teilnehmer bereitstellt. Dies schafft Möglichkeiten für die Neuentwicklung der Algorithmen aus dieser Perspektive. In der bisherigen Sichtweise gelten zentrale Stellen aufgrund der Single-Point-of-Failure-Problematik als unsicher. Die Alternative stellen Peer-to-Peer-Systeme dar, bei denen die Konsistenzerhaltung der Daten aufgrund des hohen Verteiltheitsgrades ein Problem darstellt, das nur eingegrenzt werden kann. Mit dieser Arbeit eröffnen sich neue Möglichkeiten, Daten konsistent und verfügbar zu machen.

Der nächste Abschnitt präsentiert den Aufbau des Dokuments und Hauptthemen der nächsten Kapitel.

1.3 Überblick über das Dokument

Nachdem ein Überblick über bestehende Systeme im Kapitel 2 gegeben wird, beschäftigt sich das darauffolgende Kapitel 3 mit der Konkretisierung der Aufgaben des Produkts, sowie der Einordnung der Lösung in die Systemumgebung. Anschließend wird das System im Kapitel 4 entworfen und die Komponenten werden einzeln beleuchtet. Danach wird im Kapitel 5 auf die Besonderheiten und Details der Implementierung eingegangen. Die Mechanismen des Systems werden dann im Kapitel 6 ausgewertet und es wird die Kosteneffizienz des Systems untersucht. Abschließend werden alle Ergebnisse, Ideen und andere wichtigen Aspekte zusammengefasst und ein resultierendes Fazit im Kapitel 7 gezogen.

2 Verwandte Arbeiten

In diesem Kapitel werden verwandte Arbeiten, aus dem gleichen Forschungsbereich wie diese Arbeit, beschrieben. Er dient dafür, die existierenden Mechanismen vorzustellen, um den aktuellen Stand der Forschung zu vermitteln. Viele Algorithmen werden in dieser Lösung vollständig oder teilweise eingesetzt.

Zunächst werden generelle Lösungsansätze beschrieben, die in der Theorie die Grundlage für die praktischen Lösungen bilden. Anschließend werden die konkreten Ausprägungen der Algorithmen, die in der Praxis eingesetzt werden, vorgestellt.

2.1 Generelle Lösungsansätze zur Erlangung von Konsistenz

Im vorausgehenden Kapitel wurde die Herausforderung eines Konflikts deutlich gemacht. Konflikte müssen entweder verhindert oder aufgelöst werden. Dies kann auf verschiedene Weisen geschehen. Einige Strategien, mit denen die Konsistenz erhalten werden kann, sind: Locks, ACID-Transaktionen und Read-Write-Quorums. Die bereits entstandenen Konflikte können mit Last Write Wins aufgelöst werden. Im Folgenden werden diese Algorithmen einzeln vorgestellt, sowie die Vor- und Nachteile erörtert.

2.1.1 Locks

Hierbei wird ein Datensatz vor seiner Bearbeitung von einem Prozess mit einem Lock versehen. Nur dieser Prozess hat dann die Berechtigung den Datensatz zu bearbeiten. Anschließend wird das Lock aufgehoben und der Datensatz für das Sperren vom gleichen oder einem anderen Prozess freigegeben.

Es gibt zwei Arten von Locks: Write-Locks und Read-Write-Locks. Bei einem Write-Lock ist nur der lesende Zugriff für andere Prozesse erlaubt und bei einem Read-Write-Lock dürfen andere Prozesse weder lesend noch schreibend auf den Datensatz zugreifen.

Der Vorteil liegt in der Eindeutigkeit des Zugriffsberechtigten. Es werden Konflikte vermieden, indem erst geprüft wird, ob der Zugriff auf den Datensatz erlaubt ist. Andererseits entstehen Wartezeiten für andere Prozesse, bis der Datensatz wieder

freigegeben wird. Diese Wartezeit kann sich auf die Gesamtleistung des betroffenen Systems auswirken, wenn Programmabläufe mit dem betroffenen Datensatz kausal zusammenhängen.

2.1.2 ACID-Transaktionen

ACID ist eine Abkürzung für Atomicity, Consistency, Isolation, Durability. Es ist ein Verfahren, um Datenzugriffe auf mehreren verteilten Rechnern atomar auszuführen und somit die Datenkonsistenz im System zu erhalten. Die Bedeutung der einzelnen Aspekte ist wie folgt [Pri08]:

- Atomicity - Das alles-oder-nichts-Prinzip, Änderungen werden entweder übernommen, oder verworfen. Es ist wichtig, um die Transaktionslogik herzustellen.
- Consistency - die Integritätsbedingungen der Datenbank werden vor und nach der Transaktion aufrecht erhalten.
- Isolation - Abgrenzung der einzelnen Operationen an der Datenbank, sodass sie sich nicht beeinflussen.
- Durability - Daten werden nach dem Abschluss einer Transaktion dauerhaft gespeichert und Änderungen werden nicht rückgängig gemacht.

Zunächst wird ein Abkommen zwischen allen Beteiligten getroffen, ob eine Transaktion durchgeführt wird. Erst wenn alle zustimmen, wird sie von allen Beteiligten durchgeführt. Bei mindestens einer Gegenstimme wird die Transaktion von keinem Beteiligten durchgeführt.

Der Nachteil bei diesem verfahren ist, dass alle Beteiligten im System verfügbar sein müssen. Der Ausfall eines Beteiligten verursacht bei allen anderen Wartezeiten.

2.1.3 Read-Write-Quorums

Bei diesem Verfahren entscheidet die Mehrheit der Replikas eine gemeinschaftliche Wahl, die sich auf alle Teilnehmer auswirkt. Dabei gibt es folgende Entscheidungskriterien, um Read- und Write-Operationen zu vollziehen: „Majority“, „Read One Write All“. Bei Majority müssen mindestens $\frac{N}{2} + 1$ Knoten jeder Operation zustim-

men. So muss bei einer Write-Operation die Mehrheit die gleiche Datenänderung durchführen. Geben bei einer Read-Operation nicht alle $\frac{N}{2} + 1$ Knoten die Gleiche Datensatzversion zurück, dann muss es eine andere Version geben, die mehr als die Hälfte aller Knoten hat. Bei Read One Write All muss jede Read-Operation nur einen Knoten involvieren, dagegen eine Write-Operation die Gesamtheit.

Die Quorums sind in Abbildung 4 visualisiert.

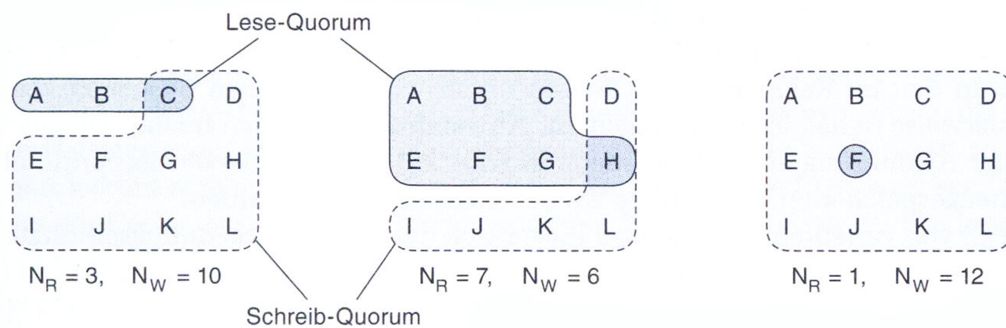

Abbildung 4: Anschauliche Darstellung des Read-Write-Quorums. Die einzelnen Replikas sind als Buchstaben dargestellt. Mit den Blöcken, die Replikas gruppieren, sind die Stimmen für das Lesen und Schreiben dargestellt. Jede Replika hat nur eine Stimme. [TS08]

Der Vorteil ist, dass dieses Verfahren ein gerechtes Lesen und Schreiben in einem dezentralen Speicher ermöglicht, ohne Konflikte zu verursachen.

Der Nachtteil dabei ist, dass für ein Write und ein Read mehr als die Hälfte aller Replikas kontaktiert werden müssen. In einem System mit vielen Replikas verursacht dieses Vorgehen einen enormen Kommunikationsaufwand. Des Weiteren muss Wissen darüber vorliegen, wie viele Knoten, die momentan nicht verfügbar sind, am System teilnehmen, wenn sie wieder online gehen und welche Dateiversion sie haben. Liegt dieses Wissen nicht vor, dann kann nicht beurteilt werden, ob ein ausreichender Prozentsatz der Stimmen abgegeben wurde, um eine Operation durchzuführen. In einem statischen System kann dieses Wissen mit Hilfe einer Registrierungsroutine erlangt werden. Im hochdynamischen System ist dieses Wissen dagegen nicht greifbar, deshalb kann dieser Algorithmus nicht eingesetzt werden. Es gibt auch einen weiteren Nachteil: Ist mehr als die Hälfte aller Knoten nicht verfügbar, dann kann entweder nicht geschrieben oder nicht gelesen werden. Somit kann der verteilte

Speicher nur eingeschränkt verwendet werden.

2.1.4 Last Write Wins

Dies ist wohl die einfachste Strategie, bei der allerdings vorausgesetzt wird, dass Uhren der Teilnehmer synchron sind. Es werden die Zeitstempel der letzten Änderung der Einträge verglichen. Dabei „gewinnt" der jüngste Eintrag. Der Vorteil ist, dass eine einfache Logik benötigt wird, um die Auflösung zu vollziehen. Andererseits wird dabei der ältere Eintrag völlig ignoriert. In den meisten Systemen hat jeder Eintrag eine lokale Wirksamkeit und globale Bedeutung und darf somit nicht ignoriert werden.

2.1.5 Zusammenfassung

Es wurden die theoretischen Lösungsansätze vorgestellt. Die Auflistung offenbart, dass sie aufgrund gewisser Nachteile das Problem der Synchronisation nicht lösen, sondern nur eingrenzen. Im nächsten Abschnitt werden die wichtigsten praktischen Mechanismen präsentiert, die teilweise auf diesen Mechanismen aufbauen. Sie setzen komplexe Logik ein, um das Problem noch weiter einzugrenzen.

2.2 Konkrete Lösungen aus der Praxis

In diesem Unterkapitel werden verschiedene Synchronisierungsalgorithmen vorgestellt. Die Arbeiten sind nach ihrer Komplexität aufsteigend sortiert. Zuerst behandeln wir die einfachen Algorithmen, bei denen der Datenaustausch nur eine 1:1-Beziehung voraussetzt. Danach werden Peer-to-Peer- und Client-Server-basierten Dateisysteme beleuchtet. Abschließend untersuchen wir komplexe Algorithmen, die über eine innovative Konfliktauflösung mittels Logs verfügen. Es werden folgende Verfahren beleuchtet: Rsync, Rumor, Roam, Coda, Bayou, SyncML, BASE und IceCube. Anschließend werden ein Überblick und eine Bewertung dieser Verfahren gegeben.

2.2.1 Rsync

Der Rsync-Algorithmus kann zwei Dateien miteinander synchronisieren. Es ist ein Client-Server-System, bei dem Dateiteile vom Client auf den Server übertragen werden. Dabei wird bei einer Änderung nicht die vollständige Datei nochmals übertragen, sondern nur der geänderte Teil. Um dies zu erreichen, wird die zu übertragene Datei in Blöcke aufgeteilt und die Checksummen der Blöcke werden berechnet. Anschließend werden die Checksummen der Datei auf dem Client mit den Checksummen der Datei auf dem Server abgeglichen. Unterschiedliche Blöcke werden ausgetauscht. [TM96]

Bei Rsync handelt es sich um einen grundlegenden Algorithmus, bei dem, aufgrund der 1:1-Beziehung und der unidirektionalen Kommunikation (vom Client zum Server) keine Konflikte entstehen und somit aufgelöst werden müssen.

2.2.2 Rumor

Rumor ist ein Algorithmus, um Dateien zwischen mehreren Rechnern mit Hilfe der Peer-to-Peer-Kommunikation zu synchronisieren. Ist eine Datenkopie, auch Replika genannt, nicht erreichbar, dann werden Daten trotzdem mit anderen Replikas über Peer-to-Peer-Verbindungen abgeglichen. Für das Vergleichen und Aktualisieren der Daten verschiedener Replikas werden Versionsvektoren verwendet. Sobald die ausgefallene Replika wieder verfügbar wird, werden Datenaktualisierungen auch dort übernommen. Sollte es Konflikte beim Synchronisieren geben, z. B. wenn eine Datei an zwei Replikas gleichzeitig geändert wurde, dann wendet Rumor einen automatischen Mergealgorithmus[1] an, der für wenige Dateitypen bereitsteht. Es gibt eine Schnittstelle, um weitere Merge-Algorithmen hinzuzuschalten. Steht kein Merge-Algorithmus für eine konfliktbehaftete Datei bereit, dann kann der Konflikt nicht automatisch aufgelöst werden. In diesem Fall werden beide Dateiversionen beibehalten und der Benutzer muss den Konflikt manuell auflösen.[Rei] [GRR+99]

Rumor unterstützt auch selektive Synchronisation. Das heißt, dass Replikas nicht

[1] Ein Mergealgorithmus erhält als Eingabe verschiedene Versionen des konfliktbehafteten Datensatzes und gibt einen konfliktfreien Datensatz zurück. Der Vorgang ist in der englischen Literatur als „reconciliation“ bekannt. In einem verteilten System mit der lockeren Konsistenz können Konflikte oft auftreten, aus diesem Grund ist das Thema reconciliation von besonderer Bedeutung für diese Arbeit.

die gleiche Dateimenge abdecken müssen, sondern dass auch Teile der Dateimenge synchron gehalten werden können (Abbildung 5). [GRR+99]

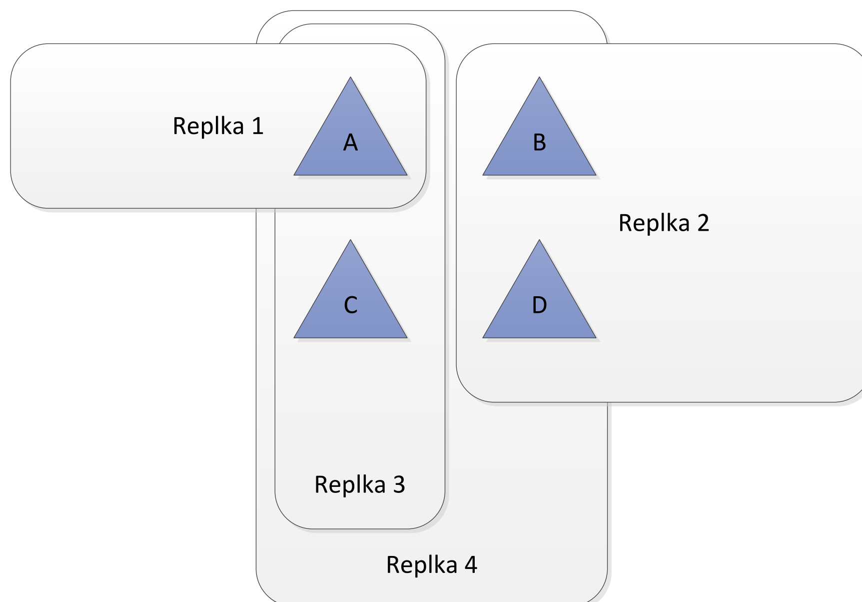

Abbildung 5: Datenmengenabdeckung bei einer selektiven Synchronisation.. A, B, C und D stellen die Datenteile dar. Verschiedene Replikas bilden unterschiedliche Datenmengen ab. Die Daten sind dennoch, in diesem Fall, an mindestens zwei Stellen repliziert.

2.2.3 Roam

Roam wurde extra für mobile Geräte entwickelt. Das System wendet das Ward-Vorgehen an, das dem Ultrapeer-System beim Gnutella-Netzwerk[2] ähnelt. So haben geografisch verteilte Replikas innerhalb einer geografischen Zone (auch Wards genannt, das für „wide area replication domains“ steht) einen Vertreter (genannt Ward

[2]Gnutella ist ein Peer-to-Peer-basiertes Overlaynetzwerk, bei dem es möglich ist, Dateien auszutauschen. Weitere Informationen über das Flooding mit Hilfe der Ultrapeers können in [LHSH05] gefunden werden.

Master), der mit anderen Vertretern des Gesamtsystems kommuniziert. Innerhalb einer geografischen Zone synchronisieren sich die Replikas selbstständig, indem sie einen Kommunikationsring bilden und die Änderungen über die entstandenen Kanäle austauschen. [Rat98] [RRPK01]

Roam zeichnet sich durch bessere Skalierbarkeit gegenüber dem Rumor-System aus. Das wird am Anstieg des Festplattenspeicherverbrauchs pro hinzugefügte Replika sichtbar [Rat98]. Des Weiteren berufen sich Ratner et al. darauf, dass Konflikte laut [RHR+94] selten sind. Und wenn sie stattfinden, dann gibt es eine Art Brennpunkt, sodass der Konflikt nur wenige Replikas tangiert.

2.2.4 Coda

Coda ist ein Client-Server-System für das verteilte Speichern von Volumes. Volumes stellen ein Teil des Dateisystems dar, z. B. kann ein Volume aus einem Ordner mit Unterordnern und Dateien bestehen. Ein Coda-System besteht aus mehreren verteilten Servern, die Daten replizieren, sowie aus Clients, die auf diese Server zugreifen (Abbildung 6). Eine Peer-to-Peer-Verbindung zwischen den Clients ist nicht vorgesehen, nur zwischen den Servern [Rat98]. Dabei ist das Protokoll für repliziertes Schreiben ROWA (Read-One, Write-All). Das Problem der Netzwerkpartitionierung wird dadurch gelindert, dass ein Client auf einen der noch verfügbaren Server in seiner Nähe zugreifen kann. Die Konflikte werden nach dem Beheben der Netzwerkpartitionierung mittels Versions-Vektoren erkannt. [TS08]

In Coda besteht der Konfliktresolutionsalgorithmus aus vier Schritten:

- Lock des betroffenen Datensatzes
- Sammeln und Zusammenführen der Änderungen
- Verteilen und Anwenden der Änderungen
- Aufheben des Locks

Zuerst wird der Datensatz vor Änderungen geschützt. Als Nächstes werden alle Änderungen des Datensatzes auf einem, für die Aktion ausgewählten, Server gesammelt und zusammengeführt. Das Ergebnis der Zusammenführung wird an alle Server verteilt. Diese wenden mittels entsprechenden Algorithmen die Änderungen an. Anschließend wird das Lock auf allen Servern aufgehoben und Clients dürfen

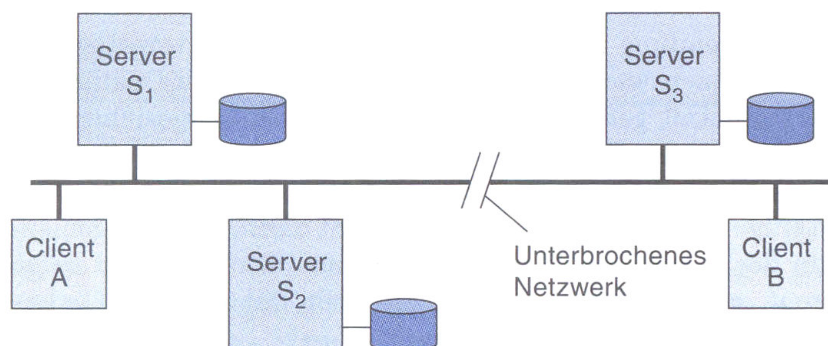

Abbildung 6: Darstellung einer möglichen Aufteilung der Server und Clients für das Dateisystem Coda. Clients verfügen über einen lokalen Cache mit der LRU[3]-Ersetzungsstrategie. Die als „sticky“ markierten Dateien bleiben immer im Cache und werden nicht ersetzt. [SKK+90] [TS08]

Daten wieder ändern, außer es ist ein Konflikt aufgetreten, denn dann wird allen Clients ein Konflikt gemeldet. Konflikte werden manuell korrigiert. Wird eine manuelle Korrektur des Konflikts während der Netzwerkpartitionierung ausgeführt, dann wird nach der Wiederverbindung erneut ein Konflikt an demselben Datensatz gemeldet, der wieder aufgelöst werden muss.[KS93]

In [KSS94] wurde ein erfolgreicher Versuch unternommen, Konflikte in Coda automatisch aufzulösen. Es wurde ein „application-specific resolver“ implementiert, der die Aufgabe auf der Anwendungsebene bewältigt.

2.2.5 Bayou

Bayou ist eine verteilte Datenbank, die auf einer Peer-to-Peer-Infrastruktur aufsetzt und über eine anwendungsspezifische Konfliktauflösung verfügt. Mit diesem Algorithmus kann man relationale Daten, sowie Flatfiles replizieren. [Rat98]

Das System ist dynamisch und Teilnehmer können hinzugefügt oder entfernt werden. Die Trennung zwischen Client und Server ist unscharf, jeder Teilnehmer des Systems erlaubt es einem anderen, auf Daten lesend und schreibend zuzugreifen. Alle Datenänderungen werden mit einem Flooding-ähnlichen Mechanismus an alle Teilnehmer verteilt. Bei jedem Schreibvorgang wird die Operation auf anwendungsspezifische Konflikte untersucht. Konflikte werden dabei mit Hilfe der Versionsvektoren und

Zeitstempel aufgespürt. Für die Konfliktauflösung, wird eine Funktion „mergeproc“ aufgerufen, die das Problem auf der Anwendungslogik-Ebene beseitigt. [DPS+94]

Bei einer Änderung wird der entsprechende Datensatz als „vorläufig“ (engl. tentative) gekennzeichnet, bis sie auf einem der Primary Server übernommen wurde. Primary Server sind gewöhnliche Teilnehmer, die gesondert gekennzeichnet sind. Sobald die Änderung auf einem Primary Server übernommen wurde, gilt der Datensatz als „ausgeliefert“ (engl. committed). Die Anwendungen, die Bayou verwenden, kennen diese zwei Datenzustände und können zwischen den zwei Konsistenzgraden der Daten selbst wählen. [DPS+94]

Arbeiten die Anwendungen mit committed-Daten, dann wird die Konsistenz gewahrt, jedoch muss einige Zeit vergehen, bis tentative-Daten repliziert werden, um abschließend als committed gekennzeichnet zu werden. Verwenden Bayou-Anwendungen die tentative-Daten, dann müssen sie berücksichtigen, dass an diesen Daten ein Konflikt gemeldet werden kann und sie somit für ungültig erklärt werden.

Aufgrund der Datenreplikation kann es vorkommen, dass eine Anwendung Daten auf einen Server geschrieben hat und später veraltete Daten vom anderen Server liest, die aus der Sicht des Servers als committed gelten. „Session Guarantees“ stellen in Bayou folgende vier Eigenschaften des Systems sicher [TDP+94]:

- Read your writes - Beim Lesen werden nicht ältere Daten zurückgegeben, als die, die vom Client gespeichert wurden.
- Monotonic reads - Es werden keinesfalls ältere Daten zurückgegeben, als die, die bereits gelesen wurden.
- Writes follow reads - Geschriebene Daten stehen immer in einer Abhängigkeit zu den zuvor gelesenen Daten.
- Writes follow writes - Daten werden aktualisiert und keinesfalls durch ältere wieder überschrieben.

2.2.6 SyncML

Aufgrund des stetigen Wachstums an mobilen Geräten, steigt auch der Bedarf, Daten zwischen den Geräten auszutauschen. Dabei verwenden Benutzer Geräte verschiedener Hersteller und wollen auf die Synchronität ihrer Daten nicht verzichten.

Deshalb entstand SyncML - ein offener Standard für die Datensynchronisation. Die Daten können Einträge in Kalendern, E-Mails, aber auch Kundendaten der Kundenmanagementtools sein. SyncML unterstützt uni- und bidirektionale Kommunikation, unabhängig vom Übertragungsprotokoll. [JN01]

Auch bei der Verwendung des SyncMLs können Konflikte auftreten. Die Konfliktauflösung beschränkt sich auf drei Methoden [HMPT03]:

- Daten des Clients werden immer übernommen - „Client gewinnt“
- Daten des Clients werden immer verworfen - „Server gewinnt“
- Zuletzt geänderte Daten werden übernommen - „Letzte Änderung gewinnt“

Weitere Konfliktauflösungsstrategien sind nicht vorgesehen, denn das ist nicht der Schwerpunkt des Standards:

> „In our approach, the job of change capture is weighted over others, because if a change is missed, it can not be remedied until the next change occurs at the same object.“ [YYLW08]

2.2.7 BASE

Es gibt ein CAP-Theorem, das besagt, dass die Konsistenz (Consistency), die Verfügbarkeit (Availability) und die Partitionierungstoleranz (Partition tolerant) sich gegenüberstehen. Es wurde in [GL02] bewiesen, dass Webdienste höchstens zwei Aspekte in einem verteilten System sicherstellen können. [Pri08]

- Konsistenz und Verfügbarkeit:
  Wenn ein System eine harte Konsistenz und hohe Verfügbarkeit bieten soll, ist es nicht tolerant gegenüber der Netzwerkpartitionierungen, denn dann blockiert das System, bis alle Datenkopien wieder erreichbar sind, um die Konsistenz nicht zu verletzen.
- Konsistenz und Partitionierungstoleranz:
  Wenn eine starke Konsistenz und die Partitionierungstoleranz erreicht werden soll, dann kann die hohe Verfügbarkeit durch Replizierung nicht erreicht werden.
- Verfügbarkeit und Partitionierungstoleranz:

> Der andere Fall wäre die Sicherstellung der Verfügbarkeit und der Partitionierungstoleranz. Wird das Netzwerk partitioniert, dann wird die Verfügbarkeit durch einen lokalen Cache gesichert. Ohne eine Netzwerkverbindung kann sich der lokale Cache mit dem Datenserver nicht synchronisieren, was zu der Datendivergenz führt und somit die Konsistenz verletzt wird.

Mit dem Zweiphasencommit-Protokoll [TS08] können die ersten zwei ACID-Eigenschaften sichergestellt werden - die Transaktionen werden atomar durchgeführt und überführen das System in jedem Fall in einen konsistenten Zustand. Jedoch wird dabei entsprechend dem CAP-Theorem nur die Konsistenz sichergestellt. Der Zweiphasencommit sieht keine Verfügbarkeits- und Netzwerkpartitionierungstoleranz-Mechanismen vor.

BASE steht für „basically available, soft state, eventually consistent“ [Pri08]. Bei diesem Ansatz wird versucht, ein Kompromiss zwischen den drei CAP-Punkten zu finden. BASE stellt mit dem optimistischen Ansatz das Gegenteil zu der pessimistischen ACID-Alternative dar. BASE speichert nicht nur Daten-Snapshots, sondern auch den Aktionslog. Eine Transaktion besteht nicht aus einem Update von Daten, wie es bei ACID der Fall ist, sondern aus dem Update der Transaktionslog-Tabelle und dem Einreihen der Datenupdate-Kommandos in sichere Message Queues. Damit ist die Transaktion abgeschlossen. Die Message Queues werden Schritt-für-Schritt abgearbeitet. Darüber hinaus sind Ereignisbenachrichtigungen der Applikation beim Abarbeiten der Queues möglich. [Pri08]

Da man davon ausgeht, dass sichere Message Queues in einer endlichen Zeit abgearbeitet werden, ist somit Eventual Consistency gegeben. Eventual Consistency bedeutet, dass das System einen konsistenten Zustand der Daten anstrebt und ihn bei Ausbleiben von weiteren Änderungen, nach einiger Zeit erreicht [TS08].

2.2.8 IceCube

IceCube ist ein Ansatz, Daten auf der Middleware-Ebene zu synchronisieren. In [PB99] wurde der Vorteil der aktionsbasierten Synchronisation über der zustandsbasierten Synchronisation hervorgehoben. So spielen auch in IceCube Aktionen die zentrale Rolle. Dabei werden alle auszuführenden Aktionen an Daten in einem Log festgehalten (ähnlich wie im BASE-Ansatz). Die Aktionen in IceCube haben vier

Eigenschaften [KRSD01]:

- Zielobjekt - Das Objekt für die auszuführende Aktion, z. B. eine Datei
- Vorbedingung - Eine boolesche Funktion, die berechnet, ob sich das Zielobjekt in einem für die Operation passenden Zustand befindet, z. B. die Datei ist nicht größer als 100KB
- Operation - Funktion, die Auswirkungen auf das Objekt und seine Umgebung bewirkt, z. B. Daten in die Datei hinzufügen
- Anhang - Alle zur Aktion zugehörige Daten, z. B. Parameter für die Vorbedingungen und die Operation

Beim Synchronisieren werden die Logs einzelner Replikas in einem konfliktfreien Log zusammengeführt und schließlich werden Daten entsprechend dem Log geändert. Tritt beim Zusammenführen der Logs ein Konflikt auf, dann wird die Reihenfolge der Operationen geändert. Dabei wird auf die Vorbedingungen der Operationen geachtet. Aber auch hier kann ein Konflikt auftreten, der manuell aufgelöst werden muss. [KRSD01]

Folgendes Beispiel veranschaulicht eine automatische Konfliktauflösung. Zwei Administratoren verwalten ein System und führen Operationen aus. In folgender Tabelle sind die Logs der Operationen aufgeführt, die zusammengeführt werden müssen [KRSD01]:

Administrator A	**Administrator B**
A1 Betriebssystem und Treiber von v4 auf v5 aktualisieren	B1 Drucker kaufen, 400 €
A2 Bandlaufwerk kaufen, 800 €	B2 Druckertreiber v4 installieren
A3 Budget um 1500 € erhöhen	

Werden zuerst Kommandos des Administrators A ausgeführt und dann die des Administrators B, dann tritt ein Konflikt auf, weil das System mittlerweile die Version 5 aufweist und die zu installierenden Druckertreiber Version 4. Die Umgekehrte Reihenfolge: zuerst Kommandos des Administrators B, dann die des Administrators A ergeben einen anderen Konflikt. In diesem Fall halten sich die Kosten nicht innerhalb des Budgets. IceCube erkennt die Abhängigkeit zwischen den Logs mit Hilfe der Vorbedingungen. So muss B2 vor A1 ausgeführt werden, sowie A3 vor B1 und

A2. Als eine der möglichen Lösungen des Konflikts wäre die Abfolge A3, B1, B2 A1, A2 richtig. [KRSD01]

2.2.9 Zusammenfassung

Rsync ist ein Client-Server-System mit einer 1:1-Verbindung. Dateien werden in Blöcke aufgeteilt. Die Unterschiede werden mit Hilfe des Vergleichs der Checksummen der Blöcke lokalisiert. Und beim Synchronisieren werden nur die unterschiedliche Blöcke übertragen.

Rumor ist ein Peer-to-Peer-Synchronisierungssystem, das Netzwerkpartitionierung toleriert und Daten mit Hilfe bereitgestellter Algorithmen, die für einzelne Datentypen existieren, zusammenführt. Fehlt ein Merge-Algorithmus für die betroffene Datei, dann muss der Benutzer die Auflösung manuell vornehmen. Rumor unterstützt die selektive Synchronisation.

Roam ist ein System für mobile Geräte. Die geografisch verteilten Replikas werden mit Hilfe des hierarchischen Flooding-Mechanismus synchronisiert. Der Vorteil von Roam gegenüber Rumor ist seine bessere Skalierbarkeit.

Coda ist ein verteiltes Dateisystem, das aus Clients und Servern besteht. Server kommunizieren über Peer-to-Peer, Clients nicht - sie greifen auf Server zu. Mit Versionsvektoren werden Datenänderungen registriert. Konflikte werden zentral aufgelöst. Kann ein Konflikt nicht automatisch aufgelöst werden, dann muss es mit Hilfe eines Resolvers oder manuell geschehen.

Bayou ist eine verteilte Datenbank für relationale Daten und Flatfiles. Die Kommunikationsstruktur ist Peer-to-Peer. Die Konflikte werden mit Versionsvektoren und Zeitstempeln lokalisiert und mittels Anwendungslogik-Resolver aufgelöst. Daten haben zwei Zustände, um eine Wahl zwischen den Konsistenzgraden zu bieten. „Session Guarantees“ sorgen dafür, dass Daten sich für die Anwendung, trotz ihrer Verteilung über die Replikas logisch verhalten und es keine paradoxen Datenänderungen auftreten.

SyncML ist ein offener Standard, um Daten wie z. B. Kalendereinträge, E-Mails und beliebige andere zu synchronisieren. SyncML verfügt nur über wenige Konfliktauflösungsstrategien, weil es während der Entwicklung nicht der Schwerpunkt der

Arbeit war.

BASE ist ein Paradigma, das im Gegensatz zu ACID steht. Bei diesem Ansatz werden hohe Verfügbarkeit und lockere Konsistenz angestrebt. Dies wird mit Hilfe der Aktionslogs und sicheren Message Queues erreicht.

IceCube ist eine Entwicklung, bei der ebenfalls wie bei BASE Aktionslogs festgehalten werden. Der Schwerpunkt der Arbeit lag darin, ein intelligentes System zu entwickeln, das Konflikte auf Anwendungsebene auflöst. Es funktioniert, indem es die Aktionen innerhalb der Aktionslogs neu anordnet und sie in einem Log zusammenführt, ohne dabei die semantischen Zusammenhänge der Daten zu verletzen.

2.2.10 Bewertung

Diese Mechanismen bieten eine gute Grundlage, die verteilten Daten synchron zu halten, jedoch weisen sie Gemeinsamkeiten auf, die die Synchronisationsqualität einschränken. So unternehmen die Algorithmen einen einzigen Konfliktauflösungs-Versuch. Schlägt er fehl, dann ist eine manuelle Resolution unumgänglich. Dies resultiert in einer Frustration von Benutzern der Endsysteme. Manuelle Auflösung ist in großen verteilten Systemen, bei denen komplexe Abläufe ausgeführt werden (z. B. Bank- und Buchungssystemen), nicht zumutbar. Aus diesem Grund werden die Algorithmen eingesetzt, die keine Betriebsstörungen durch die Notwendigkeit eines manuellen Eingriffs verursachen. Stattdessen werden in solchen Systemen Mechanismen eingesetzt, die volle Verfügbarkeit der Systemteilnehmer voraussetzen. Entsprechend können hohe Latenzen und somit ein Eindruck der Trägheit des Systems entstehen, wenn die nötige Verfügbarkeit nicht gegeben ist.

Die bisherigen Lösungsansätze setzen eine gewisse Toleranz von Inkonsistenzen voraus, die später aufgelöst werden oder die Akzeptanz, dass Teile des Systems ihren Betrieb temporär einstellen, bis die Netzwerkverbindung wieder hergestellt wurde.

> „In vielen Situationen besteht die einzige wirkliche Lösung darin, die Konsistenzbeschränkungen zu lockern.“ [TS08]

Somit lässt sich ein Fazit ziehen, dass die bisherigen Synchronisierungsalgorithmen auch in ihrer praktischen Ausprägung nicht zufriedenstellend sind und Verbesserungs- oder Innovationsbedarf besteht. Im nächsten Kapitel verschaffen wir uns einen Über-

blick über die Randbedingungen und Aufgaben des zu entwickelnden Systems.

3 Systemmodell und Problembeschreibung

Die verwandten Arbeiten im vorangehenden Kapitel zeigen den aktuellen Forschungsstand. Bevor ein neues System entwickelt wird, muss die Umgebung, Infrastruktur, Systemteile, sowie die konkreten Aufgaben geklärt werden. In diesem Kapitel werden zunächst das Systemmodell, ein praxisnahes Anwendungsszenario und anschließend die daraus resultierenden Anforderungen an das Zielsystem vorgestellt. Zum Schluss folgt eine Zusammenfassung, die die wesentlichen Kernpunkte dieses Kapitels beschreibt.

Es gibt zwei technische Umgebungen: die Cloudumgebung und die Umgebung der on-premise Knoten (Abbildung 7). Die Cloudumgebung enthält den zentralen Datenspeicher und den dazugehörigen Cloud-Rechner. Die Umgebung der on-premise Knoten besteht aus einzelnen Rechnern, die geographisch verteilt sind und mit dem Internet verbunden sind. Diese Rechner werden in dieser Arbeit als „Knoten“ oder „Clients“ bezeichnet. Ein Knoten besteht aus dem lokalen Cache mit dem lokalen Speicher und der Endanwendung, die einen Cache nutzt.

Die wichtigsten Systemkomponenten werden im nächsten Abschnitt erläutert.

3.1 Infrastruktur und Komponenten

Im Systemmodell werden entsprechend den zuvor besprochenen Umgebungen folgende Komponenten und Infrastruktur-Teile benötigt: Knoten, Cloud und das Netzwerk. Die Eigenschaften der Komponenten werden in diesem Abschnitt einzeln hervorgehoben, dabei wird ihre Rolle im System erklärt. Des Weiteren werden Annahmen über die Komponenten getroffen, die die Randbedingungen für diese Arbeit bilden. Wir beginnen mit den Knoten des verteilten Systems.

3.1.1 Knoten

Wie bereits erwähnt, befinden sich auf einem Knoten die Anwendung, der Cache und der lokale Speicher. Folgende grobe Abläufe finden auf diesem Knoten statt:

- Die Anwendung greift auf den Cache zu und nutzt die dort zwischengespeicherten Daten.

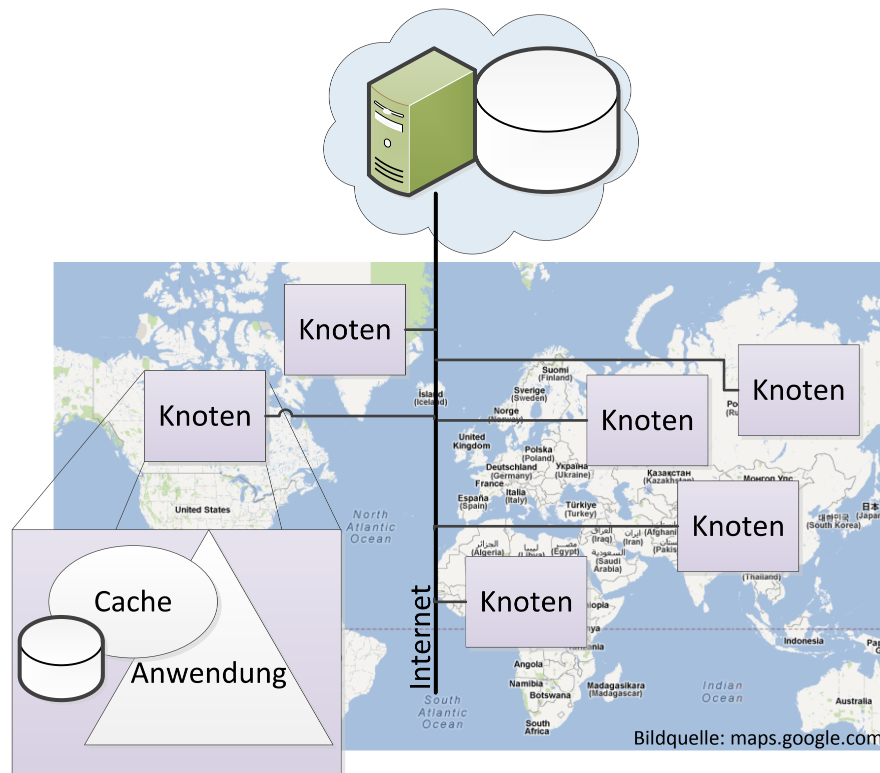

Abbildung 7: Umgebungen und Komponenten des Systems. Die Cloud und die geographisch verteilten Knoten können Informationen über das Internet austauschen.

- Der Cache stellt die Verfügbarkeit der Daten sicher und überwacht die Aktualität der Daten.
- Der lokale Speicher ermöglicht dem Cache eine Persistierung der Daten.

Die Verfügbarkeit eines Knotens ist je nach Systemkonfiguration unterschiedlich. Aus diesem Grund wird der Verfügbarkeitswert als variabel gesehen. Neue Knoten können dem System jederzeit beitreten und später für unbestimmte Zeit aus dem System austreten.

Die meisten PCs haben eine einzelne Festplatte als lokalen Speicher. Deshalb wird es angenommen, dass der lokale Speicher des Knotens unzuverlässig ist und einzelne Dateien nach einem Neustart des Systems verloren gehen können. Des Weiteren kann es eine Störung oder einen Eingriff in die gespeicherten Daten geben. Die

Sicherheit dieser Daten kann durch die Verwendung kryptographischer Verfahren, z. B. Verschlüsselungsmechanismen gewährleistet werden.

3.1.2 Cloud

Cloud Computing stellt Datenspeicher und Rechenressourcen zur Verfügung. Die Rechenressourcen werden für die Unterstützung der Synchronisierungsprotokolle benötigt. Primär ist für die Lösung der Datenspeicher relevant. Der Datenspeicher in der Cloud ist durch interne Replikation ausfallsicher gemacht. So ist unsere Annahme für diese Arbeit, dass Daten in der Cloud nicht verloren gehen, oder durch eine Störung geändert werden. Jedoch beträgt die Verfügbarkeit der Cloud weniger als 100 Prozent. So sind die Rechenkapazitäten der Cloud von Amazon, sowie von Windows Azure zu 99,95% verfügbar [Ama] [Mic]. 99,95% Verfügbarkeit bedeutet, dass die Cloud in einem Monat, laut folgender Rechnung, für rund 22 Minuten ausfällt.

$$30Tage \cdot 24Stunden \cdot 60Minuten \cdot (1 - \frac{99{,}95\%}{100\%}) = 21,6Minuten$$

3.1.3 Netzwerk

Das Netzwerk verbindet die Cloud und Knoten miteinander, sodass ein Datenaustausch ermöglicht wird. ADSL stellt die aktuelle Technologie und eine günstige Methode dar, eine Verbindung zwischen geographisch verteilten Rechnern herzustellen, weshalb angenommen wird, dass die Kommunikation über ADSL stattfinden kann.

Für die Kommunikation wird heutzutage meistens das Kommunikationsprotokoll TCP [Pos81] verwendet. Es werden deshalb die folgenden grundlegenden Eigenschaften von TCP für die Modellierung der Netzwerkschicht übernommen. So wird davon ausgegangen, dass alle Netzwerkverbindungen bidirektional sind und Daten zuverlässig übertragen werden, solange die Verbindung zwischen dem Sender und Empfänger besteht. Daten, die über eine solche Leitung versendet werden gehen nicht verloren und kommen in der gleichen Reihenfolge an, in der sie gesendet wurden.

Durch den Einsatz der Verschlüsselungs- und Authentifizierungsalgorithmen, wie z. B. TLS [Die08] und Kerberos [SNS88], wird davon ausgegangen, dass ausgetauschte

Daten von Dritten weder abgehört noch unbemerkt manipuliert werden können.

Nun wurden die wichtigsten Teile des Systemmodells besprochen. Jetzt müssen die konkreten Anforderungen an das zu entwickelnde System betrachtet werden, womit sich das folgende Unterkapitel beschäftigt.

3.2 Anforderungen an das Zielsystem

Um die Anforderungen an die zu entwickelnde Lösung herauszukristallisieren und anschaulich zu machen, wird in diesem Abschnitt ein Anwendungsszenario untersucht. Danach lassen sich wichtige Aspekte identifizieren und aus technischer Sicht beschreiben. Es werden funktionale und nichtfunktionale Anforderungen unterschieden. Anschließend werden die Anforderungen tabellarisch zusammengefasst.

3.2.1 Beispielanwendung

Eine beispielhafte Anwendung, welche mit den Herausforderungen, die im Rahmen dieser Arbeit untersucht werden, umgehen muss, wird im Folgenden als eine Smartphone-Anwendung für Supermarkt-Kunden vorgestellt. Hierdurch wird die Praxisrelevanz der untersuchten Datensynchronisations-Problemstellung gezeigt.

Aufgrund des stetig wachsenden Ernährungsbewusstseins möchten Menschen für das Kochen einer gewünschten Speise eine Hilfestellung beim Einkaufen im Supermarkt erhalten. Mögliche Zielgruppen wären Vegetarier, Veganer, Allergiker, übergewichtige Menschen, Sportler und Bodybuilder. Sie sollen beim Einkaufen mit Hilfe einer Smartphone-Anwendung unterstützt werden.

Zuerst müssen die Vorlieben an die Anwendung übergeben werden. So können die Ernährungswünsche an die Produkte wie folgt sein:

- Maximaler Eiweißgehalt
- Die Glutamatmenge ist gleich null
- Der Kalorienanteil der Speise beträgt zwischen 2800 und 3200 Kcal
- Es sollen Fairtradeprodukte bevorzugt werden
- Es sollen möglichst günstige Produkte vorgeschlagen werden

- Es dürfen keine tierischen Bestandteile enthalten sein

In der Cloud befindet sich eine Datenbank mit Rezepten. Der Käufer entscheidet sich vor dem Betritt des Supermarkts für „Pasta Bolognese“ und gibt es in die Anwendung ein. Auf dem Smartphone werden Informationen aus der Cloud lokal abgespeichert:

- Das Rezept mit allen Zutaten (Pasta, Hackfleisch, Tomaten)
- Konkrete Produkte (Itali Pasta Fussili, Schnitzelberger Rindhack, Espan Pomodoros)
- Position der Produkte im Supermarkt
- Bestandteile (Mehl, Eier, Wasser, Rindfleisch, Strauchtomaten)
- Chemische Zusammensetzung (Kalorien, Kohlenhydrate, Eiweißgehalt)

Im Supermarkt muss das Smartphone keinen Empfang haben, da er die Infos bereits abgerufen hat.

Anschließend werden verfügbare Produkte mit ihrer Position im Supermarkt-Regal, Preis und Bild angezeigt. Nun hat der Kunde die Wahl, nach einem alternativen Produkt zu suchen, wie z. B. nach einer anderen Marke, oder Fleischersatz statt Hackfleisch.

Der Kunde entdeckt im Regal ein weiteres Produkt, für das er sich entscheidet. Da er will, dass beim nächsten Mal dieses Produkt in Betracht gezogen wird, gibt er die Daten in die Anwendung ein. Wenn er wieder in den Onlinemodus wechselt, wird das neue Produkt mit der Cloud synchronisiert.

Auch Supermärkte können ihre Produkte in der Cloud updaten. Sollte die Cloud ausfallen, können Supermärkte einer oder mehrerer Supermarktketten untereinander, mit Hilfe von Peer-to-Peer-Kommunikation synchronisieren. Ein Kunde kann sich bei einem Cloudausfall über WLAN mit dem Server des Supermarkts verbinden und sein Handy synchronisieren, um aktuelle Informationen über Produkte und beliebte Rezepte zu erhalten.

Sobald die Cloud wieder verfügbar wird, spielen Supermärkte und Smartphones ihre Updates in die Cloud ein. Das Update in der Cloud wird erst ab einer Schwelle an gleichen Änderungsvorschlägen der Benutzer übernommen, um unberechtigtes

Schreiben zu verhindern. Solange es nicht geschehen ist, wird der Eintrag des Nutzers lokal auf seinem Smartphone trotzdem berücksichtigt, da er lokal abgespeichert ist.

Dieses konkrete Beispiel lässt sich nun als eine Menge der Anforderungen abstrahieren, um die Kernpunkte des Systems universal zu formulieren. Wir beginnen mit den funktionalen Anforderungen.

3.2.2 Funktionale Anforderungen

Das Zielsystem soll die Funktionalität eines **verteilten Datenspeichers** realisieren. Für die Speicherung der Daten wird die **Cloud als zentrale Stelle** verwendet. Somit lassen sich Rezepte bei bestehender Internetverbindung aus einer bekannten Quelle herunterladen.

Um die Offlinefähigkeit zu unterstützen müssen Rezepte **lokal gespeichert** werden. Es wird ein Cache benötigt, um Daten unabhängig vom Zustand anderer Teile des Systems (z. B. Netzwerk, Cloud) verfügbar zu halten.

Ein Rezept besteht aus einer Zutaten-Tabelle und den dazugehörigen Bearbeitungsanweisungen. Die Struktur der Daten ist also flach, d. h. es werden **Flatfiles**, z. B. CSV- und Text-Dateien gespeichert. Es sollen keine relationale Daten unterstützt werden.

Für die Datenverarbeitung sollen grundlegende Dateioperationen ermöglicht werden. Diese Operationen werden **CRUD-Operationen** (Create, Read, Update und Delete) genannt. Damit lassen sich Daten in der Cloud und lokal verwalten. Des Weiteren muss es ermöglicht werden, **weitere Operationen an das System anzuschließen** und sie auf Daten auszuführen. Somit wird ermöglicht, dass Rezepte beliebig bearbeitet werden können, aber auch, dass sie anderen Bearbeitungsvorgängen zur Verfügung stehen.

Es wird gefordert, dass auftretende **Konflikte automatisch aufgelöst** werden. Kann ein Konflikt nicht automatisch aufgelöst werden, so muss die Möglichkeit bereitstehen, ihn **manuell aufzulösen**. Damit wird eine Möglichkeit geschaffen, Konflikte in Rezepten automatisch zu beseitigen und bei Bedarf manuell.

Beim Start des Cache-Programms muss die **Integrität von lokalen Daten** geprüft werden. So wird sichergestellt, dass der Anwendung, die Rezepte anzeigt, keine feh-

lerhaften oder unvollständigen Daten gestellt werden.

In der Lösung soll das lockerere Konzept der letztendlichen Konsistenz (**Eventual Consistency**) realisiert werden. Damit wird gesichert, dass beim Einsatz vieler Knoten - im Beispiel: Smartphones - Änderungen an Rezepten auch bei Abwesenheit einiger Knoten durchgeführt werden können und die Konsistenz der Datenbank dabei nicht divergiert.

3.2.3 Nichtfunktionale Anforderungen

Für die Anwendung soll ein **Offlinebetrieb** ermöglicht werden. So kann die Anwendung ohne Netzwerkverbindung nach außen mit Daten versorgt werden. Des Weiteren soll sichergestellt werden, dass Zugriff auf Daten und Änderungen auch während einer Netzwerkpartitionierung (bzw. während eines Ausfalls der Cloud) erfolgen können. Diese Änderungen müssen auch für andere Caches sichtbar werden. Sobald die Netzwerkpartitionierung aufgehoben wird (bzw. die Cloud wieder verfügbar wird), sollen Daten zwischen beiden Partitionen synchronisiert werden. Während einer **bestehenden Verbindung sollen Daten laufend synchronisiert** werden. Auf diese Weise sollen Rezepte aktuell gehalten werden.

Außerdem soll das System kein Single Point of Failure aufweisen. Die **Ausfälle der Cloud und einzelner Knoten sollen durch entsprechende Mechanismen toleriert** werden. Somit besteht trotz Ausfalls des zentralen Speichers eine Möglichkeit, die Rezepte zu synchronisieren.

Daten können von mehreren Knoten gleichzeitig in den zentralen Speicher geschrieben werden, die für andere lesbar sind. Aus diesem Grund sind Maßnahmen zu treffen, die **unberechtigtes Schreiben verhindern**. Jedoch sollen die **lokalen Änderungen der Anwendung unabhängig davon zur Verfügung** stehen, also priorisiert werden. Somit werden Fehler in Rezepten vermieden und lokale Abweichungen vom Zustand im zentralen Speicher behalten.

Der **Speicherverbrauch auf dem Knoten soll möglichst gering gehalten** werden, um breiten Einsatz des Systems zu ermöglichen, z. B in Embedded Systems. Die **Transferkosten zwischen dem Knoten und der Cloud sollen minimiert werden**, um den monetären Aufwand für den Einsatz des Systems möglichst gering zu halten. Entsprechend wird das Betreiben des Rezepte-Systems günstig.

Es sollen offene Schnittstellen und Standards verwendet werden, um **Portabilität** der Lösung sicherzustellen. Auf diese Weise werden nicht nur Rezepte-Systeme, sondern beliebige andere Datenverwaltungs-Systeme unterstützt.

Ein wichtiger Punkt ist die **Verringerung der Konfliktwahrscheinlichkeit und der manuellen Konfliktauflösung**. Die entstandenen Konflikte sollen wenn möglich maschinell aufgelöst werden. Dadurch werden weniger Benutzer damit konfrontiert, einen Konflikt im Rezept manuell aufzulösen.

Nun wurden das Beispielszenario, die funktionalen, sowie nichtfunktionalen Anforderungen erläutert. Im nächsten Abschnitt folgt die Zusammenfassung des ganzen Kapitels. Dabei wird ein Überblick über alle Anforderungen in tabellarischer Form gegeben

3.3 Zusammenfassung

In diesem Kapitel wurde die dem System zugrundeliegende Infrastruktur mit den Anforderungen betrachtet. Die drei Hauptteile des Systemmodells sind Knoten, die Cloud und das Netzwerk. Knoten weisen ähnliche Eigenschaften auf, wie ein handelsüblicher PC. Die Cloud dient als eine hochverfügbare zentrale Speicherstelle im System. Und das Netzwerk ist das Bindeglied zwischen den Systemteilen, das durch eine ADSL-Verbindung realisiert wird.

Das Beispielszenario ist eine zentrale Rezepte-Datenbank in der Cloud und viele Smartphone-Clients, die ihre Lieblingsrezepte lokal speichern, ändern und synchronisieren können. Dabei wird ein Offline-Modus unterstützt, für den Fall, dass die Cloud ausfällt, oder falls der Empfang des Smartphones nicht sichergestellt ist. Aus dem Rezepte-System wurden Anforderungen abgeleitet, die in folgender Tabelle zusammengefasst werden.

Funktionale Anforderungen	**Nichtfunktionale Anforderungen**
1. Bereitstellung eines verteilten Datenspeichers	1. Offlineverfügbarkeit- und Editierbarkeit
2. Cloud als zentraler Datenspeicher	2. Toleranz gegenüber Cloud- und Knotenausfällen
3. Lokale Kopie im Cache	3. Synchronisation mit der Cloud bei bestehender Verbindung
4. Flache Datenstruktur	4. Minimierung der Speicherkosten und Optimierung der Kommunikation zwischen Cloud und Cache
5. Unterstützung der CRUD-Operationen auf Daten	5. Schutz gegen unberechtigtes Schreiben
6. Hinzuschalten weiterer Operationen	6. Lokale Änderungen haben Priorität
7. Automatische und manuelle Konfliktauflösung	7. Portabilität der Lösung
8. Integritätsprüfung der lokalen Daten	8. Reduktion der Wahrscheinlichkeit für manuelle Konfliktauflösung
9. Eventual Consistency	

Nun wurde eine Grundlage für die Entwicklung eines Systems gelegt. Das nächste Kapitel beschäftigt sich mit der Entwicklung des Systems und einzelner Mechanismen, die die abgeleiteten Anforderungen auf Basis der beschriebenen Infrastruktur realisieren.

4 Entwurf

Im vorigen Kapitel wurden das Systemmodell und wichtige Anforderungen für das Zielsystem festgelegt. Nun lässt sich eine grobe Architektur für das Zielsystem entwerfen. Hier wird zuerst eine grobe Struktur erklärt, anschließend wird sie schrittweise verfeinert und die Punkte innerhalb des Systems realisiert, welche für den Entwurf der Lösung relevante Fragestellungen aufwerfen.

4.1 Systemarchitektur

Im Kapitel 3 wurden die Komponenten des Systems dargelegt. Diese Komponenten spiegeln sich in der groben Gesamtarchitektur des Systems wieder. Die Abbildung 8 hebt sie farblich hervor und stellt die drei Hauptbereiche dar: die Cloud, den Cache und die Anwendungsumgebung.

Als nächstes wird die Rolle der einzelnen Hauptbereiche Cloud und Cache erläutert. Auf die Anwendungsumgebung wurde bereits in Kapitel 3 eingegangen.

4.1.1 Cloud

Die Cloud enthält den sicheren Speicher (Cloud Storage), die Mergekomponente (Merge Logic), sowie die Synchronisierungskomponente (Sync Logic).

Cloud Storage ist eine Komponente des Cloud Computing. Dieser Speicher wird von Cloud Computing-Anbietern verwaltet, gesichert und gewartet. Die Anbieter stellen die Verfügbarkeit und die Zuverlässigkeit des Speichers sicher, sodass der Kunde keinen Administrierungsaufwand hat und der Speicher für die zu entwickelnde Lösung als wartungsfrei und ausfallsicher gilt.

Die Mergekomponente sorgt für die Datenintegrität innerhalb des Cloud Storage, indem sie die Änderungen der Caches, die konfliktbehaftet sein können, in einen konfliktfreien Zustand überführt.

Die Synchronisierungskomponente erlaubt den Zugriff auf die Daten und steuert die Abläufe, um Daten herunter- oder hochzuladen.

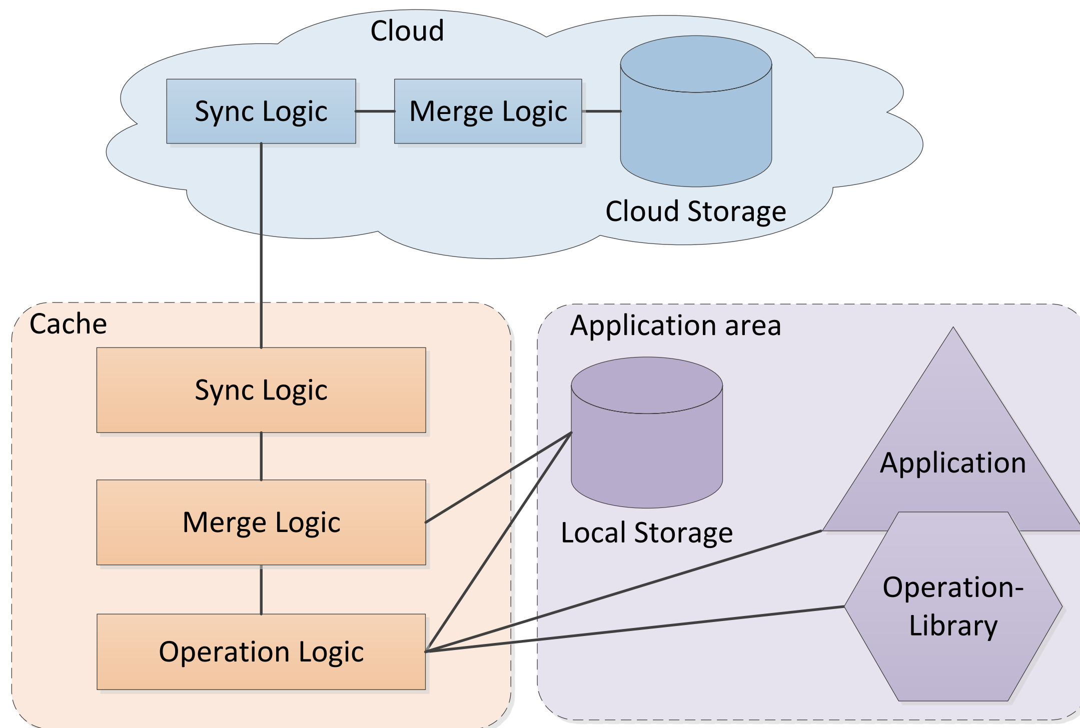

Abbildung 8: Grobarchitektur des Zielsystems. Es gibt drei Hauptbereiche: Cloud, Cache und die Anwendungsumgebung. Der Cache fungiert als ein Bindeglied in zwischen der Cloud und der Anwendung.

4.1.2 Cache

Der Cache besteht aus Sync Logic, Merge Logic und Operation Logic.

Sync Logic steuert den Austausch der verteilten Daten. Hierbei wird sie benötigt, um Daten kosteneffizient zu synchronisieren.

Merge Logic ist für die Konsistenz der Daten verantwortlich, es enthält Konfliktbeseitigungs-Algorithmen, um die lokale Integrität der Daten zu sichern.

Operation Logic nimmt Kommandos von der Anwendung entgegen und führt sie aus.

Nun wurde der Grundaufbau der Systemteile und ihre Aufgaben beschrieben. Nun können Algorithmen diskutiert werden, die in diesen Komponenten realisiert werden.

4.2 Algorithmen und Mechanismen

Im Folgenden werden Lösungen für die in Abschnitt 3.2 gestellten Anforderungen vorgestellt und begründet. Es wird dargelegt, wie die Anforderungen realisiert, welche Algorithmen verwendet und welche Mechanismen und Abläufe von anderen Systemen übernommen werden. Abschließend folgt eine Zusammenfassung der Mechanismen, die einzelnen Anforderungen zugeordnet werden. Wir beginnen mit der Grundidee, die Basis-Algorithmen für die wichtigsten Abläufe festhält.

4.2.1 Grundidee

Im Abschnitt 2.2.7 wurde das CAP-Theorem in Grundzügen erläutert und dabei der Ansatz BASE als ein State of the Art-Kompromiss für das Problem vorgestellt. In dieser Lösung werden aufgrund der Anforderungen, die Verfügbarkeit und Ausfallsicherheit sicherzustellen, die zwei Extrema Availability und Partitiontolerance des Theorems angestrebt. Dies wird im System BASE erreicht und dabei eine Sicherheit der Konvergenz gegen einen konsistenten Zustand der Daten gegeben, sodass man durch Einsatz von BASE den Konsistenz-Punkt des Theorems ebenfalls anstrebt (Abbildung 9). BASE kann die letztendliche Konsistenz nur dann bieten, wenn Daten durch festgelegte, systemweit bekannte Aktionen geändert werden.

Diese Einschränkung ist für das zu entwickelnde System vorteilhaft, denn durch Einsatz von Aktionen, wird die Verwendung von Algorithmen, wie in IceCube (Abschnitt 2.2.8) möglich, die sich durch eine bessere Konfliktauflösung, verglichen mit zustandsorientierten Auflösungsalgorithmen, auszeichnen.

Die Kombination dieser zwei Algorithmen macht das Gesamtsystem verfügbar, tolerant gegen Ausfälle, realisiert Eventual Consistency und reduziert die Anzahl an Konflikten, die manuell aufgelöst werden müssen.

Um die internen Vorgänge von der Endanwendung zu verbergen und die Benutzung zu vereinfachen, wird das System als Middleware realisiert, die über eine API verwendet werden kann.

Dadurch, dass die Cloud keine 100 % Verfügbarkeit hat, wird eine Failover-Lösung benötigt, um die möglichen kurzzeitigen Ausfälle zu überbrücken. Peer-to-Peer gehört aufgrund der Robustheit gegenüber zufälligen Ausfällen[CLMR04] der Knoten

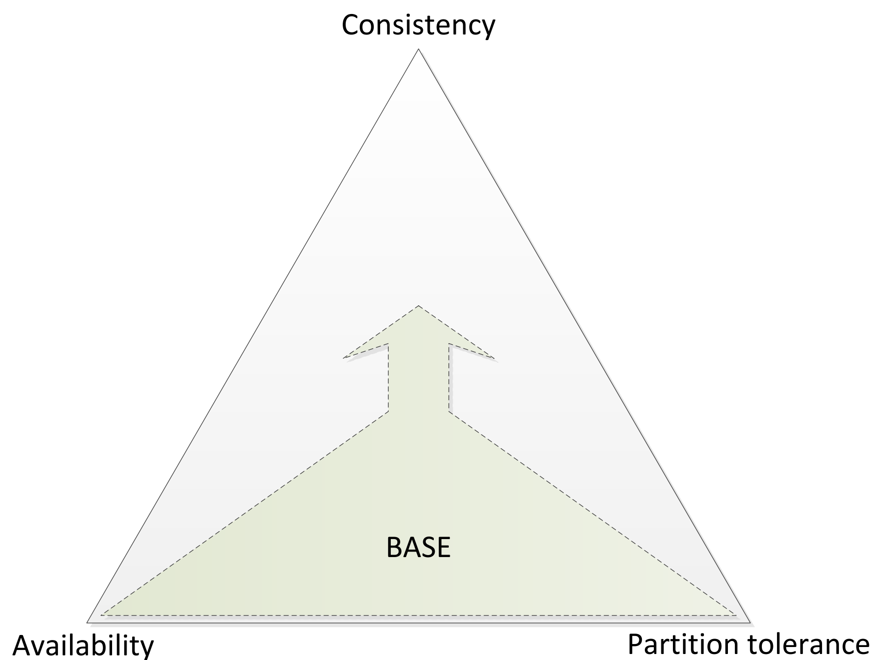

Abbildung 9: BASE-Eigenschaften abgebildet auf das CAP-Theorem. BASE sichert die Verfügbarkeit und Partitionstoleranz und strebt die letztendliche Konsistenz an.

und der hohen Verfügbarkeit[TS08] zu den sichersten Kommunikationsmodellen. Aus diesem Grund wird es alternativ zur Cloud eingesetzt.

Nun werden die funktionalen und nichtfunktionalen Anforderungen nacheinander besprochen.

4.2.2 Funktionale Anforderungen

Bereits im Systemmodell (Kapitel 3) wurde festgelegt, dass Daten sowohl in der Cloud, als auch lokal auf Caches gespeichert werden. Dadurch ergibt sich ein verteiltes Datenspeichersystem, wie es in der Anforderung **Bereitstellung eines verteilten Datenspeichers** gefordert wird.

Wie es für die Verwendung von BASE und IceCube notwendig ist, werden Aktionen

definiert, die Daten verarbeiten. Die Anforderung **Unterstützung der CRUD-Operationen auf Daten** wird erfüllt, indem eine abstrakte Operation implementiert wird, die BASE- und IceCube-kompatibel ist. Alle Operationen, die an Daten ausgeführt werden, müssen von der abstrakten Operation ableiten. Entsprechend werden Create, Read, Update und Delete auf Basis der abstrakten Operation implementiert.

Die Anforderung **Hinzuschalten weiterer Operationen** kann darauf aufbauend realisiert werden. Alle Operationen liegen dem Cache als DLL-Dateien vor, die ausgetauscht, hinzugefügt und entfernt werden können. Beim Starten des Caches, lädt er aus einem bestimmten Ordner alle DLLs, die Operationen enthalten. Sobald die DLLs geladen wurden, stehen die dort programmierten Operationen zur Verfügung und können ausgeführt werden.

Dadurch, dass alle Änderungen primär in die Cloud hochgeladen werden, bildet sie eine zentrale Datenhaltungs-Stelle im System, womit die Anforderung **Cloud als zentraler Datenspeicher** erfüllt wird.

Die Speicherressourcen des Cloud Computings können in zwei Kategorien eingeordnet werden: die relationalen und die nichtrelationalen Speicherressourcen. In der ökonomischen Hinsicht ist der nichtrelationale-Speicher, auch Blob-Speicher genannt, günstiger als der relationale. Aufgrund der Anforderungen **Flache Datenstruktur** und **Minimierung der Speicher- und Transferkosten** ist der Blob-Speicher für das Speichern der Daten ideal. Für ein erleichtertes Datenmanagement werden Datensätze in einzelnen, voneinander unabhängigen Blobs gespeichert. So können sie flexibel erstellt, bearbeitet und entfernt werden, ohne dass andere Datensätze dadurch beeinflusst werden. In Coda (Abschnitt 2.2.4), entsprechend der Arbeit von Gray et al. [GHOS96] wird das sogenannte Two Tier-System eingesetzt, bei dem Server auf der ersten Ebene konsistente Daten speichern, Änderungsanfragen erfüllen und sich untereinander über Peer-to-Peer synchronisieren. Um Änderungen an Daten vorzunehmen, greifen Clients, die zur zweiten Ebene gehören, auf einen der Server zu. In der Cloud existiert die erste Ebene auch, aber sie ist selbst für Entwickler transparent und erscheint als eine einzige Komponente.

Damit sichere Warteschlangen nach einer Datenträgerschädigung keine falschen Informationen in die Cloud übermitteln und Anwendungen nur mit richtigen Daten versorgen, werden kryptographische Hashes für die einzelnen Datensätze kalkuliert

und lokal abgespeichert. Geschieht eine, durch eine Störung verursachte Änderung der lokalen Daten, so unterscheidet sich mindestens ein neuberechneter Hashwert von dem abgespeicherten Kontroll-Hashwert. Entsprechend wird der betroffene Datensatz als „beschädigt“ gemeldet. Die Anforderung **Integritätsprüfung der lokalen Daten** ist somit erfüllt.

Die Anforderung **Automatische und manuelle Konfliktauflösung** wird ähnlich wie im System IceCube durch die Zusammenführung der Aktionslogs realisiert. Dabei werden einzelne Operationen aus zwei konfliktbehafteten Datensätzen in einem konfliktfreien Log zusammengeführt und ausgeführt. Für die manuelle Konfliktauflösung wird eine Schnittstelle bereitgestellt, über die beide konfliktbehafteten Versionen ausgelesen werden können, eine der Versionen als konfliktfrei gewählt und die andere verworfen werden kann. Alternativ kann eine dritte, manuell erstellte Version übermittelt und der Konflikt somit aufgelöst werden.

Eventual Consistency Eventual Consistency ist ein clientzentriertes Konsistenzmodell, das sicherstellt, dass Daten gegen einen konsistenten Zustand konvergieren.

> Datenspeicher mit dieser Art von Konsistenz verfügen daher über die Eigenschaft, dass bei ausbleibenden Aktualisierungen alle Replikate nach und nach konsistent werden. Diese Form der Konsistenz wird als Eventual Consistency bezeichnet. [TS08]

Tanenbaum et al. beschreibt in [TS08], dass Eventual Consistency seine Ursprünge im System Bayou (Abschnitt 2.2.5) hat. Er formuliert die einzelnen Punkte der Session Guarantees aus [TDP+94] wie folgt:

- Konsistenz für monotones Lesen - wenn ein Prozess den Wert eines Datenelementes x liest, gibt jede anschließende Leseoperation dieses Prozesses auf x stets denselben oder einen aktuelleren Wert zurück.
- Konsistenz für monotones Schreiben - Eine Schreiboperation eines Prozesses an einem Datenelement x wird abgeschlossen, bevor eine folgende Schreiboperation auf x durch denselben Prozess erfolgen kann.
- „Read Your Writes“-Konsistenz - Die Folge einer Schreiboperation eines Prozesses auf das Datenelement x wird für eine anschließende

Leseoperation auf x durch denselben Prozess stets sichtbar sein.

- „Writes Follow Reads"-Konsistenz - Einer Schreiboperation eines Prozesses auf ein Datenelement x, die auf eine vorherige Leseoperation auf x durch denselben Prozess folgt, wird garantiert, dass sie auf demselben oder einem aktuelleren Wert von x stattfindet. [TS08]

Durch den Einsatz von einer lokalen Datenkopie, die bei jeder Datenoperation zur Verfügung steht, bevor andere Kopien und der zentrale Datenspeicher kontaktiert werden, wird eine Art Session aufgebaut, sodass alle Punkte der Session Guarantees dadurch erfüllt werden.

Die Anforderung **Eventual Consistency** wird mit dem Einsatz des Systems BASE erreicht: Jede Aktion wird in eine sichere Warteschlange eingereiht, sodass die Änderungen mit dem Abarbeiten der Warteschlangen in die Cloud propagiert werden und somit im gesamten System letztendlich vorgenommen werden. Entsprechend werden Operationen in dieser Lösung an einem lokalen Datensatz sofort ausgeführt und einer sicheren Warteschlange hinzugefügt, sodass die durchgeführten Operationen beim nächsten Synchronisationsvorgang hochgeladen werden. Auf diese Weise ist es gesichert, dass eine Kopie des betroffenen Datensatzes lokal vorhanden ist und somit die Anforderung **Lokale Kopie im Cache** erfüllt ist. Inspiriert von [GHOS96] und dem Dateisystems Coda, werden zwei Datensatzversionen gespeichert: die letzte aktuelle digital signierte Version aus dem zentralen Speicher und die Änderungen dieser Version. Die Cloudversion kann über Peer-to-Peer, von Caches, die sie zwischengespeichert haben, an Knoten verteilt werden, die auf die Cloud nicht zugreifen können. Für das Verteilen der Daten über Peer-to-Peer existieren Flooding-Ansätze, wie z. B. [BGL+06], sodass die letztendliche Konsistenz auch während eines Cloudausfalls erreicht werden kann.

4.2.3 Nichtfunktionale Anforderungen

Die Anforderung **Offlineverfügbarkeit- und Editierbarkeit** wird mit dem Sicherstellen der Verfügbarkeit durch BASE erreicht.

In dieser Lösung wird Peer-to-Peer eingesetzt, um die Anforderung **Toleranz gegenüber Cloud- und Knotenausfällen** zu erfüllen. Fällt die Verbindung zur

Cloud aus, dann werden Aktualisierungen zwischen den Clients über das Peer-to-Peer-Netzwerk ausgetauscht. Eine solche Funktionalität wurde in Coda [Rat98] [SKK+90] und Bayou [DPS+94] verwirklicht, um Aktualisierungen zwischen den Datenservern auszutauschen und die Kopien konsistent zu halten (Abbildung 6).

Jeder Cache kann Daten ändern und in die Cloud hochladen, die für andere Teilnehmer sichtbar werden. Mit der Anforderung **Schutz gegen unberechtigtes Schreiben** soll verhindert werden, dass fehlerhafte Daten in der Cloud gespeichert werden. Dies ist realisierbar, indem man eine Schwelle festlegt, sodass ab einer bestimmten Anzahl der gleichen oder ähnlichen Einträge die Daten in der Cloud übernommen und für alle sichtbar werden. Dieses Vorgehen erschwert die absichtliche oder unabsichtliche Fälschung der Daten, bietet jedoch keine Garantie, denn ein Angreifer kann mehrere Clients gleichzeitig starten, Daten abändern und somit die Schwelle an gleichen Einträgen erreichen, was in einer vertrauenswürdiger Umgebung unrealistisch ist. Der Schwellwert kann je nach Anwendungsszenario variieren. Wird diese Funktionalität des Systems nicht benötigt, dann kann der Schwellwert auf eins reduziert werden, damit Änderungen in der Cloud sofort übernommen werden.

Um die Anforderung **Portabilität der Lösung** zu realisieren, werden für die Kommunikation zwischen der Anwendung und dem Cache, sowie zwischen dem Cache und der Cloud Webservices verwendet. Es werden SOAP-Nachrichten [BTN00] ausgetauscht und Schnittstellen in Form von öffentlich zugänglichen WSDL-Dateien [CCM+01] publiziert. Durch die Verwendung von Webservices, lassen sich auch B2B-Systeme an das zu entwickelnde System anschließen, bei denen der größte Teil der Kommunikation grundsätzlich über Webservices stattfindet. Somit wird eine API angeboten, mit der das System flexibel verwendet werden kann.

Die Anforderung **Reduktion der Wahrscheinlichkeit für manuelle Konfliktauflösung** wird mit der Anwendung der Aktionslog-basierten Zusammenführung der Datensatz-Zustände, mit einem IceCube-ähnlichen-System erreicht. In [GHOS96] wird deutlich gemacht, dass die Verwendung von Aktionen statt Datensatz-Zuständen, die Konvergenz der verteilten Daten gegen einen konsistenten Zustand unterstützt und somit die Wahrscheinlichkeit für manuelles Auflösen reduziert. Es werden Gemeinsamkeiten in den zwei Logs gesucht. Die unterschiedlichen Teile werden zusammengeführt, indem die Aktionen in eine eindeutige Reihenfolge gebracht werden, sodass die Ausführung dieser Aktionen konfliktfrei verläuft. Des Weiteren wird

die Auflösung der Konflikte durch die anwendungsspezifische Funktion auf Anwendungsebene durchgeführt. So können auch die CRUD-Konflikte maschinell und anwendungsgerecht behoben werden, bevor ein manueller Eingriff benötigt wird. Diese Funktionalität ist im Abschnitt 4.3.4 detailliert beschrieben.

Synchronisation mit der Cloud bei bestehender Verbindung

Zunächst muss festgestellt werden, welche Datensätze synchronisiert werden sollen. Hierfür kann entweder der Pull- oder der Push-Ansatz verwendet werden. Beim Push-Ansatz kennt der Server alle Clients und benachrichtigt sie, wenn Aktualisierungen vorliegen. Beim Pull-Ansatz senden Clients an den Server Anfragen, um zu erfahren, ob es Aktualisierungen für bestimmte Datensätze gibt.[TS08]

Es können drei Fälle auftreten, in denen eine Synchronisation notwendig ist: Änderung in der Cloud, aber nicht lokal; Änderung lokal, aber nicht in der Cloud; Änderung Lokal und in der Cloud. Gibt es weder Änderungen in der Cloud noch lokal, dann muss nicht synchronisiert werden.

Änderung in der Cloud, aber nicht lokal

Zu jedem Datensatz wird der Zeitpunkt der letzten Änderung gespeichert. Zunächst werden Zeitpunkte aus der Cloud für die Datensätze, die auch lokal existieren, heruntergeladen und mit den lokalen Werten verglichen. Werden Unterschiede festgestellt, dann müssen entsprechende Datensätze synchronisiert werden. Die Prüfung auf Änderungen in der Cloud muss in regelmäßigen Zeitabständen durchgeführt werden, denn im Pull-Ansatz ist keine Benachrichtigung über Änderungen vorgesehen.

Die Routine für den Pull-Ansatz kann in vier Schritte eingeteilt werden:

1. Tabelle mit Datensatznamen und Zeitstempel der letzten Änderung aufbauen
2. Zeitstempel der Datensätze in der Cloud herunterladen
3. Vergleichen der Zeitstempel
4. Unterschiedliche Datensätze synchronisieren

Beim Push-Ansatz wird wie folgt vorgegangen:

1. Benachrichtigung an alle Clients mit Datensatz-Namen senden
2. Clients prüfen, ob der Datensatz für sie relevant ist (selektive Synchronisation)

3. Ist der Datensatz relevant, dann initiiert der Client den Ladevorgang.

Im Peer-to-Peer-Modus kann beim Push-Ansatz ähnlich vorgegangen werden. Beim Pull-Ansatz muss der Client seine Peers nach einer Datenversion fragen, die jünger ist als seine. Bekommt er positive Antworten, kann der Ladevorgang zwischen zwei Peers beginnen.

1. Anfrage für einen Datensatz mit Name x und Änderungsdatum >y senden
2. Jüngste Version des Datensatzes aus Rückmeldungen auswählen
3. Ladevorgang starten

Um Datensätze zu aktualisieren, kann der Ansatz des Systems „Rsync“ (siehe Abschnitt 2.2.1) angewendet werden. Dabei werden nur die unterschiedlichen Teile eines Datensatzes ausgetauscht. Dies reduziert die Transferkosten.

Änderung lokal, aber nicht in der Cloud

In diesem Fall wird die lokale Änderung in die Cloud hochgeladen oder über das Peer-to-Peer-Netzwerk verteilt. Wann das Hochladen erfolgt, entscheidet die Optimierungsfunktion, die nachfolgend detailliert beschrieben wird.

Änderung lokal und in der Cloud

Dieser Fall kann durch eine Netzwerkpartitionierung auftreten, indem festgestellt wird, dass in der Cloud eine neuere Version vorhanden ist, und dass Änderungen auch lokal am gleichen Datensatz vorgenommen wurden. In dieser Situation wird der Konflikt lokal aufgelöst und die konfliktfreie Version schließlich in die Cloud hochgeladen. Das genaue Vorgehen beim Auflösen des Konflikts ist in Abschnitt 4.3.4 beschrieben.

Minimierung der Speicherkosten und Optimierung der Kommunikation zwischen Cloud und Cache

Es wird ermöglicht, dass nur ein relevanter Teil aller Daten synchron bleibt. Hierfür muss der zu entwickelnden Lösung mitgeteilt werden, welche Datensätze sollen aktuell gehalten werden. Entsprechend werden nur diese Datensätze beim Synchronisieren berücksichtigt und übertragen. Dieses Vorgehen wird „selektive Synchronisation“ genannt und verhindert das überflüssige Speichern und Synchronisieren.

Daten, auf die oft zugegriffen wird müssen zwecks guter Performance unkompri-

miert bleiben. Daten, deren Zugriffe durchschnittlich einen bestimmten Schwellwert unterschreiten, werden komprimiert und bei Bedarf temporär entpackt.

Einige Knoten, die erwartungsgemäß hohe Verfügbarkeit und geringe Kommunikationskosten haben (z. B. stationäre Rechner, Server, keine Smartphones), können Aktualisierungen von der Cloud herunterladen und über Peer-to-Peer an andere Knoten verteilen. Das würde das Traffic in der Cloud, und somit Kosten minimieren. Um die dabei möglichen Daten-Fälschungen auszuschließen, wird jede Aktualisierung bereits in der Cloud digital signiert.

Die Übertragungskosten für Daten wachsen mit der Synchronisierungsfrequenz. Entsprechend den Anforderungen an die Lösung soll der finanzielle Aufwand möglichst gering halten werden.

Aus Zugriffs- und Änderungs-Rate, sowie einer Kostenfunktion kann eine Synchronisierungsrate ermittelt werden, bei der die Kosten minimiert und die Aktualität der Daten maximiert werden. Im Folgenden wird die Optimierungsfunktion mathematisch ermittelt.

Das Problem kann als das mathematische Constrained Optimization Problem formuliert und gelöst werden. Für die Lösung wird eine Bewertungsfunktion verwendet, die das Problem in ein Unconstrained Optimization Problem umwandelt. Als Folge lassen sich die Extrempunkte der mathematischen Funktion bestimmen und somit der gesuchte Wert. Das Problem wird zweierlei gelöst, nämlich für das Hochladen der Aktualisierungen „put“ und für das Herunterladen „get”'

Bewertungsfunktion

Sei $eval(x)$ die Bewertungsfunktion. Sie enthält Extremwerte, die für unsere Lösung von Bedeutung sind. $f(x)$ ist die mathematische Funktion, deren Eingaben bewertet werden sollen. Die Bewertung ändert sich entsprechend der Penalty-Funktion $p(x)$. Yeniay [Yen05] beschreibt zwei Typen von Bewertungsfunktionen für Constrained Optimization Problems:

1. Additive Form: $eval(x) = f(x) + p(x)$
2. Multiplikative Form: $eval(x) = f(x) \cdot p(x)$

In dieser Arbeit müssen die Synchronisationsraten hinsichtlich der Kosten bewertet werden. Dabei beeinflusst $p(x)$ die Bewertung entsprechend der Wahrscheinlichkeit

für das Lesen eines inkonsistenten Datensatzes, abhängig von der Synchronisationsrate x. Die Funktion $f(x)$ errechnet die Synchronisations-Kosten, abhängig von der Synchronisationsrate. Das Minimum der Bewertungsfunktion $eval(x)$ ergibt Kosten für die optimale Synchronisationsrate x_{opt}. Die Synchronisationsrate x bezieht sich auf einen festen Zeitabschnitt t und gibt an, wie oft eine Synchronisierung innerhalb dieses Zeitabschnitts stattfindet. Die Konstante t kann z. B. *1 Tag* oder *1 Monat* betragen.

In Realität entstehen durch das Lesen eines inkonsistenten Datensatzes messbare Kosten in €. Unabhängig von der Konsistenz des Datensatzes entstehen für das Synchronisierungsvorgänge ebenfalls messbare Kosten in €. Die Summe ergibt die Gesamtkosten für den Betrieb des Systems, die entsprechend der Aufgabenbeschreibung minimiert werden sollen. Aus diesem Grund müssen diese Kosten auch in der Bewertungsfunktion addiert werden, d. h. es wird die additive Form der Bewertungsfunktion verwendet:
$eval(x) = p(x) + f(x)$

Penalty-Funktion

Nun wird die Penalty-Funktion hergeleitet. Finanzielle Kosten verursacht das Lesen eines veralteten Datensatzes oder das nichthochladen einer Änderung. Das Nichtsynchronisieren der Datensätze, die nicht gelesen werden verursacht keine Kosten. Auch das Synchronisieren der Datensätze, die nicht gelesen werden, bringt keine Vorteile - weder finanzielle, noch funktionelle. Dementsprechend muss für das Herunterladen nur die Leserate und die Aktualität der Datensätze in Betracht gezogen werden. Entsprechend ist für das Hochladen die Schreibrate wichtig. Betrachten wir zuerst die Penalty-Funktion für das Herunterladen, also für die Get-Aktion.

Die Wahrscheinlichkeit für das Lesen eines veralteten Datensatzes beträgt $\frac{1}{x}$. Das entspricht einer falschen Leseoperation aus x Synchronisierungen pro Zeitabschnitt. Der Wert wird mit der Anzahl der Leseoperationen innerhalb des Zeitabschnitts a_{reads} multipliziert, was die statistische Anzahl an falschen Leseoperationen ergibt. Anschließend wird eine eins subtrahiert, denn wenn die Anzahl der Leseoperationen und Synchronisierungen gleich ist, dann entstehen keine Kosten. Jede falsche Leseoperation verursacht Kosten in Höhe von c_{read_old}, deswegen wird dieser Wert

multipliziert. Abschließend wird ein Faktor a_{caches} hinzugefügt, um die Gesamtheit aller synchronisierenden Caches zu berücksichtigen, damit die Kosten insgesamt gesenkt werden. Zusammenfassend ergibt sich die folgende Formel:
$p_{get}(x) = (\frac{1}{x} \cdot a_{reads} - 1) \cdot c_{read_old} \cdot a_{caches}$

Ähnlich verhält es sich mit den Kosten bei der Put-Aktion, die durch das nichthochladen von Änderungen entstehen. Jedoch muss die Leserate durch die Schreibrate ersetzt werden. Die Kosten für eine nicht hochgeladene Änderung sind viel Höher, als die für das Lesen eines falschen Datensatzes. Dies kommt dadurch zustande, dass jeder einzelne der Knoten, die veraltete Version mehrmals liest. Also muss der entsprechende Kostenfaktor c_{detain_write} eingesetzt werden, der in Realität ungefähr $a_{reads} \cdot c_{read_old}$ beträgt, in der Formel aber allgemein gehalten wird. Auch bei der Put-Aktion muss die Anzahl der Caches a_{caches} berücksichtigt werden. Das resultiert in folgender Formel:
$p_{put}(x) = (\frac{1}{x} \cdot a_{writes} - 1) \cdot c_{detain_write} \cdot a_{caches}$

Kosten-Funktion

Die Kosten entstehen durch die Übertragung der Daten von- und zur Cloud. Um die Gesamtdatenmenge für Synchronisierungen innerhalb des Zeitabschnitts zu berechnen, werden Synchronisierungsrate x mit der durchschnittlichen Datenmenge d_{data}, die übertragen werden muss, multipliziert. Das Ergebnis mit dem Kostenfaktor $c_{dataset}$ multipliziert, ergibt die Kosten für die gesamte Datenübertragung über den Zeitabschnitt. Mit dem Faktor a_{caches} werden Kosten berechnet, die alle Caches verursachen. Des Weiteren gibt es Fix-Kosten c_{fix} pro Zeitabschnitt t.

$f_{put}(x) = x \cdot d_{data_up} \cdot c_{dataset} \cdot a_{caches} + c_{fix}$
$f_{get}(x) = x \cdot d_{data_down} \cdot c_{dataset} \cdot a_{caches} + c_{fix}$

Constrained Optimization Problem

Als Nächstes lässt sich das Optimierungsproblem mit Nebenbedingungen wie folgt aufstellen.

Minimize:

- $eval_{put}(x) = p_{put}(x) + f_{put}(x)$
- $eval_{get}(x) = p_{get}(x) + f_{get}(x)$

subject to:

- $p_{put}(x) = (\frac{1}{x} \cdot a_{writes} - 1) \cdot c_{detain_write} \cdot a_{caches}$
- $p_{get}(x) = (\frac{1}{x} \cdot a_{reads} - 1) \cdot c_{read_old} \cdot a_{caches}$
- $f_{put}(x) = x \cdot d_{data_up} \cdot c_{dataset} \cdot a_{caches} + c_{fix}$
- $f_{get}(x) = x \cdot d_{data_down} \cdot c_{dataset} \cdot a_{caches} + c_{fix}$
- $c_{detain_write} = a_{reads} \cdot c_{read_old}$
- $a_{reads} = r_{read_write} \cdot a_{writes}$
  Ein Datensatz wird in manchen Systemen öfter gelesen als geschrieben. Entsprechend ist die Leserate um einen Faktor größer als die Schreibrate.

Um das Minimierungsproblem zu lösen, werden die Evaluierungsfunktionen abgeleitet und anschließend werden die Nullstellen bestimmt und die letztendliche Formel expandiert. Die Nullstellen geben die Optimale Synchronisierungsrate an.

$$eval_{put}(x) = (\frac{1}{x} \cdot a_{writes} - 1) \cdot c_{detain_write} \cdot a_{caches} + x \cdot d_{data_up} \cdot c_{dataset} \cdot a_{caches} + c_{fix}$$
$$eval'_{put}(x) = d_{data_up} \cdot c_{dataset} \cdot a_{caches} - \frac{a_{writes} \cdot c_{detain_write} \cdot a_{caches}}{x^2}$$
$$x_{opt_put} = \pm\frac{\sqrt{a_{writes} \cdot c_{detain_write}}}{\sqrt{d_{data_up} \cdot c_{dataset}}}$$
$$x_{opt_put} = \pm\frac{\sqrt{a_{writes} \cdot r_{read_write} \cdot a_{writes} \cdot c_{read_old}}}{\sqrt{d_{data_up} \cdot c_{dataset}}}$$

$$eval_{get}(x) = (\frac{1}{x} \cdot a_{reads} - 1) \cdot c_{read_old} \cdot a_{caches} + x \cdot d_{data_down} \cdot c_{dataset} \cdot a_{caches} + c_{fix}$$
$$eval'_{get}(x) = d_{data_down} \cdot c_{dataset} \cdot a_{caches} - \frac{a_{reads} \cdot c_{read_old} \cdot a_{caches}}{x^2}$$
$$x_{opt_get} = \pm\frac{\sqrt{a_{reads} \cdot c_{read_old}}}{\sqrt{d_{data_down} \cdot c_{dataset}}}$$
$$x_{opt_get} = \pm\frac{\sqrt{r_{read_write} \cdot a_{writes} \cdot c_{read_old}}}{\sqrt{d_{data_down} \cdot c_{dataset}}}$$

Entsprechend müssen Änderungen pro Zeitabschnitt x_{opt_get} Mal herunter- und x_{min_put}

Mal hochgeladen werden, um das beste Kosten-/Nutzen-Verhältnis einzuhalten. Dabei muss beachtet werden, dass Aktualisierungen nicht öfter heruntergeladen werden sollen, als sie gelesen werden:

$x_{opt_get} \leq a_{reads}$

Und die Aktualisierungen sollen nicht öfter hochgeladen werden, als sie produziert werden:

$x_{opt_put} \leq a_{writes}$

Entsprechend gilt:

$$x_{opt_get_limited} = \begin{cases} x_{opt_get}, & \text{falls } x_{opt_get} \leq a_{reads} \\ a_{reads}, & \text{sonst} \end{cases}$$

$$x_{opt_put_limited} = \begin{cases} x_{opt_put}, & \text{falls } x_{opt_put} \leq a_{writes} \\ a_{writes}, & \text{sonst} \end{cases}$$

4.2.4 Zusammenfassung

Die Grundidee ist die Konsistenzerhaltung der Daten mit Hilfe des BASE-Algorithmus. Dabei werden Synchronisationskonflikte mit dem System IceCube aufgelöst. BASE und IceCube setzen die Verwendung der Operationen voraus, die als Grundelemente in diesem System gesehen werden. Dadurch, dass die Cloud weniger als 100 % Verfügbarkeit hat, wird Peer-to-Peer-Kommunikation als Fail-Over-Lösung verwendet, um Aktualisierungen auszutauschen. Die Kernpunkte der Lösung für die einzelnen Anforderungen werden in folgenden Tabellen dargestellt.

Funktionale Anforderungen	**Strategie**
Bereitstellung eines verteilten Datenspeichers	Daten über Cloud und Caches verteilt
Cloud als zentraler Datenspeicher	Daten im Blob-Speicher der Cloud halten
Lokale Kopie im Cache	Replikas im lokalen Speicher
Flache Datenstruktur	Blobs, nicht hierarchisch aufgeteilt
Unterstützung der CRUD-Operationen auf Daten	Operations
Hinzuschalten weiterer Operationen	Schnittstelle für DLLs
Automatische und manuelle Konfliktauflösung	Zusammenführung der Aktionslogs
Integritätsprüfung der lokalen Daten	Liste mit Hashwerten der Datensatzinhalte
Eventual Consistency	BASE, Flooding

Nichtfunktionale Anforderungen	**Strategie**
Offlineverfügbarkeit- und Editierbarkeit	BASE
Toleranz gegenüber Cloud- und Knotenausfällen	Peer-to-Peer als Failover-Kommunikation
Synchronisation mit der Cloud bei bestehender Verbindung	Zeitstempelvergleich der letzten Änderungen, Rsync
Minimierung der Speicherkosten und Optimierung der Kommunikation zwischen Cloud und Cache	Selektive Synchronisation, Komprimierung, Optimierungsfunktion
Schutz gegen unberechtigtes Schreiben	Schwellenwert für gleiche Einträge
Portabilität der Lösung	Webservices, WSDL
Reduktion der Wahrscheinlichkeit für manuelle Konfliktauflösung	Maschinelle Konfliktauflösung

Nachdem die einzelnen Algorithmen diskutiert wurden, wird nun die Stelle betrachtet, an der BASE und IceCube verzahnt werden. Der gemeinsame Ausgangspunkt beider Algorithmen sind Operationen. Der folgende Abschnitt geht auf die Rolle der Operationen ein und beschäftigt sich detailliert mit ihrem Einsatz in diesem

System. Die Sammlung der Mechanismen für die Datenverarbeitung mittels Operationen wird in dieser Arbeit als Operations bezeichnet. Operations kann als ein Baustein des Systems gesehen werden.

4.3 Operations-Komponente

Entsprechend den Überlegungen in Abschnitt 4.2.1, wird ein ähnlicher Ansatz erfordert, wie in [PB99] und [KRSD01]: eine Strategie, bei der Aktionen für Manipulation der Daten bereitgestellt werden und diese beim Ausführen, für einzelne Datensätze mitgeschnitten werden. Eine Komponente, die diese Strategie realisiert, wird zwischen die Anwendung und die Rohdaten geschaltet (Abbildung 10).

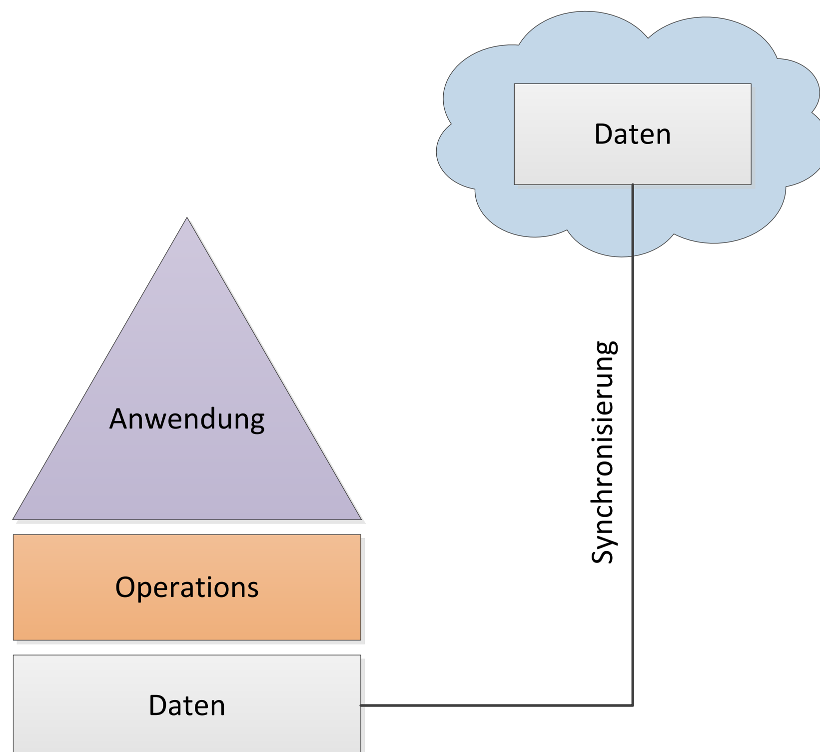

Abbildung 10: Aufbau der Struktur Anwendung-Operations-Daten. Die Anwendung greift auf Daten ausschließlich über Operations zu. Das interne Datenmanagement bleibt der Anwendung verborgen, es kann jedoch mit frei programmierbaren Operationen offenbart werden.

Um die Konfliktwahrscheinlichkeit einzuschränken, müssen höherwertige Abstraktionen der Operationen existieren, z. B. neben CRUD-Operationen auch die, die Anwendungslogik- und Semantik berücksichtigen. Betrachten wir das folgende Beispiel: Mehrere Teilnehmer tragen gleichzeitig jeweils einen Eintrag für den gleichen Termin in eine Kalenderdatei ein. Auf der CRUD-Ebene würde es zum Konflikt führen, denn eine Tabellenzelle der Datei kann nur einen Eintrag enthalten. Entsprechend der Anwendungslogik wäre diese Handlung jedoch erlaubt, denn es würde für mehrere gleichzeitige Termine ein einziger, komplexer Eintrag gespeichert werden, der von der Anwendung interpretiert und in einer Tabellenzelle gespeichert werden kann.

In nächsten Abschnitten wird das Konzept der Operations erläutert, die die Middleware realisieren. Zunächst wird gezeigt, wie ein Datensatz gespeichert wird und welche Eigenschaften er hat. Danach werden Operationen beleuchtet, die Datensätze bearbeiten. Anschließend wird präsentiert, wie Konflikte durch eine Mergefunktion aufgelöst werden. Danach wird erklärt, wie Operations in bestehende Anwendungen integriert werden können. Schlussendlich wird das Unterkapitel zusammengefasst. Wir beginnen mit der Betrachtung der Datensätze als Informationseinheiten.

4.3.1 Datensatz

Ein Datensatz ist die Informationseinheit, die im System synchronisiert wird. Sie besteht aus drei Teilen. In jedem Fall müssen Nutzdaten gespeichert werden, sonst kann das System nicht sinnvoll verwendet werden. Des Weiteren müssen Metadaten gespeichert werden, um die systeminterne Datenverarbeitung zu organisieren. Anschließend ist ein Log notwendig, um die Verwendung von BASE und IceCube zu ermöglichen.

Ein Datensatz besteht somit aus:

- Inhalt
- Metadaten
- Log

Ein Datensatz kann Text- sowie Binärinhalte beinhalten. Die Endanwendung entscheidet über den Dateityp des Datensatzes.

Als Metadaten kann gespeichert werden, ob der Datensatz schreibgeschützt ist, seine Zugriffsrechte, der Datensatztyp und sonstige für Operationen relevante Informationen. Außerdem wird dort festgehalten, welche Operationen den Datensatz bearbeiten dürfen. Dies wird in Form von einer Whitelist und einer Blacklist bewerkstelligt. Nur die in der Whitelist eingetragenen Operationen dürfen den Datensatz bearbeiten und die Operationen, die in der Blacklist definiert wurden dürfen den Datensatz nicht bearbeiten. Die Operation, die den Datensatz erstellt, legt fest, ob die White- oder Blacklist verwendet werden soll und welche Operationen auf den Datensatz Zugriff haben.

Der Log wird mit jeder Ausführung der Operationen geschrieben. Dort werden Informationen über vergangene und die anstehende Operation inklusive Parameter, des aktuellen Zeitstempels und des Erfolgs der Operationsausführung festgehalten.

Formal ist der Datensatz ein Tupel aus dem Datensatznamen, Datensatzinhalten, Datensatz-Metadaten und dem dazugehörigen Log:
$Dataset := (Name, Content, Metadata, Log)$

Operationen, die die eben beschriebenen Datensätze bearbeiten, werden nun ebenfalls detailliert betrachtet.

4.3.2 Operation

Eine Operation ist eine Funktion, die Datensätze bearbeiten kann. Jede Funktion hat die Ein- und Ausgabe, so auch die Operation. Sie benötigt Informationen über die zu verarbeitende Datensätze, also die Eingabequelle und das Ausgabeziel. Daten werden jedoch nicht nur aus einem Datensatz gelesen, sondern können vom Benutzer oder einem anderen System vorgegeben sein. Diese werden über einen Parameter übergeben. Bei der Datenverarbeitung gibt es Einflussfaktoren, die unbedingt zu berücksichtigen sind. Hierfür gibt es einen Informationskanal, über den die Operation diese Daten erhält.

Dementsprechend hat jede Operation die folgende Form:
$OperationName(Source, Destination, Payload, SpecificParameters)$

Dabei enthalten die Parameter Referenzen auf die Quelle und das Ziel der Daten, sowie einen binären Dateninput. Um eine hohe Diversität der Operationen zu ermöglichen, gibt es eine zusätzliche Variable, in der weitere, operationsspezifische Parameter übergeben werden können. Die Ausgabe der verarbeiteten Daten kann durch den Rückgabewert der Funktion und durch das Schreiben in einen Datensatz stattfinden. Die Parameter einer Operation sind in Abbildung 11 visualisiert.

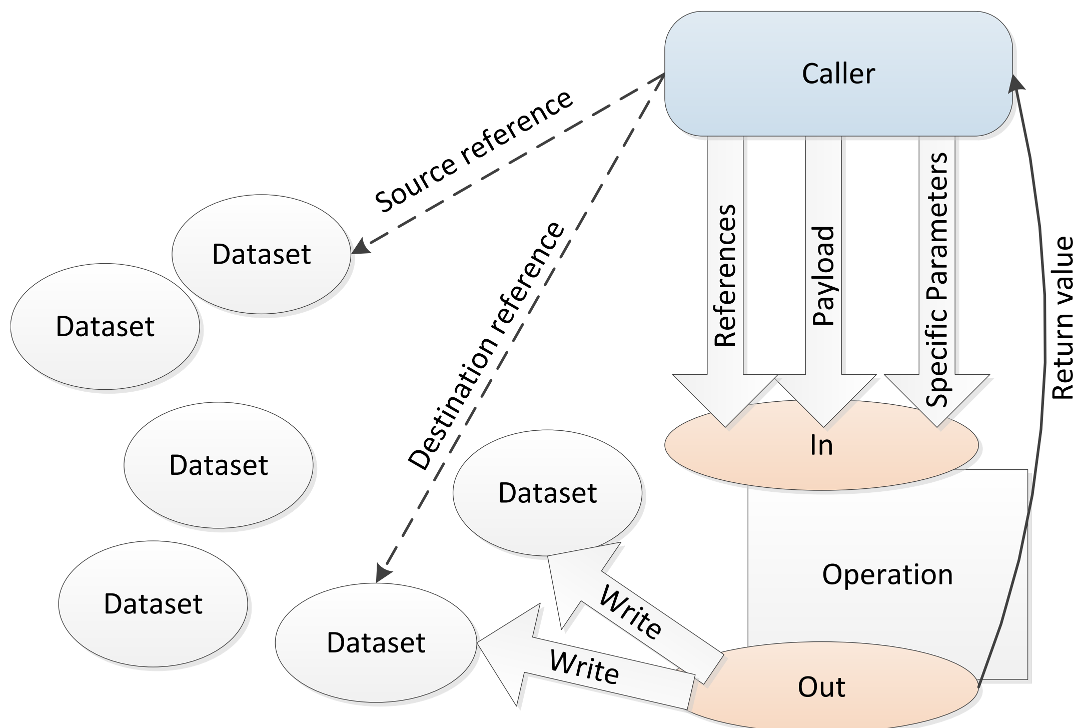

Abbildung 11: Parameter einer Operation. Die Operation bekommt Referenzen auf die Quelle und das Ziel, sowie die für die Verarbeitung nötigen Daten. Die Ausgabe muss nicht in den unter „Destination“ angegebenen Datensatz erfolgen, denn es kann auch ein anderer Datensatz als Ausgabeziel verwendet werden.

Im Folgenden sind einige Beispiele der möglichen Operationen aufgeführt:

- *Rename*(“*orange.txt*”, “*banana.txt*”, $NULL, NULL$)
  Mit dieser Operation wird ein Datensatz mit dem Namen orange.txt in banana.txt umbenannt.

- $AppendText(NULL, \text{“}banana.txt\text{”}, \text{“}Very_fruity_banana.\text{”}, NULL)$
  Diese Operation hängt an das Ende des Datensatzes banana.txt eine Zeichenkette an.
- $Delete(\text{“}banana.txt\text{”}, NULL, NULL, NULL)$
  Der Datensatz banana.txt wird mit Hilfe dieser Operation gelöscht.

Die Instanz einer Operation enthält zusätzlich den Erfolg, Zeitstempel des Starts und dem Ende der Ausführung und eine eindeutige Id :
$OperationInstance = OperationName \cup$
$\{Source, Destination, Payload, SpecificParameters\} \cup$
$\{Guid, Success, TimestampStart, TimestampEnd\}$

Der Erfolg der Ausführung ist für die Integrität des Datensatzes notwendig. Wird es nicht berücksichtigt, dann kann bei der Konfliktauflösung (Abschnitt 4.3.4) zu einer Verfälschung der Daten kommen. Die Zeitstempel sind für die anwendungsspezifischen Konfliktauflösungsstrategien notwendig (vgl. Last Write Wins in Abschnitt 2.1.4). Die eindeutige Id ist ebenfalls für die Konfliktauflösung wichtig, um Gemeinsamkeiten und Unterschiede zu erkennen, falls Operationen mit gleichen Parametern ausgeführt wurden. Wird es missachtet, dann werden gleiche Operationsinstanzen doppelt ausgeführt.

Es bietet sich an, Operationen bereitzustellen, die mit geringem Modifikationsaufwand als Grundlage für die individuellen Anwendungen dienen können und somit für verschiedene Anwendungen wiederverwendbar sind. Das ermöglicht ein höherwertiges Datenmanagement im ganzen System und ermöglicht einfachere Entwicklung von anwendungsspezifischen Operationen.

Somit ergibt sich eine Hierarchie von Operationen, die aus drei Schichten besteht (Abbildung 12).

Im Folgenden wird auf die Basisoperationen, Erweiterungsoperationen und anwendungsspezifischen Operationen näher eingegangen.

Basisoperationen

Die Basisoperationen stellen die grundlegenden Operationen bereit, mit denen Arbeit mit Dateien ermöglicht wird. Für die Datenbearbeitung werden die CRUD-

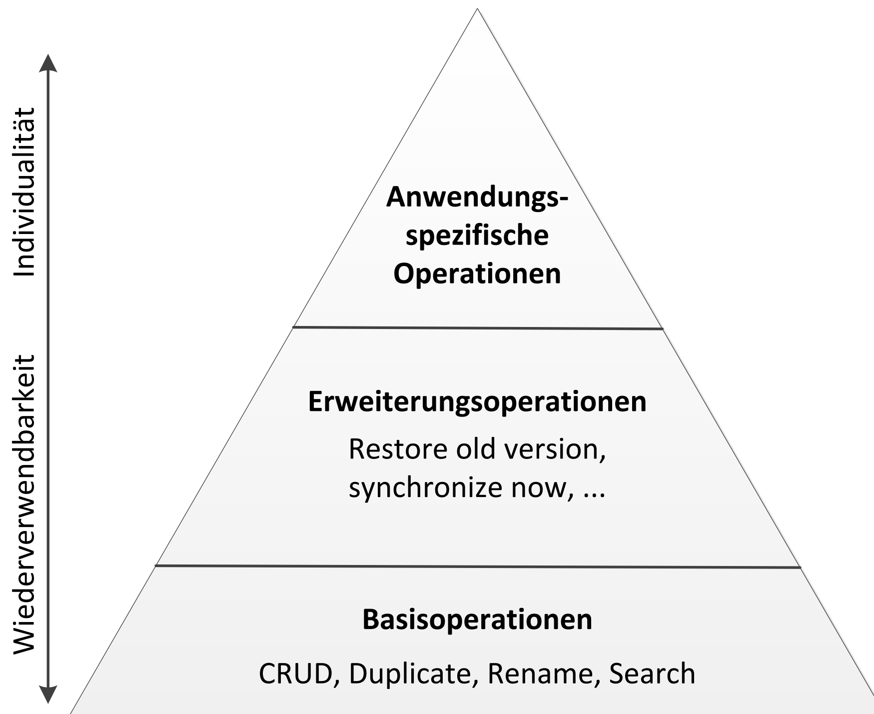

Abbildung 12: Hierarchie von Operationen. Die oberste Schicht ist frei programmierbar und für jede Anwendung individuell. Die mittlere kann wiederverwendet und erweitert werden. Die untere Schicht bietet die Grundlage und ist fest.

Operationen bereitgestellt

- Create
- Read
- Update
- Delete

Des Weiteren werden die grundlegenden Operationen bereitgestellt, die das Verwalten der Datensätze ermöglichen:

- Search by name (ähnlich List)
- Search by metadata

- Rename (ähnlich Move)
- Duplicate (ähnlich Copy/Paste)

Die Operation „Search by name“ ermöglicht eine Suche nach Namen eines Datensatzes. In einer flachen Datenordnung macht die Operation „List directory content“ keinen Sinn, denn es würden alle gespeicherten Datensätze ungefiltert zurückgegeben werden. Ist eine Auflistung der Datensätze mit bestimmten Kriterien erwünscht, muss „Search by metadata“ ausgeführt werden. Allerdings müssen diese Kriterien als Metadaten des Datensatzes gespeichert sein. Um die Suche nach Datensatz-Inhalten zu realisieren, muss die Operation, aufgrund der anwendungsspezifischen Daten die anwendungsspezifische Logik beinhalten, die nicht für jede Anwendung generisch realisiert werden kann. Sollte diese Funktion von Nöten sein, dann muss sie als eine anwendungsspezifische Operation implementiert werden.

Die Funktionalität von „Move“ und „Copy/Paste“ ist sinnvoll in einer hierarchischen Organisation der Daten. Eine Move-Operation in einer flachen Struktur bewirkt nur eine Namensänderung, deshalb ist der Operationsname „Rename“ sinnvoller. Ebenfalls bewirkt „Copy/Paste“ nur eine Verdopplung des Datensatzes („Duplicate“) im Speicher.

Die Operation „Link“, die eine Verknüpfung zu einer anderen Datei erstellt, wird aufgrund der Anforderung „Flache Datenstruktur“ nicht unterstützt, denn sie würde eine Relation zwischen zwei Datensätzen erstellen.

Die Basisoperationen können durch weitere ergänzt werden, um komplexere Datenzugriffe oder Steuerungsvorgänge und somit effiziente Arbeit mit Daten zu ermöglichen. Diese können je nach Anwendungsfall nützlich sein und durch andere Anwendungen wiederverwendet werden.

Erweiterungsoperationen

Basisoperationen, die hauptsächlich für die Datenverarbeitung bereitstehen bieten keine Möglichkeit Einfluss auf die systeminterne Vorgänge zu nehmen. So könnte man mit einer Erweiterungsoperation einen Synchronisierungsvorgang erzwingen, um beispielsweise die Aktualität der Daten zu einem bestimmten Zeitpunkt zu garantieren. Weitere nützliche Operationen sind:

- Shred dataset - Löscht alle Versionen eines Datensatzes in der Cloud und im

Cache aller Peers

- Entangle two datasets - Verschränkt zwei Datensätze miteinander, die beim Synchronisieren atomar behandelt werden
- Restore old version - Überschreibt die aktuelle Version durch eine ältere

Basisoperationen und Erweiterungsoperationen bieten nicht die bestmögliche Flexibilität für die Endanwendung. Sie haben den Zweck, Daten zu verarbeiten und Steuerung von Operations zu ermöglichen. Um das volle Potential des Systems zu nutzen, werden anwendungsspezifische Operationen benötigt, die auf die Endanwendung zugeschnitten sind.

Anwendungsspezifische Operationen

Die anwendungsspezifischen Operationen müssen je nach Anwendungslogik individuell entwickelt werden. Sie können beispielsweise die Operationen der Tuple Spaces implementieren, was einer möglichen Endanwendung entspricht.

Tuple Spaces ist ein assoziativer Speicher, der aus der Sicht eines datenkonsumierenden Prozesses eine Art zentraler Speicher mit Daten in Form von Tupeln ist. Diese Tupeln werden in einem Blackboard-System (vgl. [Cor91]) gespeichert, dabei kann ein Tupel von genau einem Prozess verarbeitet werden. Es geschieht, indem ein Tupel aus dem Tuple Space entnommen und dem Prozess übergeben wird. Dieser verarbeitet diesen Tupel und kann ihn wieder zurück, in das Tuple Space einfügen.

Abgebildet auf das entwickelte System wäre der zentrale Speicher die Cloud und Tupeln wären Datenstrukturen, die in Flatfiles serialisiert gespeichert werden. Das Verwenden der Tupel durch einen Prozess geschieht mit einer Operation. Für die Lösung wäre der Einsatz von Tupel Spaces eine spezifische Anwendung. So müssen nach Foster [Fos95] nur die folgenden speziellen Operationen realisiert werden:

- Einfügen des Tupels
- Blockierendes Lesen des Tupels
- Nicht blockierendes Lesen des Tupels
- Blockierendes Lesen und Entfernen des Tupels
- Nicht blockierendes Lesen und Entfernen des Tupels

> „Five operations are supported: insert (out), blocking read (rd), non-blocking read (rdp), blocking read and delete (in), and nonblocking read and delete (inp)“ [Fos95]

Durch den exklusiven Ausschluss des Zugriffs auf die Tupel kann es in diesem Anwendungsfall zu keinen Konflikten kommen.

In vielen Systemen ist es nicht möglich den exklusiven Ausschluss zu realisieren, wodurch es zu Konflikten kommen kann. Jedoch bevor ein Konflikt aufgelöst werden kann, muss ein Log der ausgeführten Operationen existieren.

4.3.3 Log

Entsprechend dem System BASE, gibt es für jeden Datensatz genau einen Log, der mit einem neuen Datensatz erstellt wird und mit dem Löschen eines Datensatzes entfernt wird. Jede Replika des Datensatzes enthält somit den Endzustand nach der Ausführung der Operationen, sowie einen zugehörigen Log als Protokoll jeder Änderung, der für IceCube verwendet werden kann. Ein Log ist somit eine Liste mit durchgeführten Operationen und der aktuell anstehenden Operation am entsprechenden Datensatz. Formal wird er wie folgt beschrieben.

$$Log = \{OperationInstance_1, ..., OperationInstance_n\} \cup OperationInstance_{current}$$

Existieren zwei unterschiedliche Logs desselben Datensatzes, so können diese Logs mit der Mergefunktion zusammengeführt werden.

4.3.4 Mergefunktion

Die Mergefunktion wird benötigt, sobald zwei Zustände eines Datensatzes in einen überführt werden müssen. Daten sollen auch im Peer-to-Peer-Modus zusammengeführt werden, deshalb muss die Ausführung der Mergefunktion im Cache stattfinden. In der Cloud wird eine andere Mergefunktion benötigt, die die Anforderung „Schutz gegen unberechtigtes Schreiben“ realisiert. Deshalb werden in dieser Lösung zwei unterschiedliche Mergefunktionen eingesetzt: Cloud Merge und Peer Merge.

Cloud Merge

Sobald eine bestimmte Anzahl an gleichen Veränderungsanfragen in der Cloud erreicht wird, wird die Änderung übernommen und für alle lesbar gemacht. Dies resultiert in einem neuen Zustand des betroffenen Datensatzes. Dieser Schwellenwert kann für einzelne Datensätze beliebig festgelegt werden. So ist der Einsatz des Systems in einer öffentlichen, sowie vertrauenswürdigen Umgebung möglich. Liegt der Wert bei 1, dann werden Änderungen sofort übernommen. Außerdem ist es sinnvoll, bei Datensätzen, die oft geändert werden, diesen Wert zu erhöhen, um eine Stabilität des Zustands zu gewährleisten. Entsprechend soll der Wert bei Datensätzen, die selten geändert werden - klein sein, damit die Änderungen zeitnah vorgenommen werden.

Ein quorumbasiertes Verfahren ist in dem Fall nicht einsetzbar, weil man bei jeder Änderung oder regelmäßig die Gesamtanzahl an aktiven Knoten ermittelt müsste, was ineffizient wäre. Es würde mit der Anforderung „Minimierung der Speicherkosten und Optimierung der Kommunikation zwischen Cloud und Cache“ kollidieren, weil hohe Transferkosten aufgrund dieses Verhaltens anfallen würden.

Um eine gewisse Toleranz zwischen den Abweichungen der Änderungsanfragen festzulegen, wird eine anwendungsspezifische Funktion $Equals$ verwendet. Sie vergleicht Datensätze miteinander und entscheidet, ob die Abweichung unter der Toleranzgrenze liegt. Mit dieser Information kann man die Zugehörigkeit zu Äquivalenzklassen prüfen. Nach dem Bilden der Äquivalenzklassen wird die größte Äquivalenzklasse gewählt und aus dieser Menge die neue Datensatzversion zufällig bestimmt.

Um Raceconditions zwischen zwei Äquivalenzklassen zu vermeiden, müssen Caches, deren Änderungsanfrage abgelehnt wurde, vor der nächsten Änderungsanfrage auf die neue Version des Datensatzes gebracht werden. Hierfür steht die Funktion „Peer Merge“ bereit, die im Folgenden diskutiert wird.

Betrachten wir folgendes Beispiel: Sei eine Version des Datensatzes in der Cloud d_0 und Änderungsanfragen $\{d_1, ..., d_5\}$. Dabei ergibt die $Equals$-Funktion folgendes (zwecks Übersichtlichkeit wurden die trivialen Kombinationen der Reflexivität, Transitivität und Symmetrie weggelassen):

- $Equals(d_1, d_2) = true$
- $Equals(d_1, d_3) = true$
- $Equals(d_1, d_4) = false$

- $Equals(d_1, d_5) = false$
- $Equals(d_2, d_3) = true$
- $Equals(d_2, d_4) = false$
- $Equals(d_2, d_5) = false$
- $Equals(d_3, d_4) = false$
- $Equals(d_3, d_5) = false$
- $Equals(d_4, d_5) = true$

Aufgrund dieser Berechnung können folgende zwei Äquivalenzklassen gebildet werden:
$[a_1] = \{d_1, d_2, d_3\}$ und $[a_2] = \{d_4, d_5\}$

Mit $|[a_1]| = 3$ und $|[a_2]| = 2$ bildet die erste Äquivalenzklasse die Mehrheit. Nun wird ein zufälliger Kandidat aus dieser Menge, z. B. d_2 mit dem zugehörigen Log und Metadaten als neue Version des Datensatzes in der Cloud gewählt und für alle Caches lesbar gemacht:
$CloudMerge(d_0, \{d_1, ..., d_5\}) = d_2$

Letztendlich sollen Caches, die die Änderungsanfragen d_4 und d_5 machten, synchronisiert werden. Hierfür laden sie den Datensatz d_2 aus der Cloud herunter und führen d_4 und d_2, bzw. d_5 und d_2 mit Peer Merge zusammen.

Peer Merge

Ein Konflikt tritt immer dann auf, wenn während eines Synchronisierungsvorgangs eine Datensatzversion auf den Knoten geladen wird, die von der lokalen Version abweicht. Eine Abweichung liegt dann vor, wenn die zufällig generierten Instanz-Ids der letzten Operationen unterschiedlich sind. Das ist ein Zeichen dafür, dass Änderungen, die an zwei Stellen durchgeführt wurden, nicht synchron sind. Wurde ein Konflikt mittels Vergleich der Ids erkannt, kann die Konfliktauflösung durchgeführt werden.

Konfliktauflösung

Die Konfliktauflösung auf dem Knoten erfolgt schrittweise: kann ein Konflikt im ersten Schritt nicht aufgelöst werden, wird es im nächsten Schritt versucht. Der Ablauf

ist als Zustandsdiagramm in Abbildung 13 dargestellt. Die erste Stufe führt Datenzustände mit einem IceCube-ähnlichen Algorithmus zusammen, indem eine gemeinsame Reihenfolge der Operationen für beide Logs festgelegt wird. Dabei entsteht ein gemeinsamer konfliktfreier Zustand. Die zweite Stufe ist eine von der speziellen Anwendung festgelegte Routine. Dieser Funktion werden zwei Datensatz-Versionen übermittelt, auf Basis deren sie eine dritte, konfliktfreie Version zurückgibt. Die dritte Stufe ist die manuelle Auflösung und wird durch den Benutzer vorgenommen. Ihm werden zwei Versionen vorgeschlagen und er wählt eine der Versionen oder übermittelt eine dritte, von ihm persönlich kreierte Version.

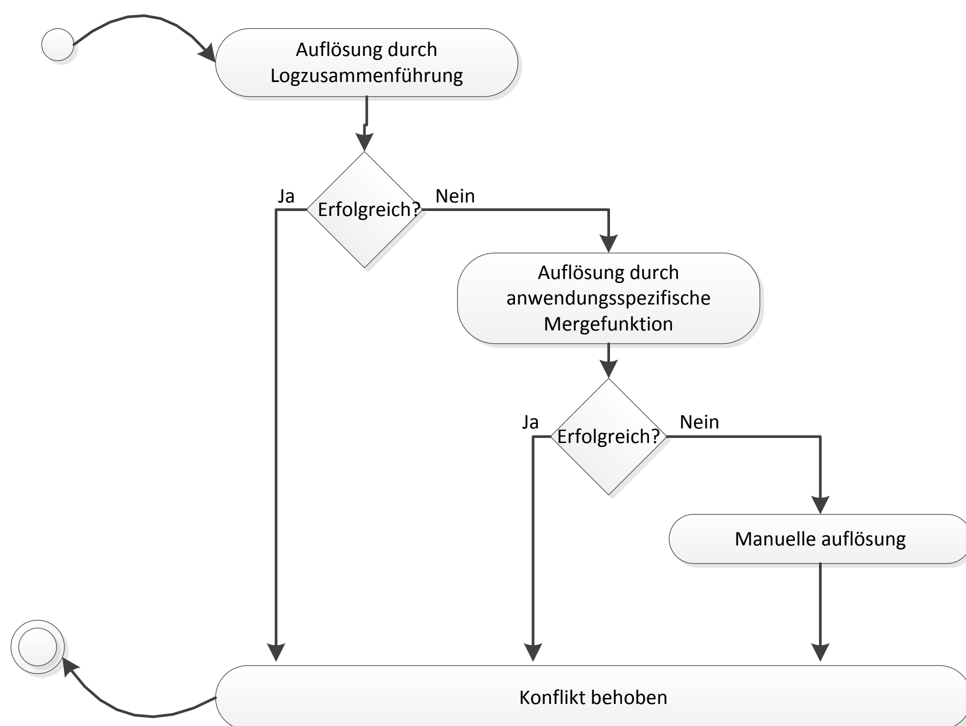

Abbildung 13: Ablauf der Konfliktauflösung auf dem Knoten. Peer Merge besteht aus mehreren Teilen, die Konfliktauflösung übernehmen.

Ein solcher dreistufiger Resolutions-Ablauf ist in keinem der in dieser Arbeit vorgestellten Systeme vorhanden, jedoch dem Ansatz in IceCube und Bayou ähnlich. IceCube verwendet nur die erste und dritte Stufe, Bayou nur die zweite und dritte.

Alternativ könnte man die anwendungsspezifische Auflösung als ersten Schritt ein-

setzen. Jedoch wird in dieser Arbeit davon ausgegangen, dass die anwendungsspezifische Auflösung nur für wenige Datensatztypen, ähnlich wie in Coda, bereitsteht. Entsprechend ist es besser, ein universelles Auflösungsverfahren zuerst einzusetzen, um bereits in der ersten Stufe die meisten Konflikte aufzulösen. Aus diesem Grund wird der erste Ablauf (Abbildung 13) in dieser Lösung eingesetzt.

Erste Stufe

Die zweite und dritte Stufen sind trivial, jedoch die erste umso komplexer. Der IceCube-Ansatz wurde in Abschnitt 2.2.8 vorgestellt. In IceCube müssen zu jeder Operation alle Vorbedingungen genau definiert werden, damit die Operation beim Zusammenführen der Logs vom Algorithmus an die richtige Stelle gesetzt wird. Dies ist in einem komplexen System sehr aufwendig. Des Weiteren verlieren Operationen auf diese Weise an Wartbarkeit, denn die Vorbedingungen müssen immer mit der Funktion genau korrelieren. Deshalb muss der IceCube-Algorithmus an dieser Stelle angepasst werden.

Der Ablauf ist nun wie folgt. Als Erstes muss ein gemeinsamer Ausgangspunkt beider Logs gefunden werden. Die Logs werden deshalb operationsweise durchgegangen und die Ids der Operationinstanzen verglichen. Die Operation vor der ersten Diskrepanz stellt den gemeinsamen Ausgangspunkt dar, ab dem zusammengeführt werden soll. Als Nächstes werden Operationen gesucht, die aufgrund von Abhängigkeiten die Permutierungsfreiheit einschränken. Diese Operationen definieren einen Rahmen, in dem einzelne Logteile mittels IceCube zusammengeführt werden können. Dieses Vorgehen reduziert die Anzahl und die Komplexität der Vorbedingungen für den originalen IceCube-Algorithmus.

Entsprechend muss das Operations-System in der ersten Stufe die Operationen mit Abhängigkeiten erkennen. Die Beurteilungskriterien für solche Operationen werden im nächsten Abschnitt diskutiert.

4.3.5 Qualitätsstufen der Operationen

Operationen können in unterschiedliche Qualitätsstufen eingeteilt werden. In diesem Dokument werden drei Stufen verwendet. Um die Beschreibung zu vereinfachen, werden einzelnen Stufen Farben zugeordnet. Das Qualitätskriterium ist die, aus

den Operationen resultierende, Komplexität des Mergevorgangs. Können Operationen sehr effizient zusammengeführt werden, dann haben sie eine gute Qualität und entsprechen der Farbe Grün. Ist mit dem Mergen ein erheblicher Rechenaufwand verbunden, dann sind diese Operationen qualitativ mittelmäßig und entsprechen der Farbe Gelb. Behindern Operationen die Zusammenführung, dann ist die Qualität schlecht und es wird ihnen die Farbe Rot zugeordnet.

Die Effizienz der Logzusammenführung hängt mit den mathematischen Eigenschaften der Operationen zusammen. Ist die Reihenfolge der Operationen im Log irrelevant, dann sind sie kommutativ und können dementsprechend an beliebiger Stelle im konfliktfreien Log eingesetzt werden. Wenn es eine kausale Abhängigkeit zwischen zwei Datenzuständen gibt, dann ist es eine Einschränkung und die richtige Reihenfolge muss zuerst berechnet werden. Werden nichtkommutative und kausal abhängige Operationen verwendet, dann ist der Lösungsraum der Mergefunktion extrem gering und kann unter Umständen keine Lösung enthalten. Diese Verhältnisse lassen sich in folgender Tabelle veranschaulichen:

Qualität	**Farbe**	**Kommutativ**	**Kausal unabhängig**
Schlecht	Rot	-	$\surd$/-
Mittelmäßig	Gelb	$\surd$	-
Gut	Grün	$\surd$	$\surd$

Die CRUD-Operationen sind als rot einzustufen, denn mit Read und Write lässt sich eine kausale Abhängigkeit herstellen. Des Weiteren sind Create und Delete nicht kommutativ. Aus diesem Grund, wirkt sich die Verwendung von anwendungsspezifischen grünen Funktionen, statt den CRUD-Operationen, positiv auf die Leistung des Mergealgorithmus aus.

Grüne Operationen sind besser

Es empfiehlt sich der verstärkte Einsatz von grünen Operationen, die kommutativ und kausal unabhängig sind, um eine bessere Konfliktauflösung zu ermöglichen. Auch in [GHOS96] wird gesagt, dass die Verwendung von kommutativen Operationen die Konvergenz der Datensätze gegen einen konsistenten Zustand unterstützt, weil sie einfach zusammengeführt werden können.

Betrachten wir das folgende Beispiel: In einem Club mit zwei Ein- und Ausgängen soll die aktuelle Anzahl an Besuchern gezählt werden. Es gibt eine Datei mit einem

numerischen Wert 0, der zweimal hochgezählt wird. Das inkrementieren kann mit einer anwendungsspezifischen Operation Increment und mit den CRUD-Operationen Read und Write realisiert werden.

Zähler	**Anw.spez.-Operation**	**CRUD-Operation**	**Dateiinhalt**
1		a = Read()	0
1	Increment()	Write(a+1)	1
2		b = Read()	0
2	Increment()	Write(b+1)	1

In beiden Fällen werden insgesamt zwei Besucher an zwei Eingängen gezählt. Nun werden diese Dateien synchronisiert. Ein Konflikt mit Zwei aufeinanderfolgenden Operationen $Increment()$ lässt sich einfach auflösen, weil die Reihenfolge beliebig sein kann.

Die Reihenfolge der aufeinanderfolgenden *Read*() und $Write()$ Operationen darf nicht geändert werden, weil es Abhängigkeiten zwischen den Funktionen gibt, die bei einer Neuanordnung verletzt werden würden. Deshalb muss dieser Konflikt durch die zweite Stufe des Mergealgorithmus aufgelöst werden. Demnach muss die anwendungsspezifische Mergefunktion diesen Fall kennen und entsprechend die Werte beider Zähler addieren.

Rote Operationen in grüne umwandeln

Möchte man Werte schreiben, die auf gelesenen basieren, dann muss das Lesen und Schreiben atomar innerhalb einer Operation durchgeführt werden. Die Operation $Increment()$ ist nach diesem Muster entwickelt, denn zunächst wird ein Wert gelesen, dann inkrementiert und abschließend in die Datei geschrieben. Die Folge ist: es kann kein veralteter Wert aus der Vergangenheit gelesen werden. Entsprechend kann auch kein Wert mit veralteten Daten überschrieben werden. Dies stellt lokal einen gewissen Isolierungs-Grad sicher (vgl. [ALO00]).

Auf jedem Knoten kann auf den Datensatz gleichzeitig nur eine Operation zugreifen. Entsprechend wird auch nach dem Zusammenführen zweier Logs eine Operation nach der anderen ausgeführt. Die serielle Ausführung der in sich abgeschlossenen Operationen gleicht der Anwendung der Read- und Write-Locks in einer Datenbank mit dem Isolierungsgrad SERIALIZABLE.

Beispiel für die Logzusammenführung mit roten Operationen

Manchmal ist die Anwendung roter Operationen aufgrund der Anwendungslogik unumgänglich. Werden kommutative und nichtkommutative Operationen an einem Datensatz ausgeführt, dann bilden die nichtkommutativen, entsprechend dem Vorgehen in der ersten Stufe, einen Rahmen für die kommutativen Operationen. So können die kommutativen Operationen innerhalb dieses Rahmens zusammengeführt werden.

Als Beispiel dienen uns zwei Zähler, die ihre Messungen in eine Datei schreiben, die anschließend synchronisiert wird. Es ist ein Konflikt entstanden und er wird in der ersten Stufe aufgelöst (Abbildung 14).

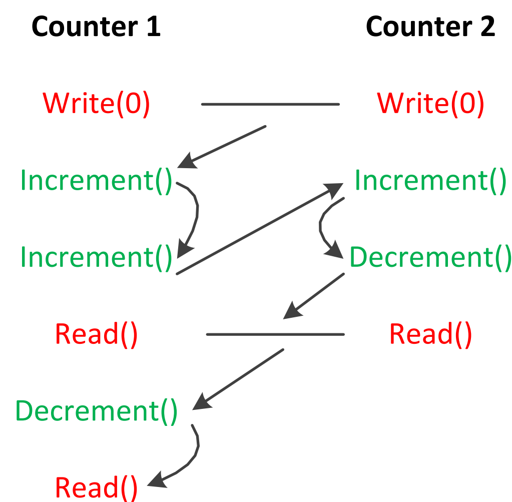

Abbildung 14: Erste Stufe des Peer Merge-Algorithmus. Die permutierbare Operationen werden innerhalb des Rahmens zusammengeführt, der von nicht permutierbaren Operationen gebildet wird.

Die Read- und Write-Operationen werden als Rot eingestuft. Die Increment- und

Decrement-Operationen dagegen als Grün. So wird in Abbildung 14 zunächst der erste Rahmen zwischen Write und Read betrachtet, danach der zweite, der von den beiden Read-Operationen des ersten Zählers gebildet wird. Auf diese Weise werden Logs zusammengeführt und die kausalen Abhängigkeiten bleiben unverletzt.

Operations wurde im Detail diskutiert, jetzt bietet es sich an, aus der Makroperspektive zu untersuchen, wie es in Entwürfen anderer Systeme präsent wird. Hierfür wird Operations als Baustein gesehen, der in Architekturmustern sichtbar wird. Der nächste Abschnitt beschäftigt sich mit der Frage, wie Operations während der Planungsphase eines Systems eingegliedert werden kann.

4.3.6 Integration der Operations in Architekturmuster

Operations ist ein architektonischer Baustein, der bei Verwendung diverser Architekturmuster für den grundlegenden Aufbau einer Anwendung verwendet werden kann.

Die meisten Anwendungen sind nach einem Drei-Schichten- oder MVVM-Architekturmuster entwickelt. Sonstige Architekturmuster kann man auf die genannten abbilden. Aus diesem Grund wird hier nur die Integration der Operations in diese zwei Architekturmuster betrachtet.

Drei-Schichten

Drei-Schichten gehört zu den klassischen Architekturmustern. Es besteht aus Komponenten: Data, Logic und View. Die Komponente Data umfasst Daten und Datenzugriffe für die dazugehörige Verarbeitung in der Komponente Logic. View ist für die Darstellung der Daten verantwortlich. Es ist nicht erlaubt, dass View mit Daten direkt kommuniziert.

Für die Integration der Operations muss der Entwickler nur die Komponente Data abändern, sodass Datenzugriffe über ein Webservice und Operations stattfinden. Operations lässt sich weder in die Komponente Data, noch in Logic einordnen. Es enthält Funktionen, die grundlegende anwendungsspezifische Datenaufbereitung übernehmen. Aus diesem Grund muss es als Bindeglied zwischen Data und Logic gesehen werden. Entsprechen erkennt man in der Abbildung 15, dass diese Komponente die, mit gestrichelten Linien, dargestellte Trennung der Ebenen überbrückt.

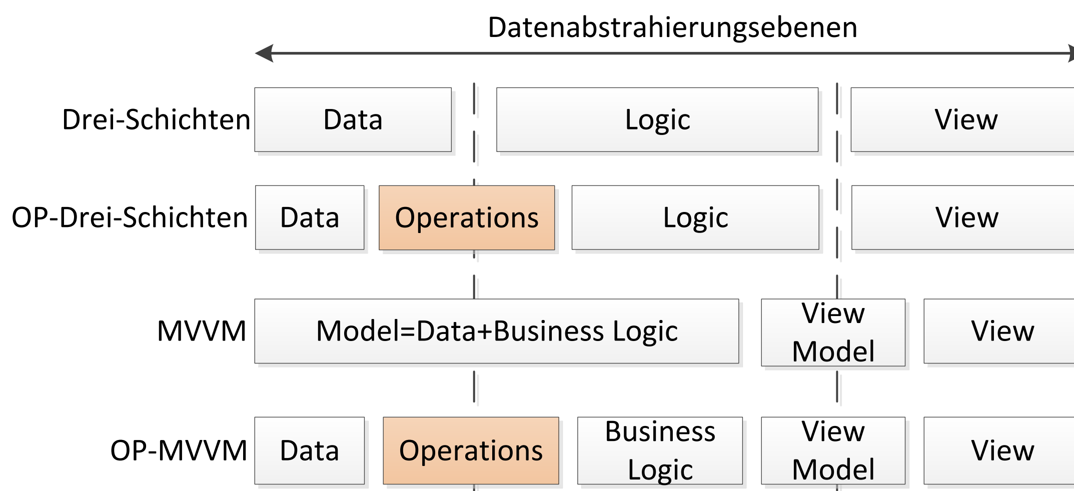

Abbildung 15: Eingliederung der Operations in Drei-Schichten und MVVM. Die Architekturmuster wurden auf eine Gerade abgebildet, die Abstraktionsebenen der Daten darstellt. Operations ist ein Baustein, den man in Architekturmuster zwischen Rohdaten und Logik einbauen kann.

MVVM

Model-View-ViewModel ist ein Architekturmuster, das zwischen dem Model und dem View eine Schicht hat, die Daten für die Präsentation vorbereitet. Die Komponente Model umfasst Daten und die dazugehörige Verarbeitungslogik.

Eine MVVM-Anwendung kann Operations verwenden, nur muss der Entwickler das Model reorganisieren, sodass Daten und Logik von einander entkoppelt werden. Anschließend kann die Operations-Schicht zwischen Daten und Logik geschaltet werden, ähnlich wie zwischen Data und Logic der Drei-Schichten-Architektur (Abbildung 15).

Die Integration von Operations in MVVM resultiert in einer Fünf-Schichten-Architektur:

Daten (Data), anwendungsspezifische Daten (Operations), Verarbeitungslogik (Business Logic), Darstellungslogik (ViewModel), Darstellung (View). Entsprechend ist der Einsatz von Operations in einer Fünf-Schichten-Architektur mit einem geringen Änderungsaufwand verbunden, da nur die zweite Schicht ersetzt werden muss und keine Anpassungen der Grundarchitektur notwendig sind.

Verwendung durch bestehende Anwendungen

Generell lässt sich Operations auch in bestehende Systeme einbauen. Dies erfordert jedoch eine lose Kopplung zwischen Daten und Logik. Am wenigsten Aufwand verursacht die Integration, wenn bei der Endanwendung ein Enterprise Service Bus eingesetzt wird. In diesem Fall muss Operations lediglich an den Message Bus angeschlossen werden und entsprechende Routing- und Transformations-Informationen hinterlegt werden. In anderen Fällen muss ein Adapter für die Anwendung entwickelt werden, der Operationen über Webservices und entsprechende Schnittstelle aufruft.

4.3.7 Zusammenfassung

Operationen sind der Kern der Middleware. Sie bearbeiten Datensätze, die aus Nutzdaten, Metadaten und Logs bestehen und können in eine Hierarchie eingeordnet werden. Die Operationen ermöglichen eine verbesserte Konfliktauflösung durch Mergefunktionen, von denen es zwei gibt - eine in der Cloud, mit der Konflikte für alle Teilnehmer aufgelöst werden und eine auf dem Peer, um lokale Konflikte zu beseitigen. Außerdem können Operationen in verschiedene Qualitätsstufen eingeordnet werden, was für den Peer Merge-Algorithmus von großer Wichtigkeit ist. Der Baustein Operations lässt sich flexibel in die populärsten Entwurfsmuster eingliedern und verwenden, sodass er in der Planungsphase berücksichtigt, oder auch nachträglich in ein bestehendes System integriert werden kann. Das nächste Kapitel geht in die Details der Implementierung von der Middleware.

5 Implementierung

In diesem Kapitel wird beschrieben, wie das entworfene System praktisch umgesetzt wird. Zunächst wird die Architektur des Systems aus dem vorausgehenden Kapitel verfeinert, anschließend werden die verwendeten Entwurfsmuster und ihre Rolle für das Gesamtsystem beleuchtet. Die Schnittstellen bilden einen wichtigen Teil der Lösung und werden in einem separaten Abschnitt betrachtet. Des Weiteren beschäftigen wir uns mit den Abläufen, die durch die Komponenten des Systems gesteuert und ausgeführt werden. Anschließend wird eine Zusammenfassung des Kapitels gegeben.

5.1 Architektur

Die Architektur des Systems kann auf zwei Ebenen betrachtet werden. So werden in diesem Kapitel zuerst die Interaktionen zwischen den einzelnen Umgebungen präsentiert, danach das Zusammenspiel einzelner Komponenten und Klassen.

5.1.1 System

In Abbildung 16 ist eine detailreiche Architektur des Systems dargestellt. In diesem Schaubild erkennt man zusätzlich zu den Grundkomponenten die einzelnen Klassen, die in weiß dargestellt sind und die Schnittstellen. Alle Schnittstellen, die für RPC-Aufrufe verwendet werden, sind grün gekennzeichnet. Die Schnittstellen, die auf Klassen-Ebene eingesetzt werden, sind grau.

Jede Kommunikation zwischen der Cloud und dem Cache findet über die Schnittstelle *ICloudSync* statt. Die Verbindung zwischen Cache und Application Area geschieht über drei Wege. Die Operationsaufrufe und Aufrufe für das manuelle Mergen erfolgen über Webservices. Der Zugriff auf die anwendungsspezifischen Operationen erfolgt über die dynamische Anbindung einer DLL. Die Nutzung des lokalen Speichers erfolgt direkt mit Hilfe des .NET-Frameworks.

Um die Funktionalität genauer zu erörtern, wird nun auf die Mikroebene gewechselt und die einzelnen Komponenten werden fokussiert betrachtet.

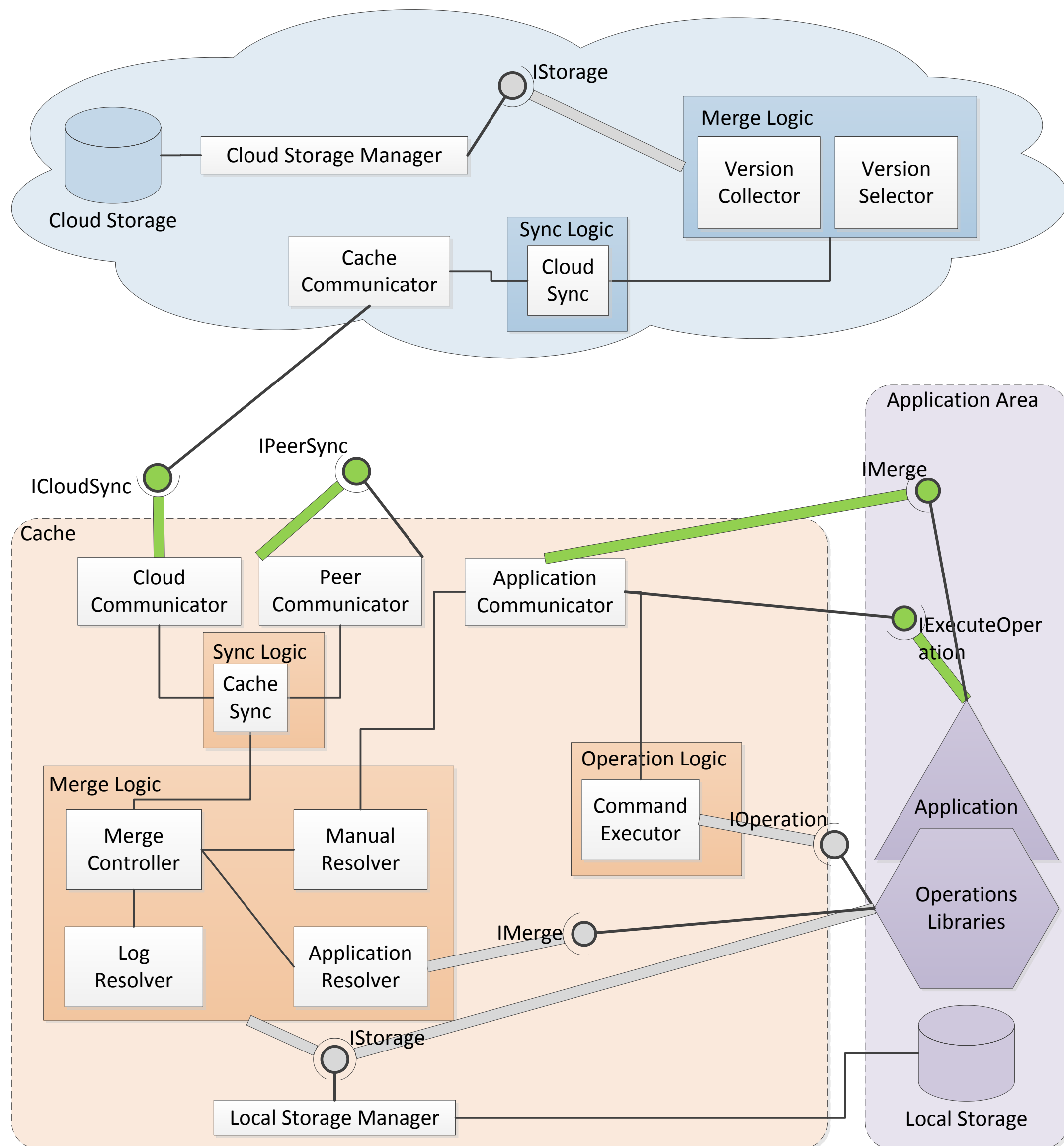

Abbildung 16: Feinarchitektur des Systems. Die Bereiche und Komponenten entsprechen der Abbildung 8. Diese Darstellung enthält die wichtigsten Schnittstellen des Systems.

5.1.2 Komponenten

Cache Sync und *Cloud Sync* sind die beiden Komponenten, die die Synchronisationslogik umsetzen und Daten für das eventuelle Mergen bereitstellen.

Cloud Storage Manager und *Local Storage Manager* ermöglichen den Zugriff auf den jeweiligen Datenspeicher. In der Cloud ist es das *Cloud Storage* und in der Application Area das *Local Storage*.

Cache Communicator, Cloud Communicator, Peer Communicator, Application Communicator machen den Zugriff auf entfernte Objekte und die Ausführung der RPC-Funktionen komfortabel, indem sie die RPC-Aufrufe als einfache Funktionsaufrufe repräsentieren.

Command Executor sorgt für eine sichere Ausführung der Operationen und behandelt die auftretenden Fehler.

Die Komponente *Merge Logic* auf dem Cache besteht aus vier Klassen. *Merge Controller* steuert den Konfliktauflösungsvorgang, der drei weitere Klassen involviert. *Log Resolver* übernimmt die Zusammenführung von zwei Logs. *Application Resolver* ist eine Klasse, die Daten für die Konfliktbeseitigung durch die Applikation vorbereitet und diese Ausführt. *Manual Resolver* bereitet Daten für die manuelle Resolution durch den Benutzer vor.

Merge Logic in der Cloud beinhaltet andere Klassen, weil sie andere Konfliktauflösungsalgorithmen einsetzt. Sie besteht aus zwei Klassen. *Version Collector* sammelt Änderungsanfragen, die von Caches eingebracht werden. Des Weiteren triggert er den *Version Selector*, der aus einer Menge von Änderungsvorschlägen die Nachfolger-Version wählt. Außerdem kann *Version Selector* ältere Versionen eines konkreten Datensatzes auf explizite Anfrage zurückgeben, sofern der Cloud-Anbieter diese Funktion unterstützt.

Das entworfene System kann mit jeder objektorientierten Sprache umgesetzt werden, z. B. C#, C++, Java, etc. und kann mit allen Cloud Computing-Plattformen betrieben werden, solange der Anbieter IaaS oder PaaS bereitstellt und offene Standards für die Webkommunikation einsetzt. In der praktischen Ausarbeitung wird C# und Windows Azure eingesetzt. Verwendet wird PaaS des Cloud Computings, weil die Infrastrukturebene nicht im Fokus des praktischen Teils dieser Arbeit liegt. Ent-

sprechend wird auf das zugrundeliegende Betriebssystem nicht eingegangen. Bietet der Anbieter nur IaaS an, dann muss zuerst ein Betriebssystem aufgesetzt werden und anschließend die für die Lösung benötigten Bibliotheken (z. B. die des .NET-Frameworks). In jedem Fall hat die Verwendung von IaaS keinen Einfluss auf die Technologien oder Algorithmen dieser Lösung, da der Entwurf in allgemeiner Form durchgeführt wurde.

Um den einzelnen Klassen innerhalb der Komponenten eine Grundstruktur zu verleihen, wurden Entwurfsmuster eingesetzt. Im nächsten Abschnitt werden sie einzeln, bezogen auf die Architektur und Funktion vorgestellt.

5.2 Verwendete Entwurfsmuster

Die in der Lösung verwendeten Entwurfsmuster sind: Singleton, Proxy, Abstract-Factory, Command, Memento, Template Method, Mediator.

5.2.1 Singleton

Das Singleton-Entwurfsmuster erlaubt die Erzeugung maximal eines Objektes von einem Typ. Alle Communicators, Storage-, Merge- und Sync-Komponenten sind singleton. Nur die Datenobjekte, die ausgetauscht werden, können mehrmals erzeugt werden. Das System gewinnt dadurch an Robustheit gegenüber asynchronen Aufrufen und Vereinfachung der Verwendung der Objekte.

5.2.2 Proxy

Das Proxy-Entwurfsmuster kapselt die Aufrufe eines Objekts, sodass eine implizite Verarbeitung ermöglicht wird. Alle Communicators (*Cache Communicator, Cloud Communicator, Peer Communicator, Application Communicator*) unterstützen RPC-Aufrufe über die dazugehörigen Schnittstellen und realisieren Proxies. Dies ermöglicht einen transparenten Zugriff auf entfernte Objekte. In diesen Klassen wird auch die zu der Kommunikation zugehörige Fehlerbehandlung übernommen. Das System gewinnt dadurch an Übersichtlichkeit im Code und einer losen Kapselung zwischen den Klassen.

5.2.3 Abstract-Factory

Das Abstract-Factory-Entwurfsmuster ermöglicht die Erzeugung der Objekte eines vorgegebenen Typs. Es besteht aus einer Abstrakten Fabrik, die konkrete Fabriken erzeugen kann. Diese Fabriken sind in der Lage konkrete Objekte eines abstrakten Types zu erzeugen. Alle Kommunikationsobjekte in Communicators werden von einer Fabrik erzeugt, bevor sie verwendet werden können. So muss eine ChannelFactory und anschließend ein Channel des korrespondierenden Typs erzeugt werden, bevor ein RPC-Aufruf erfolgen kann. Das System gewinnt dadurch an einer Monotonie im Code, sodass die Übersichtlichkeit verbessert und die Code-Wiederverwendbarkeit ausgenutzt wird.

5.2.4 Command

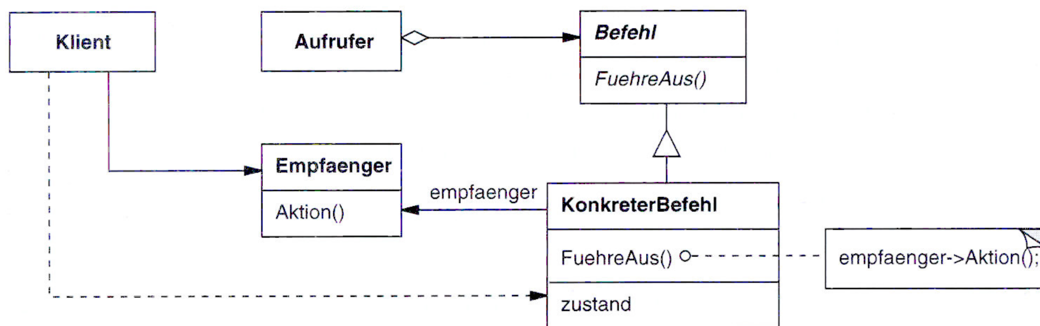

Abbildung 17: Command-Entwurfsmuster. Das Command-Entwurfsmuster spiegelt sich in der Architektur der Lösung wieder. Dabei ist jedes Command eine Operation.[GHJV10]

Das Command-Entwurfsmuster kann einen Befehl als Objekt repräsentieren, der sich ausführen lässt. Operationen basierten auf dem Command-Entwurfsmuster. Das Pattern besteht aus: Aufrufer, Befehl, konkreter Befehl, Empfänger und Klient (Abbildung 17) [GHJV10]. Der Aufrufer ist die Anwendung, die Operations verwendet. Der Befehl entspricht einer Operation des Typs *IOperation* aus einer angeschlossenen *Operations Library*. *Command Executor* erstellt einen konkreten Befehl, der einen Ausführungs-Zustand hat, somit ist er der Klient. Der Empfänger ist abhängig von der eigentlichen Funktion der Operation. In den meisten Fällen wird es ein *IS-*

torage-Objekt sein, der einen Datenzugriff ermöglicht.Das System gewinnt dadurch an Erweiterbarkeit und hoher Flexibilität im Einsatz.

5.2.5 Memento

Memento ermöglicht, einen Zustand des Objekts zu speichern und zu einem späteren Zeitpunkt wieder zu laden. Es besteht aus folgenden Teilen: Urheber, Memento, Aufbewahrer [GHJV10]. Die Serialisierungsroutinen in den Klassen, die *IStorage* implementieren, ermöglichen den Einsatz des Memento-Entwurfsmusters. So können z. B. Logs und Metadata der Datensätze flexibel im Speicher abgelegt und bei Bedarf wieder geladen werden. Den Urheber stellt *Cloud Storage Manager* und *Local Storage Manager* dar. Sie verfügen über die Serialisierungsroutinen für Objekte. Die Memento sind die Objekte, die serialisiert werden - in diesem Fall Logs und Metadata. Der Aufbewahrer ist der Speicher selbst, also *Cloud Storage* und *Local Storage*. Erst die Verwendung des Memento ermöglicht dem System das Erfüllen gewisser Anforderungen, die das Speichern der Objekte betreffen.

5.2.6 Template Method

Dieses Entwurfsmuster gibt einer abstrakten Klasse einen bestimmten Ablauf vor, der auch von den geerbten Klassen erfüllt wird. Template Method wird mit Hilfe der abstrakten Klasse *IOperation* realisiert. Es wird ein Ablauf mit Funktionen vorgegeben, der unmittelbar vor und nach der Hauptfunktion der Operation ausgeführt wird. Ein Teil dieser Funktionen wird von der abgeleiteten Klasse implementiert. Das System gewinnt dadurch an Stabilität und einem geregelten Ablauf der Ausführung von individuell programmierten Operations.

5.2.7 Mediator

Dieses Entwurfsmuster ermöglicht eine Entkopplung der Klassen, indem der Ablauf von einer zentralen Klasse gesteuert wird. Es besteht aus einem Vermittler, der den Ablauf steuert und Kollegen - den Klassen, die im Ablauf involviert sind. Der Vermittler ist der *Merge Controller* und die Kollegen sind: *Log Resolver*, *Application Resolver* und *Manual Resolver*. Der Vermittler steuert den Ablauf der

einzelnen Resolver, ohne dass sie sich gegenseitig und den Gesamtablauf kennen müssen. Das System gewinnt dadurch an Flexibilität und lässt sich mit weiteren Resolver-Algorithmen erweitern.

Nachdem die einzelnen Teile der Architektur besprochen wurden, betrachten wir die Verbindungen zwischen ihnen - die Schnittstellen .

5.3 Schnittstellen

Für Systemteile, die eine lose Kopplung voraussetzen oder eine ähnliche Aufgabe erfüllen, werden Schnittstellen verwendet. Die Schnittstellen *ICloudSync, IMerge, IExecuteOperation* sind für die Interoperabilität zwischen den drei Umgebungen essentiell. *IStorage, IPeerSync, IOperation* spielen ihre wichtigste Rolle bei der internen Verarbeitung.

5.3.1 IStorage

Die Schnittstelle *IStorage* ist eine abstrakte Klasse, die den Zugriff auf Daten sowohl in der Cloud als auch lokal auf eine vereinheitlichte Weise anbietet. Sie ist für die applikationsspezifischen Operationen von besonderer Wichtigkeit, denn so können beliebige Operationen dateisystemunabhängig auf Daten lesend und schreibend zugreifen. Entsprechend werden Datensätze in der Cloud und Lokal über gleiche Aufrufe gelesen und geschrieben, was der Übersichtlichkeit des Codes dient.

5.3.2 ICloudSync

ICloudSync ermöglicht die Kommunikation über Webservices zwischen der Cloud und dem Cache. Sie wird verwendet, um neue Datensatzversionen auf den Cache zu laden, sowie die gewünschten Änderungen an Datensätzen in die Cloud hochzuladen.

5.3.3 IPeerSync

IPeerSync ermöglicht die Peer-to-Peer-Kommunikation zwischen den Caches. Es stehen RPC-Aufrufe zur Verfügung, die die neueste, auf anderen Caches vorhandene Cloudversion zurückgeben. Des Weiteren kann auch eine von Caches geänderte

Version angefragt werden. Entsprechend sind in dieser Schnittstelle auch Antworten auf die Anfragen definiert.

Dadurch, dass über diese Schnittstelle immer nur mit der Gesamtheit der Peers kommuniziert werden kann, ist sie Unidirektional formuliert. Das heißt, dass es keine Response-Nachrichten gibt. Die Antworten werden genauso wie die Anfragen im Kommunikationsraum für alle sichtbar publiziert. Die Zuordnung der Antworten zu den Anfragen geschieht intern im *Peer Communicator*.

5.3.4 IMerge

IMerge ist eine Schnittstelle in Form einer abstrakten Klasse, die einmal in der Operations-Library der Anwendung implementiert ist und einmal in der Applikation selbst. Der Code in der Bibliothek wird von der Komponente *Application Resolver* ausgeführt, um eine anwendungsspezifische Konfliktauflösung durchzuführen. Der Code in der Applikation wird von *Manual Resolver* ausgeführt, um entsprechenden Dialog dem Benutzer anzuzeigen, damit er die Konfliktauflösung vornehmen kann.

5.3.5 IOperation

IOperation ist eine abstrakte Klasse, die gewisse Abläufe und Funktionsaufrufe enthält, damit alle Operationen nach dem gleichen Grundschema ausgeführt werden können. Die Abläufe enthalten wichtige Prüfungen und Fehlerbehandlungen, sodass ein korrekter Operationsaufruf erfolgen kann.

5.3.6 IExecuteOperation

IExecuteOperation ist eine Schnittstelle für ein Webservice, um der Anwendung einen Operationsaufruf zu ermöglichen und gleichzeitig eine lose Kopplung zwischen dem Cache und der Anwendung sicherzustellen.

An dieser Stelle sind alle Komponenten und ihre Verbindungen besprochen. Im Folgenden werden Vorgänge beschrieben, die im System stattfinden.

5.4 Abläufe

In diesem Abschnitt werden Abläufe beschrieben, um die wichtigsten Funktionen der Komponenten in Verbindung zu bringen. Es wird angenommen, dass die anwendungsspezifischen Operationen als DLLs im bestimmten Ordner des Caches abgelegt sind. Des Weiteren wird vorausgesetzt, dass alle drei Teile: Cloud, Cache und die Anwendung gestartet sind und die Verbindung zueinander hergestellt wurde. Wir beginnen mit dem Ablauf einer Operationsausführung und gehen anschließend über Synchronisationsabläufe zu der Konfliktauflösung.

5.4.1 Operationsaufruf

Die Applikation ruft über ein Webservice mit der Schnittstelle *IExecuteOperation* die gewünschte Operation auf. Der *Application Communicator* empfängt den Aufruf und übergibt die Informationen über die auszuführende Operation an den *Command Executor*. Dieser prüft die Existenz der auszuführenden Operation und erstellt eine Operation-Instanz des Typs *IOperation*. Dieser Instanz werden beim Erstellen folgende Informationen übermittelt: Operationsname, Operationsparameter und eine Instanz des *Local Storage Managers* als *IStorage*. Nachdem die Operation mit diesen Informationen instanziiert wurde, wird sie gemäß folgendem Ablauf ausgeführt:

1. Vorbedingungen für die Ausführung prüfen
2. Intention speichern
3. Operation ausführen: Daten und Metadaten Lesen und Schreiben
4. Intention an Log anhängen, Erfolg vermerken
5. Rückgabewert zurückgeben

Der Rückgabewert, der ein beliebiges Objekt beinhalten kann, wird an *Command Executor* zurückgegeben. Dieser reicht diesen Wert an *Application Communicator* weiter. Er übermittelt den Wert über Webservices als eine Antwort auf die Anfrage an die Applikation.

Im Hintergrund wird die Instanz der Operation durch den Garbage Collector aufgeräumt, weil alle Referenzen darauf gelöscht wurden.

Nach einigen Operationsausführungen können Änderungen entsprechend der Synchronisierungsroutine ausgetauscht, die in nächsten Abschnitten beleuchtet wird.

5.4.2 Synchronisation mit der Cloud

Sync Logic des Caches initiiert einen Synchronisierungsvorgang mit der Cloud. Der Auslöser kann ein durch Timer getriggertes Ereignis oder ein explizites Kommando sein. Der *Cloud Communicator* sendet eine Anfrage über die Schnittstelle *ICloud-Sync* an den *Cache Communicator* in der Cloud, die Aktualität bestimmter Datensätze zu prüfen und die Änderungen zu übermitteln. Der *Version Selector* der Cloud gibt die gewünschten Datensätze zurück. Dabei greift er über die Schnittstelle *IStorage* auf den Cloud-Speicher *Cloud Storage* zu. Die Antwort des *Cache Communicators* wird vom *Cloud Communicator* an *Sync Logic* weitergereicht. Dieser initiiert bei einem Konflikt den Mergevorgang in der Komponente *Merge Logic*.

Anschließend werden die lokal geänderten Daten in die Cloud als Änderungsanfragen hochgeladen. Das geschieht auf dem gleichen Weg über die Communicators. Die neuen Zustände der Datensätze werden an *Version Collector* übergeben, der diese abspeichert und prüft, ob eine neue Datensatzversion über *Version Selector* generiert werden kann. Sind bestimmte Vorbedingungen erfüllt, dann wird die neue Datensatzversion erstellt und in der Cloud abgespeichert. Beim nächsten Synchronisierungsvorgang wird die neueste Datensatzversion an alle Caches übermittelt, die ihre lokale Version aktualisieren möchten und entsprechende Synchronisations-Anfragen stellen.

Im Falle einer Netzwerkpartitionierung wird die Peer-to-Peer-Verbindung verwendet, um die Datensätze zu synchronisieren. Der entsprechende Ablauf ist wie folgt.

5.4.3 Synchronisation über Peer-to-Peer

Auch im Peer-to-Peer-Modus initiiert *Sync Logic* den Synchronisierungsvorgang. Über *Peer Communicator* und die Schnittstelle *IPeerSync* wird bei erreichbaren Peers eine Datensatzversion angefordert, die aktueller ist als die lokale Version. Dabei antworten Peers, die die gewünschte Version lokal vorhanden haben. Anschließend wählt der Anfragende einen konkreten Peer aus und lädt seine Version

herunter. Als nächstes werden die beiden Versionen mit Hilfe der Mergekomponente zusammengeführt.

Der genaue Ablauf, wie die Versionen zusammengeführt werden, wird im nächsten Abschnitt beschrieben.

5.4.4 Konfliktauflösung

Der Mergevorgang wird von *Sync Logic* im *Merge Controller* initiiert. Dieser versucht den Konflikt entsprechend dem Ablauf aus Abschnitt 4.3.4 aufzulösen. Zunächst kommt *Log Resolver* zum Einsatz. Bei Misserfolg versucht *Application Resolver* den Konflikt über die Schnittstelle *IMerge* und die entsprechende, in einer Operations-DLL hinterlegte anwendungsspezifische Funktion aufzulösen. Schlägt auch dieser Versuch fehl, dann kommt der *Manual Resolver* zum Einsatz. Die konfliktbehafteten Datensatzversionen werden der Anwendung über *Application Communicator* und die Schnittstelle *IMerge* übergeben. Die Anwendung löst mit Hilfe des Benutzers den Konflikt auf und gibt auf dem gleichen Pfad das Ergebnis des Mergens zurück.

Abschließend wird das Resultat über die *IStorage*-Schnittstelle lokal abgespeichert.

Die in diesem Kapitel beschriebenen Komponenten, Schnittstellen und Abläufe sind nicht nur theoretisch - sie wurden auch praktisch umgesetzt. Hierfür existiert eine Beispielanwendung, die die Funktionalitäten veranschaulicht und im nächsten Abschnitt präsentiert wird.

5.5 Beispielanwendung

Entsprechend dem Entwurf aus Kapitel 4 und den Überlegungen aus diesem Kapitel wurde ein konkretes System mit grundlegenden Funktionen implementiert. Des Weiteren wurde die im Beispielszenario (Abschnitt 3.2.1) beschriebene Endanwendung in Grundzügen ebenfalls programmiert. Damit wir die zwei Implementierungen unterscheiden können, nennen wir das System, das Operations realisiert „OpSync“ und die Beispielanwendung „RecipeApp“.

Da es sich um ein Proof of Concept handelt, wurden nur die grundlegenden Funk-

tionen implementiert. Dabei wurde die Machbarkeit der Interaktion zwischen dem Cache und der Endanwendung (Operationsaufrufe), Synchronisation zwischen den Caches über Peer-to-Peer, sowie zwischen dem Cache und der Cloud geprüft. Außerdem wurde das Mergen zweier Datensatzversionen auf der Log-Ebene getestet und schlussendlich die Verwendung der Datensätze von der Endanwendung.

Wir beginnen mit dem Aufbau des Proof of Concept, dann gehen wir zu den einzelnen Anwendungsfällen über und schlussendlich werden Hintergrundinformationen gegeben.

5.5.1 Aufbau

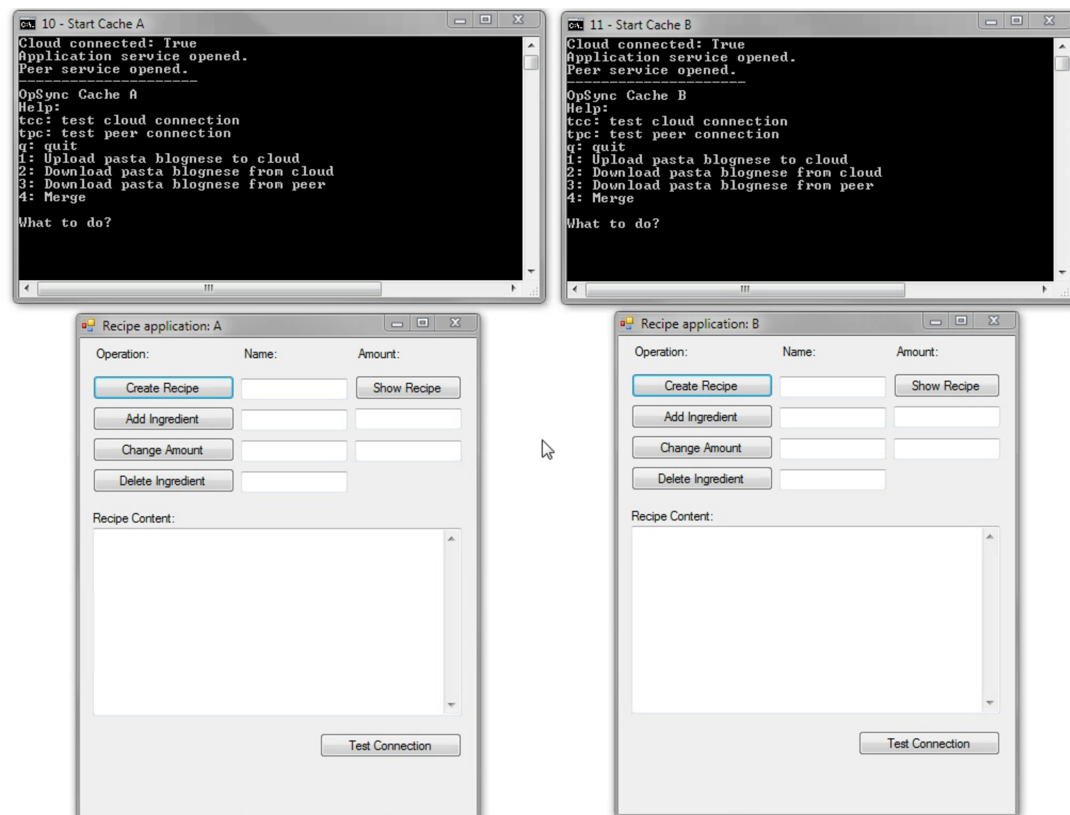

Abbildung 18: Bildschirmfoto der Endanwendungen und Demonstrationskonsolen. Die einzelnen Endanwendungen sind an jeweils einen Cache angeschlossen, der durch eine Demonstrationskonsole präsentiert wird.

In Abbildung 18 sind zwei Konsolenfenstern und zwei Formularfenster zu sehen. Die

Konsolenfenster stehen zu Demonstrationszwecken bereit; in einer Produktivumgebung arbeitet OpSync vollautomatisch und unsichtbar. Über die Konsolenfenster lassen sich einzelne Anwendungsfälle Schritt-für-Schritt ausführen, die im Folgenden konkret beschrieben werden. Die Formularfenster visualisieren die Endanwendung, die Daten gebraucht. Damit lassen sich Rezepte erstellen und anzeigen, sowie Zutaten hinzufügen, modifizieren und entfernen.

5.5.2 Anwendungsfälle

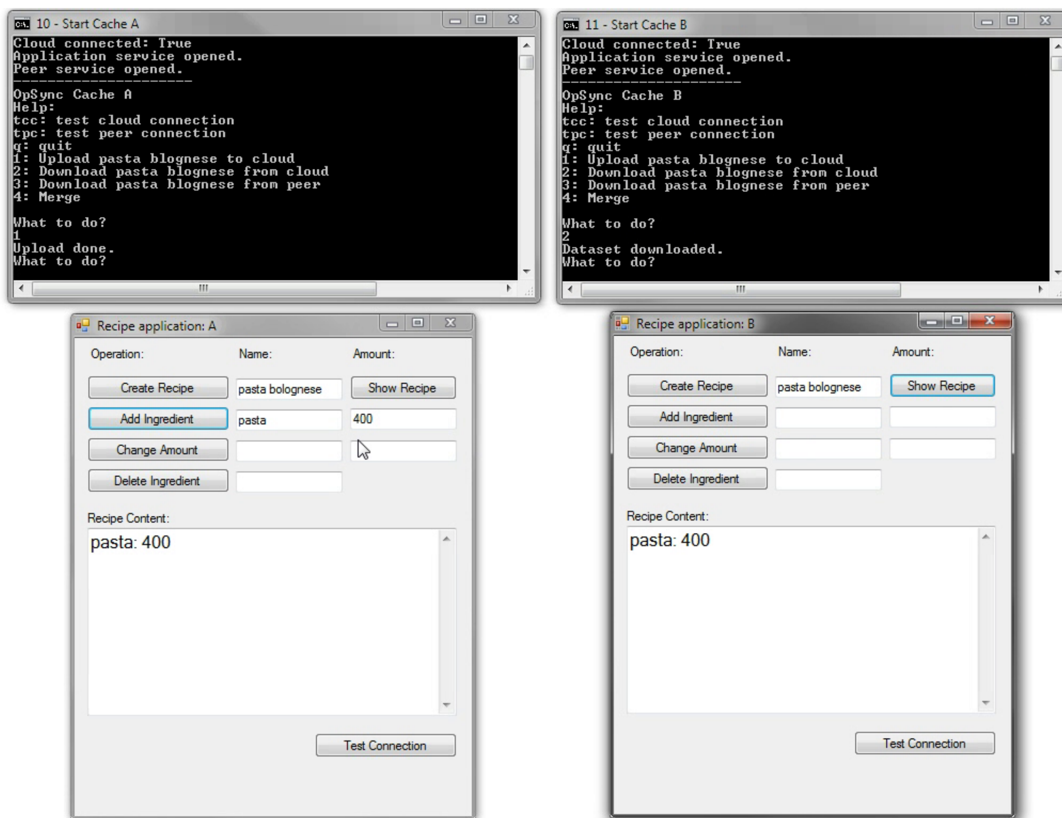

Abbildung 19: Demonstration der Implementierung - Anwendungsfall 1. Synchronisierung eines Rezeptes über die Cloud.

Zunächst wird mit RecipeApp A ein Rezept erstellt und von OpSync A mit der Cloud synchronisiert. Anschließend synchronisiert OpSync B mit der Cloud und die Anwendung RecipeApp B zeigt das eben erstellte Rezept an (Abbildung 19).

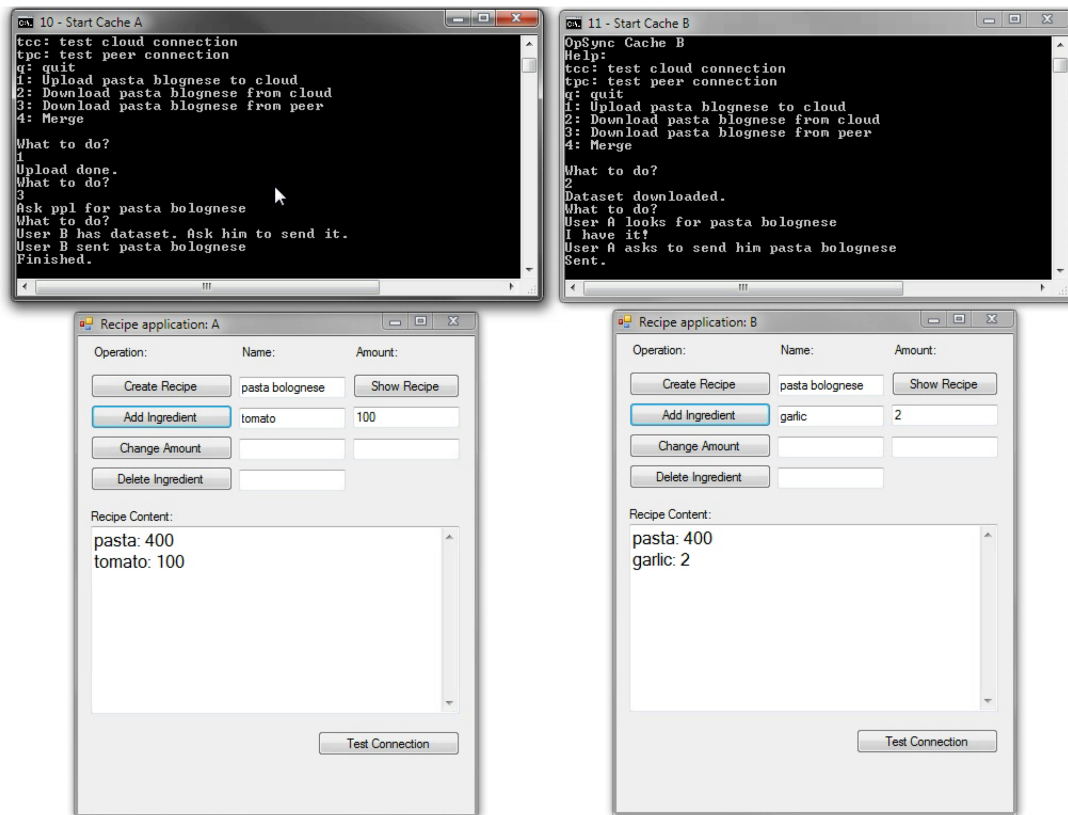

Abbildung 20: Demonstration der Implementierung - Anwendungsfall 2. Synchronisierung eines Rezeptes über Peer-to-Peer. Der Datensatz wird übertragen, aber noch nicht gemereged.

Im nächsten Anwendungsfall wird das Rezept von beiden Endanwendungen modifiziert und über die Peer-to-Peer-Kommunikation synchronisiert, um die Funktionsfähigkeit bei einem Cloud-Ausfall zu demonstrieren. Den Dialog zwischen den OpSync-Instanzen lässt sich in Abbildung 20 nachvollziehen. OpSync A fragt alle erreichbaren Peers nach einer Version des Rezeptes Pasta Bolognese, OpSync B gibt Bescheid, dass er Pasta Bolognese hat. Anschließend wird der Datensatz übertragen.

Der nächste Anwendungsfall zeigt, dass OpSync A die zwei Datensatzversionen zusammengeführt hat und nach dem Wiederherstellen der Verbindung zur Cloud wieder hochgeladen. RecipeApp A stellt dabei die Daten von dem vorherigen Anwendungsfall dar, weil sie von der Endanwendung nicht neugeladen wurden (Abbildung 21).

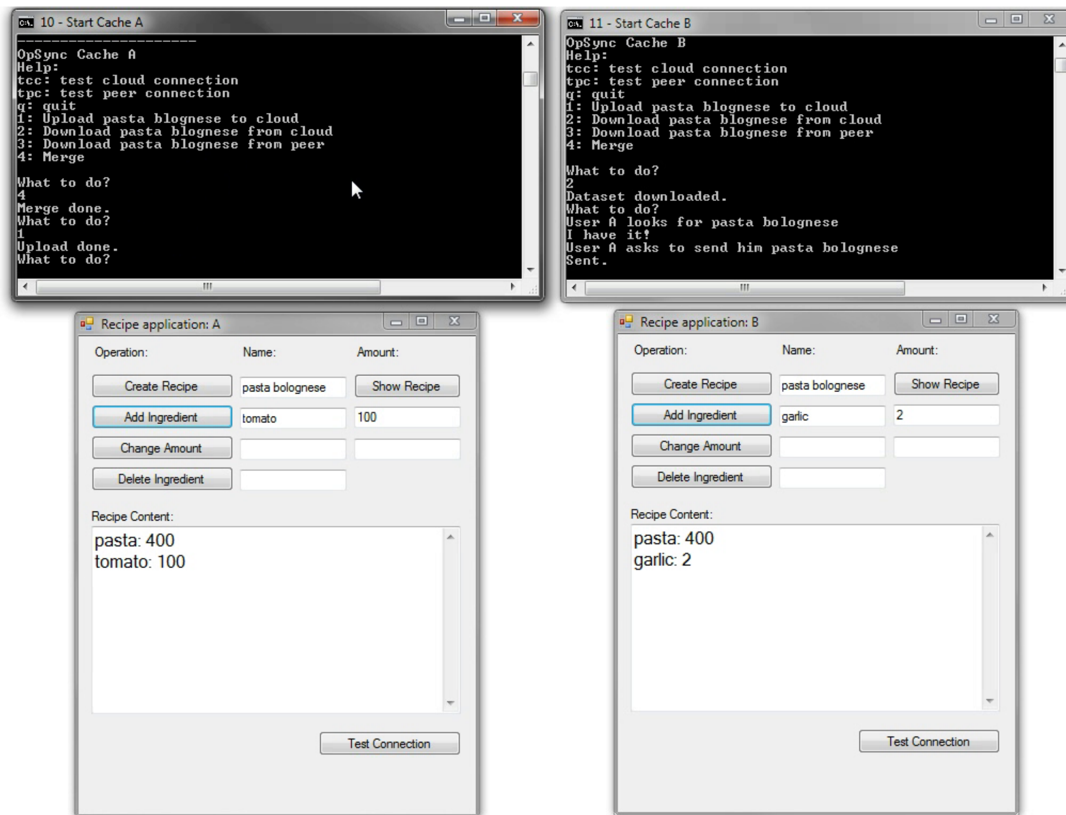

Abbildung 21: Demonstration der Implementierung - Anwendungsfall 3. Mergen von zwei Datensatzversionen und Hochladen in die Cloud.

Der letzte Anwendungsfall der Demonstration zeigt OpSync A mit geleertem Cache, der einen neuen Teilnehmer im System darstellt. Pasta Bolognese wird aus der Cloud heruntergeladen und mit einem Klick auf „Show Recipe“ werden Auswirkungen des Bearbeitens, der Synchronisation über Peer-to-Peer, des Mergens und der Synchronisation über die Cloud aus den letzten Anwendungsfällen sichtbar (Abbildung 22).

5.5.3 Hintergrundinformationen

Der Datensatz besteht aus einer binär serialisierten Tabelle. In der ersten Spalte werden Zutaten eingetragen, in der zweiten wird die Menge als numerischer Wert gespeichert.

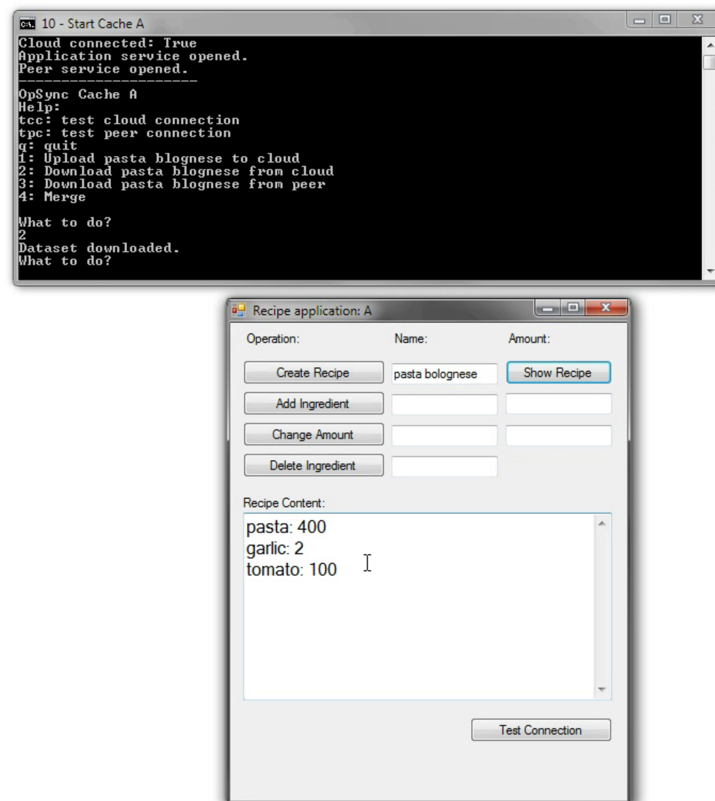

Abbildung 22: Demonstration der Implementierung - Anwendungsfall 4. Ein neuer Teilnehmer tritt dem System bei und synchronisiert ein Rezept.

– CreateRecipe(„pasta bolognese“)
– AddIngredient(„pasta“, 400)
– AddIngredient(„garlic“, 2)
– AddIngredient(„tomato“, 100)

Abbildung 23: Zusammengeführter Log nach simultaner Bearbeitung eines Rezeptes. Die grün hervorgehobene Zeile zeigt die aus dem anderen Log hinzugefügte Operation.

Das Mergen wurde nicht auf Datenzustandsebene durchgeführt, indem die von einer Version abweichende Tabellenzeile hinzugefügt wurde, sondern auf der Logebene, indem die abweichende Operation aus dem anderen Log eingefügt (Abbildung 23) und der neue Log anschließend ausgeführt wurde, um einen konfliktfreien Zustand

des Datensatzes zu erhalten.

5.6 Zusammenfassung

In diesem Kapitel wurde eine genaue Architektur des Systems vorgestellt. Dabei wurde das System aus der Mikro-, sowie Makroperspektive betrachtet. Die Lösung besteht aus einzelnen Komponenten und Klassen, die mit der Verwendung der Entwurfsmuster strukturiert sind. Die beschriebenen Schnittstellen zwischen den Komponenten bilden Kommunikationswege, über die verschiedene Abläufe stattfinden können. Das implementierte System wird anhand einer Beispiel-Endanwendung demonstriert: die Abläufe des Systems ermöglichen eine Datenverarbeitung- und Verwaltung, wie es am Anfang des Dokumentes beabsichtigt war.

Als Nächstes folgt ein Kapitel, indem das System evaluiert wird. Dabei liegt der Schwerpunkt auf den finanziellen Aspekten, die das System optimiert.

6 Evaluierung

In diesem Kapitel soll die Synchronisierungsroutine der Lösung hinsichtlich der Betriebskosten evaluiert werden. Es wird nur die Synchronisierung mit der Cloud betrachtet, denn es ist entsprechend der Aufgabenstellung dieser Arbeit der Schwerpunkt. Die Synchronisation über Peer-to-Peer wird nur am Rande erwähnt.

Zunächst wird die Evaluationsumgebung beschrieben, um die Umstände zu definieren, unter welchen die Evaluierung stattfindet. Des Weiteren werden Parameter erläutert, die das System beeinflussen. Anschließend wird ein konkretes Kostenmodell vorgestellt, das den finanziellen Aufwand für den Betrieb des Systems berechnet und die wichtigsten Kostenfaktoren herauskristallisiert. Danach werden Szenarien für verschiedene Konfigurationen dieser Kostenfaktoren vorgestellt und unter dem Gesichtspunkt der Optimierung bewertet. Abschließend wird ein Fazit über die Kosteneffizienz des Systems gezogen.

6.1 Setup

In diesem Unterkapitel wird die Grundlage für die Evaluation gelegt. Es wird ein Aufbau der Systems und die Randbedingungen beschrieben. Wir beginnen mit der Evaluationsumgebung, danach folgen die für die Evaluation wichtigen Parameter. Anschließend werden die finanziellen Aspekte diskutiert.

6.1.1 Evaluationsumgebung

Bei der Evaluation wird mit der Synchronisierung von 1, 10, 100 und 1000 Caches gerechnet, um verschiedene Szenarien abzudecken. Die Caches sind entsprechend dem Systemmodell (Kapitel 3) geografisch verteilt und an das Internet angeschlossen. Die Caches synchronisieren ihre Kopien mit der Cloud. Es wird nur der Fall evaluiert, wenn die Cloud verfügbar ist. Im Ausnahmefall wenn sie nicht verfügbar ist, geschieht die Synchronisation über Peer-to-Peer, was nicht zu den Schwerpunkten der Arbeit gehört und somit nicht evaluiert wird.

In der Windows Azure-Cloud wird ein Rechner mit 1 GHz CPU, 768 MB RAM und 20 GB Festplattenspeicher eingesetzt. Auf diesem Rechner wird folgende Software

ausgeführt: Windows Server 2008 R2 mit .NET 4.5 und IIS 7, sowie dem Cloud-Teil der Implementierung.

Es wird davon ausgegangen, dass Caches auf Rechnern ausgeführt werden, die mindestens so leistungsfähig sind, wie der Entwicklungsrechner, auf dem die Implementierung und Tests (Kapitel 5) vorgenommen wurden. Dabei ist die Hardwarekonfiguration wie folgt: Prozessor: 2x2,5 GHz, 3 GB RAM und 320 GB Festplatte. Betrieben wird die Hardware mit Microsoft Windows 7 und .NET 4.5.

Der Einsatz von schwächeren Rechnern kann die Leistung des Caches reduzieren, dies wird jedoch in den nachfolgenden Rechnungen nicht weiter berücksichtigt. Die Leistung des Systems kann auch in Verbindung mit der Systemkonfiguration evaluiert werden, was jedoch nicht im Fokus dieser Arbeit liegt.

6.1.2 Evaluationsparameter

Es wird der Worst Case angenommen, dass Datensätze aus zufällig generierten Daten bestehen und bei einer Änderung durch neue Daten vollständig ersetzt werden. Entsprechend lassen sich keine Transferkosten durch Rsync (Abschnitt 2.2.1) reduzieren. Des Weiteren wird in dieser Evaluation mit Brutto-Datenmengen gerechnet. Das heißt, wenn 100 MB an Datensätzen synchronisiert werden, dann ist es die Summe aus Nutzdaten, den Logs, sowie den zugehörigen Metadaten. Das Netto/Brutto-Verhältnis variiert je nach Anwendungsfall sehr stark. Entsprechend ist die Evaluation nur dann repräsentativ, wenn die Brutto-Datenmengen verwendet werden. Das Verhältnis der individuellen Anwendungsfälle kann mit dem Ergebnis der Evaluation verrechnet werden, um die Netto-Angaben zu erhalten. Aufgrund dieser Annahmen wird die reine Synchronisations-Routine evaluiert.

Es wird angenommen, dass es sich um ein System handelt, bei dem regelmäßig gelesen und geschrieben wird, was für Business-Anwendungen üblich ist.

Die variierten Werte sind:

- Anzahl der Caches: 1 - 1000
- Übertragene Datenmenge pro Synchronisierung: 1-1024 MB
- Synchronisierungs-/Lesehäufigkeit: täglich bis alle fünf Minuten

- Kosten für das Lesen eines veralteten Datensatzes: 1-20 €
  (z. B. durch das Verkaufen einer Ware, für die der Preis erhöht, aber der alte Preis gelesen und verbucht wurde)

Ausgewertet wird der finanzielle Aufwand für das Betreiben des Systems in Euro, abhängig von den genannten Parametern. Wie genau die Kosten berechnet werden, wird im nächsten Unterkapitel beschrieben.

6.1.3 Finanzielle Kosten

Es werden die Kosten für den Einsatz des Systems evaluiert. Entsprechend müssen die monatlichen Kosten für den Betrieb der Cloud-Infrastruktur berücksichtigt werden. Um konkrete Kosten berechnen zu können, wird ein existierender Cloud-Anbieter hinzugezogen. Es kann ein beliebiger Anbieter sein; in dieser Arbeit wird, aufgrund seiner Einfachheit und weiten Verbreitung, das System Windows Azure von Microsoft betrachtet. Im Folgenden wird das Preismodell des Anbieters beschrieben und erklärt.

Die Kosten für den Peer-to-Peer Betrieb werden durch andere Faktoren beeinflusst, als die Cloud-Kosten. Dort spielt die Bandbreite eine große Rolle, denn durch auf Flooding basierende Verfahren entsteht eine große Menge an Nachrichten, die verschickt werden. Außerdem ist je nach Anwendungsfall möglich, dass laufende Kosten für die Übertragung der einzelnen Nachrichten anfallen, oder dass die Übertragungsgeschwindigkeit nach dem Erreichen eines Limits erheblich reduziert wird. Die Kosten für die Cloud werden anhand eines konkreten Kostenmodells berechnet, in diesem Fall anhand der Windows Azure-Kostenverteilung.

Windows Azure

Für die Kostenberechnung wird ein Zeitraum von einem Monat betrachtet. Die Fixkosten für den Betrieb von der entwickelten Lösung in der Microsoft Azure Cloud berechnen sich wie folgt.

Fixkosten: Es fallen $6,64$ € für die Bereitstellung der Rechenkapazitäten in Form eines Rechners und $6,6$ € für einen 100 GB-großen Blob-Speicher an. Die Summe ergibt $13,24$ €.

Laufende Kosten: Die laufenden Kosten entstehen durch die Speichertransaktionen

in der Cloud, den Datenverkehr und den über Service Bus versandten Nachrichten. Jede Speicher- oder Lade-Transaktion kostet $8.0 \cdot 10^{-8}$ €. Jedes übertragene GB kostet $0,0809$ €. Jede Nachricht, die über den Service Bus versandt wird, kostet $7,1 \cdot 10^{-7}$ €.

Eine Synchronisierungsoperation besteht aus dem Herunterladen einer aktualisierten Version, oder dem Hochladen einer neuen Version. Dabei wird eine Speicher- oder Lade-Transaktion benötigt, und müssen zwei Nachrichten versandt werden - eine Anfrage und eine Antwort. Dabei ist die durchschnittliche Datensatzgröße je nach Anwendungsszenario unterschiedlich und wird hier durch die Variable d (in GB) repräsentiert. Die Variable h stellt die Anzahl der Synchronisierungsoperationen pro Monat dar und n gibt die Anzahl der synchronisierenden Caches an.

Die monatlichen Gesamtkosten werden durch folgende Formel berechnet: $f(h,d,n) = \underbrace{6,64 + 6,6}_{Fixkosten} + h \cdot (\underbrace{8.0 \cdot 10^{-8} \cdot h}_{Speichertransaktions-Kosten} + \underbrace{0,0809 \cdot h \cdot d}_{Transferkosten} + \underbrace{7,1 \cdot 10^{-7} \cdot 2 \cdot h}_{ServiceBus-Kosten})$

Somit sind die ausschlaggebenden Parameter, die die Kosten beeinflussen: die zu synchronisierende Datenmenge, Synchronisierungshäufigkeit und die Anzahl der Caches.

Anhand dieser Formel lassen sich Kostenverteilungen für verschiedene Szenarien erstellen. Die Szenarien werden nachfolgend beschrieben.

6.2 Szenarien

In den Evaluations-Szenarien betrachten wir zunächst, entsprechend den zwei Evaluationsparametern Synchronisierungshäufigkeit und Datenmenge, vier realistische Extremfälle, die sich kombinatorisch ergeben. Die Extremfälle sind in Form eines Quadrats in der Abbildung 24 dargestellt und mit Bezeichnungen, wie z. B. „0H 1D“ abgekürzt. Dabei steht „H“ für die Synchronisierungs**h**äufigkeit und „D“ für die **D**atenmenge. Die führende Ziffer gibt die Größe des jeweiligen Parameters an. 0 steht für das Minimum und 1 für das Maximum. Die Grenzen werden anhand des für die Evaluierung relevanten Beispiels wie folgt festgelegt.

Ein Unternehmen synchronisiert Stammdaten[4] alle 24 Stunden, also in einem Monat

[4]Stammdaten: „In der betrieblichen Datenverarbeitung wichtige Grunddaten (Daten) ei-

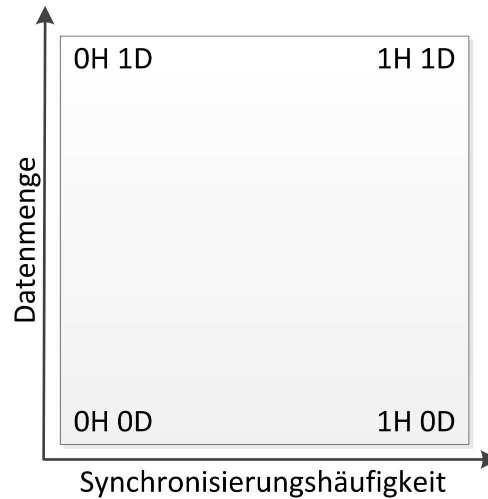

Abbildung 24: Darstellung der Evaluationskriterien. Aus den Kriterien Synchronisierungshäufigkeit und Datenmenge ergeben sich vier Szenarien.

30 Mal[5] und damit sehr selten, d. h. 0H = 30 Mal/Monat.

Die Bewegungsdaten[6] des Unternehmens werden alle 15 Minuten synchronisiert und damit sehr häufig, d. h. $1H = 30 \cdot 24 \cdot 4 = 2880$ Mal/Monat.

Die Stammdaten ändern sich täglich um 100 MB. Nur die Änderung wird beim erneuten Synchronisationsvorgang übertragen. Die Bewegungsdaten ändern sich alle 15 Minuten um 1 MB. Daraus ergeben sich folgende Grenzwerte: 0D = 1 MB und

nes Betriebs, die über einen gewissen Zeitraum nicht verändert werden; z.B. Artikel-Stammdaten, Kunden-Stammdaten, Lieferanten-Stammdaten, Erzeugnisstrukturen (Stücklisten) u.a. Stammdaten werden oft nicht permanent, sondern periodisch aktualisiert (Dateifortschreibung).“ [Spr]

[5] Um die Rechnung zu vereinfachen wird der Durchschnittswert der Länge aller Monate verwendet: $\frac{365}{12} \approx 30$.

[6] Bewegungsdaten: „In der betrieblichen Datenverarbeitung Daten, die Veränderungen von Zuständen beschreiben und dazu herangezogen werden, Stammdaten zu aktualisieren.“ [Spr]

1D = 100 MB.

Mit diesen Grenzwerten lassen sich die folgenden realistischen Szenarien erstellen:

- Ein Unternehmen synchronisiert selten große Stammdaten (0H 1D) (z. B. Artikel- oder Kundenstamm eines Einzelhändlers)
- Ein Unternehmen synchronisiert oft kleine Bewegungsdaten (1H 0D) (z. B. Preisänderungen und Änderungen der Kundeninformationen)
- Ein Privatanwender synchronisiert selten geringe Datenmengen (0H 0D) (z. B. Abgleich seines Kalenders auf verschiedenen Geräten)
- Ein Wetter-Forschungsinstitut synchronisiert oft große Datenmengen (1H 1D) (z. B. Sensor- und Berechnungsdaten)

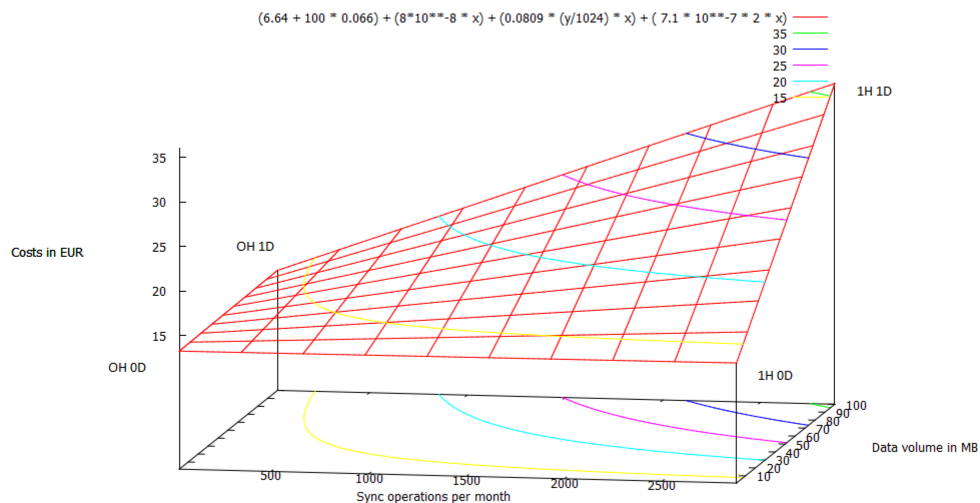

Abbildung 25: Kosten für die Synchronisation eines Caches ohne Optimierung. Die Kosten bei drei Szenarien liegen knapp über den Fixkosten von 13,24 €.

Ohne jegliche Optimierung würden im Rahmen der einzelnen Szenarien folgende Kosten entstehen (Abbildung 25).

- 0H 0D: $f(30, 1MB, 1) \approx 13,24$ €
- 0H 1D: $f(30, 100MB, 1) \approx 13,48$ €
- 1H 0D: $f(2880, 1MB, 1) \approx 13,47$ €

- 1H 1D: $f(2880, 100MB, 1) \approx 36,00$ €

Aufgrund einer geringen Steigerung der Kosten in den Szenarien 0H 0D, 0H 1D und 1H 0D von maximal 0,24 € gegenüber den Fixkosten, werden sie nach der Optimierung nur für das Szenario 1H 1D ausgewertet, bei dem der Unterschied bei 22,76 € liegt. Die Auswertung folgt im nächsten Abschnitt, in dem auch der Grund für die Kostenexplosion untersucht wird. Dementsprechend werden Kosten berechnet, wenn das System die Synchronisation optimiert.

6.3 Auswertung

In diesem Abschnitt wird das Szenario 1H 1D unter die Lupe genommen. Für diesen Extremfall werden Kosten und die dazugehörige Kostensenkung berechnet. Anschließend werden die einzelnen Parameter des Szenarios variiert. Wenn diese Parameter Extremwerte annehmen, lässt sich ein Trend für die Kostenverteilung beobachten, der anschließend bewertet wird.

6.3.1 Kosten im Szenario 1H 1D

Zunächst müssen wir die einzelnen Summanden der additiven Kostenfunktion untersuchen, um die ausschlaggebenden Parameter zu bestimmen, die am meisten Kosten verursachen. Die Funktion wurde bereits in Abschnitt 6.1.3 präsentiert und hier wird sie zwecks Übersichtlichkeit wiederholt.

$$f(h,d,n) = \underbrace{6,64 + 6,6}_{Fixkosten} + h \cdot (\underbrace{8.0 \cdot 10^{-8} \cdot h}_{Speichertransaktions-Kosten} + \underbrace{0,0809 \cdot h \cdot d}_{Transferkosten} + \underbrace{7,1 \cdot 10^{-7} \cdot 2 \cdot h}_{ServiceBus-Kosten})$$

Man erkennt, dass die Speicherzugriffskosten **und** die Service Bus-Kosten selbst bei hoher Synchronisationsrate x aufgrund des vorangestellten Faktors von 10^{-8} bzw. 10^{-7} keine signifikante Kostensteigerung verursachen:

- Speicherzugriffs-Kosten: $8.0 \cdot 10^{-8} \cdot 2880 \approx 0.00$ €
- Service Bus-kosten: $7,1 \cdot 10^{-7} \cdot 2 \cdot 2880 \approx 0.00$ €

Das Ausschlaggebende sind die Datenübertragungskosten pro synchronisierten Cache: $0,0809 \cdot 2880 \cdot \frac{100}{1024} \approx 22,75$ €. Bei zehn Caches betragen diese Kosten $227,5$ €, was ein beträchtlicher Wert ist, der mit jedem weiteren Cache linear wächst. Es

wird nun untersucht, in wieweit die Transfer-Kosten durch den Einsatz der in dieser Arbeit entwickelten Lösung gesenkt werden.

6.3.2 Kostensenkung durch die Optimierungsfunktion

In Abschnitt 4.2.3 wurden Funktionen $x_{opt_get_limited}$, $x_{opt_put_limited}$, p_{get}, p_{put}, $eval_{get}$ und $eval_{put}$ definiert, die von einigen Konstanten abhängig sind. Diese werden in der nachfolgenden Tabelle definiert, dabei wird weiterhin das Szenario 1H 1D mit entsprechenden Werten betrachtet.

Variable	**Wert(e)**	**Beschreibung**
$a_{writes} = a_{reads}$	2880	Pro Zeitabschnitt wird 1H oft gelesen und geschrieben
a_{caches}	1; 10; 100; 1000	Variable Anzahl an Caches, die evaluiert werden
c_{read_old}	5 €	Jedes Lesen eines veralteten Wertes kostet 5€
d_{data_up}	$\frac{100}{1024}$	1D werden geändert und hochgeladen, in GB
$c_{dataset}$	0,0809	Kosten in Windows Azure für die einmalige Übertragung von 1 GB
d_{data_down}	$\frac{100}{1024}$	1D Aktualisierungen werden pro Cache heruntergeladen, in GB
c_{fix}	13,24 €	Entspricht den Fixkosten im Azure-Kostenmodell

Für diesen Anwendungsfall ergeben die optimalen Synchronisierungsraten für alle Größenordnungen der Cache-Anzahl folgende Werte:
$x_{opt_put_limited} = 2880$ (Häufigkeit für das Hochladen der Updates; entspricht a_{writes} und somit der naiven Synchronisationsrate 1H)
$x_{opt_get_limited} = 1350$ (Häufigkeit für das Herunterladen der Updates)

Die monatlichen Kosten, berechnet mit der generischen Optimierungsformel mit **naiver** Synchronisationsrate von 1H (Abbildung 26):

Formel	a_{caches}=**1**	a_{caches}=**10**	a_{caches}=**100**	a_{caches}=**1000**
$eval_{get}(2880)$	35,99 €	240,77 €	2288,55 €	22766,36 €
$eval_{put}(2880)$	35,99 €	240,77 €	2288,55 €	22766,36 €

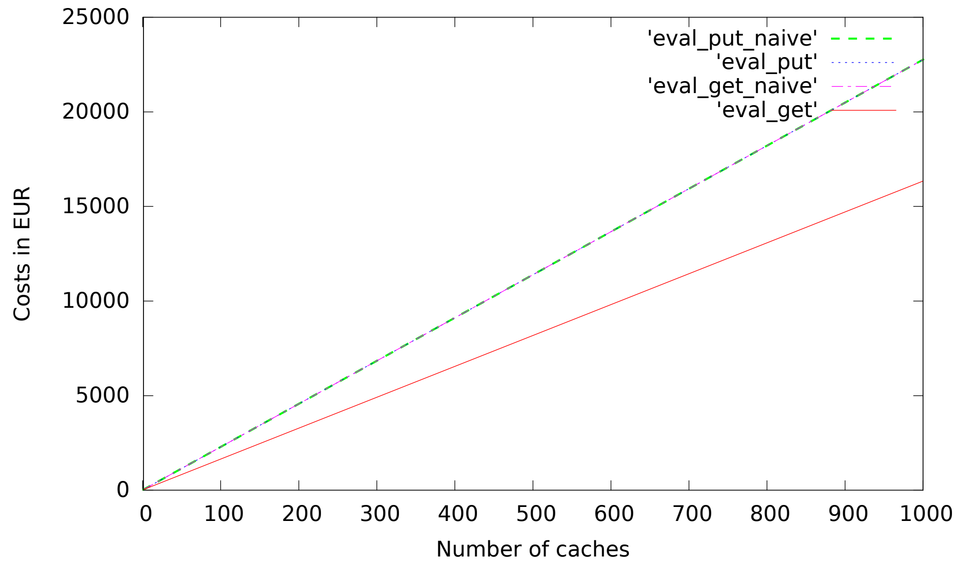

Abbildung 26: Kostenverteilung der generischen Kosten-Funktion. Die Kurven eval_put_naive, eval_put und eval_get_naive liegen übereinander.

Die monatlichen Kosten, berechnet mit der generischen Optimierungsformel und **optimierten** Synchronisationsraten für unterschiedliche Anzahl der Caches sind wie folgt (Abbildung 26):

Formel	a_{caches}=**1**	a_{caches}=**10**	a_{caches}=**100**	a_{caches}=**1000**
$eval_{get}(1350)$	29,57 €	176,56 €	1646,46 €	16345,43 €
$eval_{put}(2880)$	35,99 €	240,77 €	2288,55 €	22766,36 €

Bevor die Azure-Kosten berechnet werden können, muss die Rechnung durch die Penalty-Funktion ergänzt werden, die die Kosten für das Lesen der veralteten Werte berechnet und somit bei optimierten Synchronisierungsraten berücksichtigt werden muss. Auf diese Weise werden die Optimierungsformel und die Azure-Kostenformel miteinander vergleichbar, denn beide berechnen die vollen Betriebskosten.

Die monatlichen Windows Azure-Kosten im Fall 1H 1D **ohne Optimierung** (Abbildung 27). Dabei ist p(2880) = 0, weil die Synchronisierungsrate im naiven Fall nicht herabgesenkt (und somit optimiert) wird:

Formel	**G/P**	a_{caches}=**1**	a_{caches}=**10**
$p_{get}(2880) + f(2880, 100MB, a_{caches})$	Get	36 €	240,81 €
$p_{put}(2880) + f(2880, 100MB, a_{caches})$	Put	36 €	240,81 €

Formel	**G/P**	a_{caches}=**100**	a_{caches}=**1000**
$p_{get}(2880) + f(2880, 100MB, a_{caches})$	Get	2288,98 €	22770,68 €
$p_{put}(2880) + f(2880, 100MB, a_{caches})$	Put	2288,98 €	22770,68 €

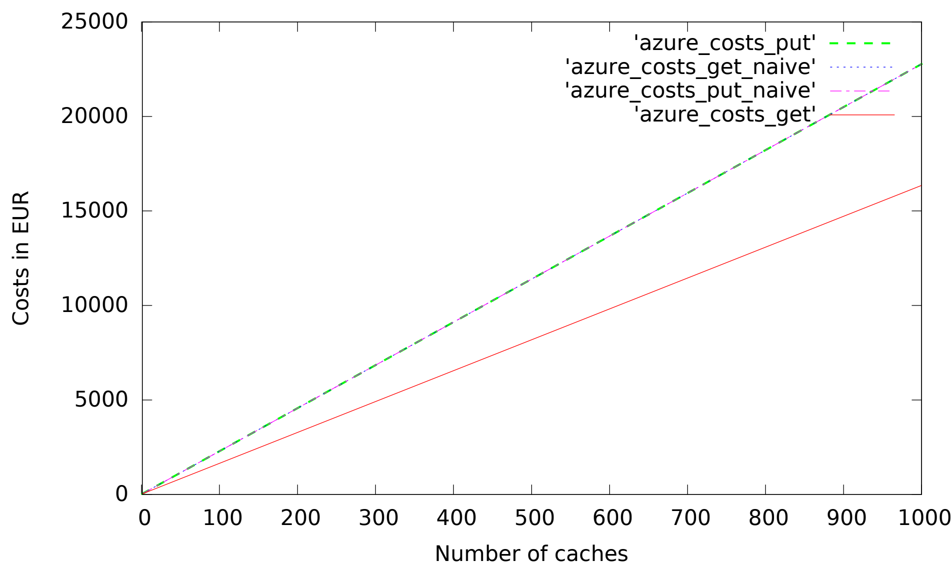

Abbildung 27: Kostenverteilung mit und ohne Optimierung. Die Kurven azure_costs_put, azure_costs_get_naive und azure_costs_put_naive liegen übereinander.

Die monatlichen Windows Azure-Kosten für die **optimierten** Synchronisationsraten sind (Abbildung 27):

Formel	**G/P**	a_{caches}=**1**	a_{caches}=**10**
$p_{get}(1350) + f(1350, 100MB, a_{caches})$	Get	29,57 €	176,58 €
$p_{put}(2880) + f(2880, 100MB, a_{caches})$	Put	36 €	240,81 €

Formel	**G/P**	a_{caches}=**100**	a_{caches}=**1000**
$p_{get}(1350) + f(1350, 100MB, a_{caches})$	Get	1646,66 €	16347,46 €
$p_{put}(2880) + f(2880, 100MB, a_{caches})$	Put	2288,98 €	22770,68 €

Ein Vergleich der Kostenverteilung der generischen Optimierungsformel und der

Azure-Kostenformel zeigt, dass die Funktionen miteinander korrelieren und die Optimierungsformel der Realität entspricht.

Aus den berechneten Kosten mit dem naiven Synchronisierungsansatz und dem optimierten Ansatz, lässt sich ein Ersparniswert in Prozent angeben. Nachfolgend konzentrieren wir uns auf die Ersparniswerte, die sich durch den Einsatz des Systems ergeben.

6.3.3 Kostenersparnis

Die Kostenersparnis wird mit folgender Formel berechnet. Es werden immer die Azure-Kosten ausgewertet. Die Variable x ist die optimierte Synchronisierungsrate und y die nichtoptimierte Rate. Auch hier muss bei der Verwendung der optimierten Synchronisierungsrate die Penaltyfunktion $p(x)$ berücksichtigt werden.

$$f_{saving}(x, y, d, n) = (1 - \frac{p(x) + f(x, d, n)}{f(y, d, n)}) \cdot 100\%$$

Für die Optimierung aus dem vorangehenden Abschnitt lassen sich folgende Werte berechnen:

Formel	**G/P**	a_{caches}**=1**	a_{caches}**=10**
$x_{opt_get_limited}$	Get	1350	1350
$x_{opt_put_limited}$	Put	2880	2880
$f_{saving}(x_{opt_get_limited}, 2880, 100MB, a_{caches})$	Get	18%	27%
$f_{saving}(x_{opt_put_limited}, 2880, 100MB, a_{caches})$	Put	0%	0%

Formel	**G/P**	a_{caches}**=100**	a_{caches}**=1000**
$x_{opt_get_limited}$	Get	1350	1350
$x_{opt_put_limited}$	Put	2880	2880
$f_{saving}(x_{opt_get_limited}, 2880, 100MB, a_{caches})$	Get	28%	28%
$f_{saving}(x_{opt_put_limited}, 2880, 100MB, a_{caches})$	Put	0%	0%

Es fällt auf, dass die Ersparnis nach dem Optimieren der Hochladehäufigkeit (put) bei null liegt. Bereits zu Beginn des Abschnitts wurde festgestellt, dass die optimierte Hochladehäufigkeit $x_{opt_put_limited}$ der naiven Häufigkeit entspricht. Eine Ersparnis von 0% ist eine offensichtliche Folge des mangelnden Optimierungsfreiraums beim Veröffentlichen von Änderungen an Datensätzen. Jede nichtveröffentlichte Änderung

verursacht enorme Penalty-Kosten von jedem einzelnen Cache, der einen veralteten Wert liest. Entsprechend ist es finanziell untragbar, eine Aktualisierung zurückzuhalten. Des Weiteren ist bemerkenswert, dass sich die optimierte Häufigkeit, wegen der Unabhängigkeit von der Anzahl der Caches, nicht ändert, während der Ersparniswert größer wird. Das ist damit in Verbindung zu bringen, dass die Fixkosten mit steigender Anzahl der Caches weniger ins Gewicht fallen.

Als nächstes wird die Kostenersparnis, abhängig von den ausschlaggebenden Parametern wie Datenmenge, Lese-/Schreibrate und dem Penalty-Kostenfaktor betrachtet. Hierfür werden Parameter wie folgt variiert:

Variable	**Wert(e)**	**Beschreibung**
$a_{writes} = a_{reads}$	30; $30 \cdot 24$; $30 \cdot 24 \cdot 4$; $30 \cdot 24 \cdot 12$	Pro Monat wird täglich, stündlich, viertelstündlich, alle 5 Minuten geschrieben/gelesen
a_{caches}	100	100 Caches nehmen am System teil
c_{read_old}	1; 5; 10; 20	Jedes Lesen eines veralteten Wertes kostet 1, 5, 10 und 20 €
d_{data_up}	$\frac{1}{1024}$; $\frac{100}{1024}$; $\frac{512}{1024}$; $\frac{1024}{1024}$	Es werden unterschiedliche Datenmengen hochgeladen, gerechnet in GB
d_{data_down}	$\frac{1}{1024}$; $\frac{100}{1024}$; $\frac{512}{1024}$; $\frac{1024}{1024}$	Es werden unterschiedliche Datenmengen heruntergeladen, gerechnet in GB

Es wird immer eine Variable als Laufvariable gewählt, alle anderen bleiben fest, mit Werten aus dem vorangehenden Abschnitt.

Laufvariable a_{writes}: Beim Variieren der Anzahl an Synchronisierungen pro Monat verteilen sich die Ersparnisse wie folgt (Abbildung 28).

Formel	**G/P**	a_{writes}**=30**	a_{writes}**=720**
		(Täglich)	**(Stündlich)**
$x_{opt_get_limited}$	Get	30	675
$x_{opt_put_limited}$	Put	30	720
$f_{saving}(x_{opt_get_limited}, a_{writes}, 100MB, 100)$	Get	0%	0%
$f_{saving}(x_{opt_put_limited}, a_{writes}, 100MB, 100)$	Put	0%	0%

Formel	**G/P**	a_{writes}**=2880**	a_{writes}**=8640**
		(Alle 15 Min.)	**(Alle 5 Min.)**
$x_{opt_get_limited}$	Get	1350	2338
$x_{opt_put_limited}$	Put	2880	8640
$f_{saving}(x_{opt_get_limited}, a_{writes}, 100MB, 100)$	Get	28%	53%
$f_{saving}(x_{opt_put_limited}, a_{writes}, 100MB, 100)$	Put	0%	0%

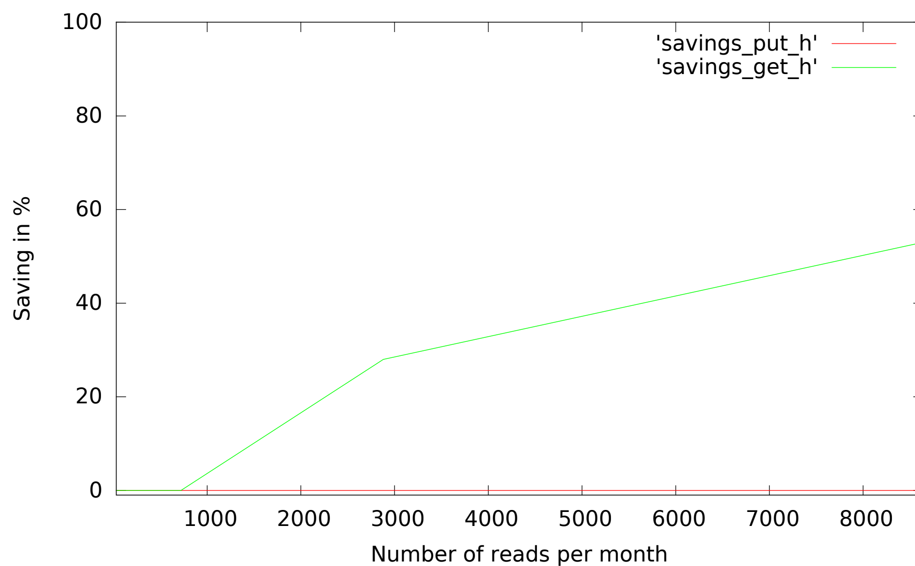

Abbildung 28: Kostenersparnis abhängig von der Lese-/Schreibrate. Der Vorteil entsteht erst ab c. a. 1000 Aktualisierungen pro Monat.

Laufvariable d_{data_up} und d_{data_down}: Beim Erhöhen der zu synchronisierender Datenmenge, sind die Ersparnisse entsprechend Abbildung 29.

Formel	**G/P**	d_{data_up}**=1 MB**	d_{data_up}**=100 MB**
$x_{opt_get_limited}$	Get	2880	1350
$x_{opt_put_limited}$	Put	2880	2880
$f_{saving}(x_{opt_get_limited}, 2880, d_{data_up}, 100)$	Get	0%	28%
$f_{saving}(x_{opt_get_limited}, 2880, d_{data_up}, 100)$	Put	0%	0%

Formel	**G/P**	d_{data_up}**=512 MB**	d_{data_up}**=1024 MB**
$x_{opt_get_limited}$	Get	597	422
$x_{opt_put_limited}$	Put	2880	2880
$f_{saving}(x_{opt_get_limited}, 2880, d_{data_up}, 100)$	Get	63%	73%
$f_{saving}(x_{opt_get_limited}, 2880, d_{data_up}, 100)$	Put	0%	0%

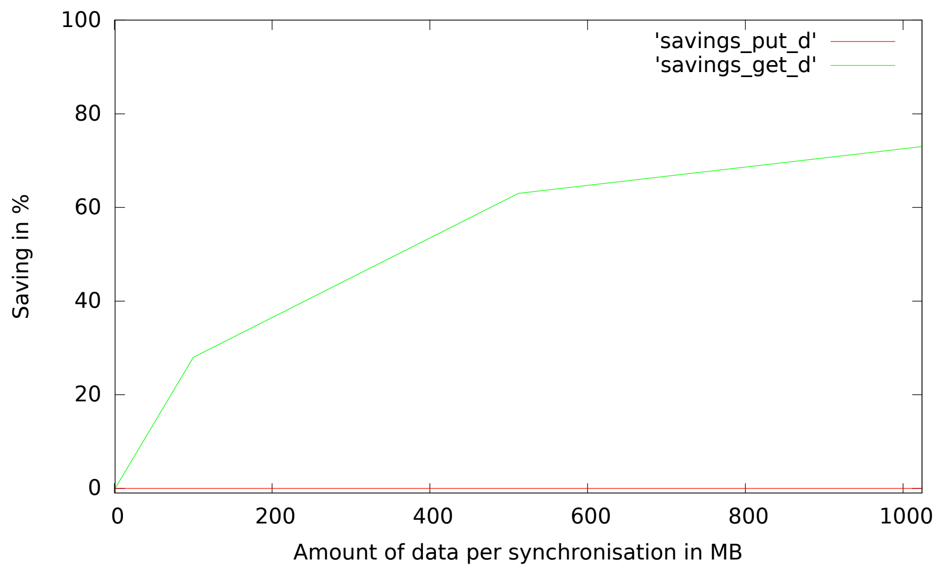

Abbildung 29: Kostenersparnis abhängig von der Datenmenge. Die Datenmenge beeinflusst die Ersparnisse erheblich.

Laufvariable c_{read_old}:Beim Senken der Kosten für jedes veraltete Lesen der Daten, erhöht sich die Ersparnis wie in Abbildung 30. Es hängt damit zusammen, dass Penalty-Kosten erheblich sinken und die optimale Synchronisationsrate einen kleineren Wert annimmt.

Formel	**G/P**	c_{read_old}**=1 €**	c_{read_old}**=5 €**
$x_{opt_get_limited}$	Get	604	1350
$x_{opt_put_limited}$	Put	2880	2880
$f_{saving}(x_{opt_get_limited}, 2880, 100MB, 100)$	Get	62%	28%
$f_{saving}(x_{opt_put_limited}, 2880, 100MB, 100)$	Put	0%	0%

Formel	**G/P**	c_{read_old}=**10 €**	c_{read_old}=**20 €**
$x_{opt_get_limited}$	Get	1909	2700
$x_{opt_put_limited}$	Put	2880	2880
$f_{saving}(x_{opt_get_limited}, 2880, 100MB, 100)$	Get	11%	0%
$f_{saving}(x_{opt_put_limited}, 2880, 100MB, 100)$	Put	0%	0%

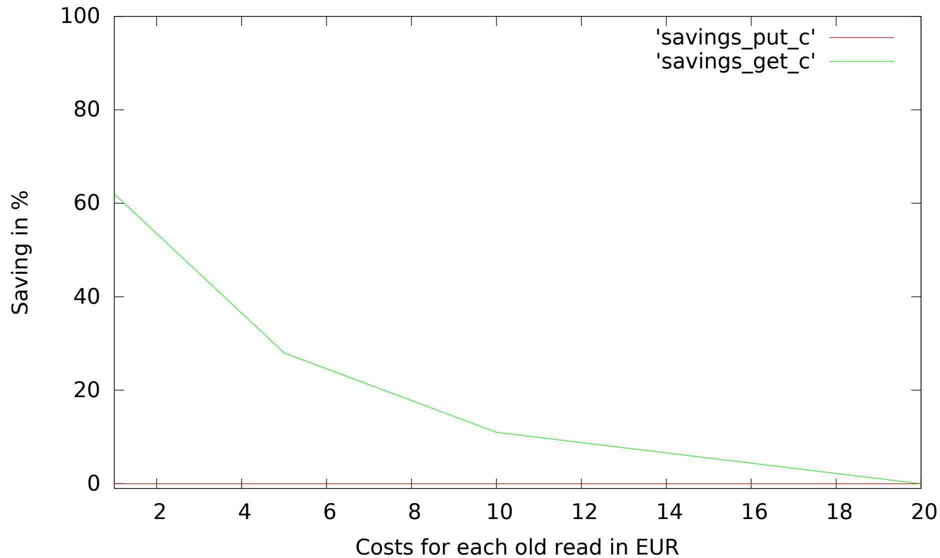

Abbildung 30: Kostenersparnis abhängig von den Kosten für das Lesen eines veralteten Datensatzes. Die Kosten lassen sich enorm einsparen, wenn geringer finanzieller Aufwand für das Lesen veralteter Daten entsteht.

In diesem Abschnitt wurden die Werte einzeln variiert und es wurde festgestellt, dass die Ersparnis maximal wird, wenn die Leserate und die Datenmenge erhöht und dabei die Kosten für das Lesen eines veralteten Datensatzes verringert werden.

6.3.4 Kosten für den Ausfall der Cloud

Die oben beschriebenen Kosten entstehen, solange die Cloud verfügbar ist. In Abschnitt 3.1.2 wurde festgehalten, dass die Cloud Monatlich 0,05% nicht verfügbar ist. In diesem Fall würden die monatlichen Stillstand-Kosten, bei 100 Caches ($a_{caches} = 100$), wie Folgt anfallen, dabei ist h die Anzahl der Leseoperationen und entspricht a_{reads}.

$f_{no_cloud}(h) = h \cdot 0.0005 \cdot c_{read_old} \cdot a_{caches}$

Formel	c_{read_old}=**1 €**	c_{read_old}=**5 €**	c_{read_old}=**10 €**	c_{read_old}=**20 €**
$f_{no_cloud}(30)$	1,5 €	7,5 €	15 €	30 €
$f_{no_cloud}(720)$	36 €	180 €	360 €	720 €
$f_{no_cloud}(2880)$	144 €	720 €	1440 €	2880 €
$f_{no_cloud}(8640)$	432 €	2160 €	4320 €	8640 €

Diese Kosten fallen jedoch nur an, wenn keine Synchronisierung stattfindet. Dieses System nutzt die Peer-to-Peer-Infrastruktur, um die Synchronisierung weiterhin zu ermöglichen und somit diese Kosten zu vermeiden.

Die Berechnungen aus vorangehenden Abschnitten zeigen einen klaren Trend, der nachfolgend im Fazit bewertet wird.

6.3.5 Fazit der Evaluation

Durch das Verringern der Hochlade-Häufigkeit, lassen sich keine Kosten einsparen, denn jedes nicht hochgeladene Update ist mit vervielfachten Kosten für das Lesen der falschen Daten von jedem einzelnen Cache verbunden. Entsprechend übersteigen in diesem Fall Penaltykosten die vergleichsweise geringen Transferkosten. Aus diesem Grund gleicht in allen Szenarien $x_{opt_put_limited}$ der naiven Synchronisierungshäufigkeit 1H. Dagegen lassen sich die Kosten durch die Optimierung der Get-Häufigkeit tatsächlich herabsenken. Es ist vom Anwendungsfall abhängig, wie viel sich einsparen lässt, jedoch zeigt die Evaluation, dass besonders bei Szenarien, in denen viel Traffic verursacht wird, eine deutliche Kostensenkung erzielt wird.

6.4 Zusammenfassung

Entsprechend dem Schwerpunkt dieser Arbeit, die Kommunikationskosten zu reduzieren, verringert die Lösung tatsächlich die Kosten, verglichen mit der naiven Synchronisierung. Der Vorteil dieser Lösung wird besonders in Szenarien deutlich, in denen eine hohe Datenmenge oft synchronisiert wird. Durch die Optimierungsfunktion wird die Synchronisierungshäufigkeit soweit verringert, bis die Kosten für das Lesen der veralteten Datensätze mit den Kosten für die Datenübertragung aus-

balanciert sind, wodurch sich ein Kostenminimum ergibt.

Im nachfolgenden Kapitel wird ein kurzer Überblick über die ganze Arbeit gegeben und ein Fazit gezogen.

7 Fazit

In diesem abschließenden Kapitel wird ein Rückblick der ganzen Arbeit gegeben und ein Ausblick in die Zukunft gewagt. Es wird der Kern der Aufgabenstellung, des zu entwickelnden Systems, des Entwurfs sowie der Implementierung hervorgehoben. Die Evaluierung untermauert den praktischen Teil der Arbeit. Diese Arbeit bietet weiterhin Forschungsraum für weitere Arbeiten - dies wird im letzten Abschnitt erläutert.

7.1 Zusammenfassung

Das Ziel dieser Ausarbeitung war es, bestehende Algorithmen zur Synchronisation zu untersuchen und ein System, das eine verbesserte Synchronisation ermöglicht, zu entwickeln. Dabei soll Cloud Computing als wichtiges Teil des Systems eingesetzt werden, um die Daten-Konsistenz und -Verfügbarkeit zu unterstützen.

Zunächst wurde das Synchronisierungsproblem untersucht. Dabei stellte sich heraus, dass Netzwerkpartitionierung die Datenkonsistenz oder -Verfügbarkeit sehr einschränkt.

Einige bestehende Systeme, wie z. B. Coda oder Bayou bieten gute Ansätze, um das Synchronisierungsproblem einzugrenzen, sind jedoch unzureichend in einigen Punkten. Als nützlich erwiesen sich das System BASE, das die Konsistenz des verteilten Systems sicherstellt und IceCube, das auftretende Konflikte auf der Anwendungsebene beseitigt. Es wurde ein System benötigt, um diese beiden Entwicklungen miteinander zu verzahnen.

So wurde zunächst die Infrastruktur mit den beteiligten Komponenten herauskristallisiert. Das verteilte System besteht aus dem zentralen Speicher in der Cloud und den geografisch verteilten Caches. So können Caches die gespeicherten Daten mit der Cloud synchronisieren. Ist die Cloud nicht verfügbar, dann werden Aktualisierungen über das Peer-to-Peer-Netzwerk ausgetauscht, in dem Caches miteinander direkt kommunizieren. Auf diese Weise wurde die Verfügbarkeit der Daten und Ausfallsicherheit der Komponenten sichergestellt.

Darauf aufbauend wurde die Grundidee für die Problemlösung entwickelt. Die bei-

den Systeme BASE und IceCube verwenden Operationen als grundlegende Elemente. Operationen sind ausführbare Programmteile, mit denen das Lesen und Schreiben der Daten ermöglicht wird. So greift eine Endanwendung über eine definierte Schnittstelle auf den Cache zu und übergibt die Parameter für das Ausführen einer Operation. Anschließend wird der durch eine Operation geänderte Datensatz mit BASE synchronisiert. Tritt beim Synchronisieren ein Konflikt auf, wird er mit IceCube aufgelöst.

Ein wichtiger Punkt der Aufgabe, ist es die Gesamtkosten für den Einsatz der Lösung zu reduzieren. Der Synchronisationsvorgang findet deshalb entsprechend einer Optimierungsfunktion statt, die die Anzahl der Synchronisierungen reduziert, um die Transferkosten zu senken, aber dabei die dadurch entstehenden Kosten für das Lesen eines veralteten Datensatzes berücksichtigt.

Das Ergebnis dieser Arbeit ist eine ganzheitliche Middleware-Lösung, die das Synchronisierungsproblem umfassend und effektiv reduziert. Die Implementierung der Middleware zeigt, dass trotz der in Realität auftretenden Netzwerkpartitionierung eine effiziente Synchronisation ermöglicht wird und Daten verfügbar und Konsistent gehalten werden. Die Implementierung der Beispielanwendung beweist die Effektivität des Systems hinsichtlich der der Möglichkeit, Daten zu verarbeiten, der Synchronisierung, sowie der Konfliktauflösung.

Die Evaluierung bestätigt die Kosteneffizienz. Dabei wurde festgestellt, dass eine größere zu synchronisierende Datenmenge, eine größere Synchronisierungshäufigkeit und die geringere Kosten für das Lesen eines veralteten Datensatzes am meisten Optimierungsraum bieten und somit einen höherer Erparnis-Wert erreicht wird.

I. Foster prophezeit in [FZRL08], dass Cloud Computing und Client Computing werden in Zukunft stärker ausgeprägt sein und dementsprechend Themen wie Netzwerkpartitionierung, Caching und Replizierung eine stärkere Rolle spielen werden als bisher. Aus dieser Perspektive ist diese Lösung zukunftssicher und bildet eine Basis für spätere wissenschaftliche Arbeiten. Nachfolgend werden konkrete wissenschaftliche Arbeiten vorgeschlagen, die auf dieser aufbauen können.

7.2 Ausblick auf weitere Arbeiten

Dieses System kann dafür dienen, Daten mit Hilfe von Operationen sowohl im Rohzustand, als auch anwendungsspezifisch zu erfassen und zu untersuchen. Es ist also möglich, Sensordaten mit weiteren Daten aus dem Kontext des Systems zu verbinden. Somit können sie semantisch gekennzeichnet werden. Ohne das Wissen der Anwendungssemantik ist es sehr schwierig, hochwertige semantische Metadaten zu produzieren. Deshalb wäre es in einer wissenschaftlichen Arbeit zu untersuchen, wie Metadaten mit Operationen auf semantischer Ebene erfasst und den verarbeitenden Anwendungen übermittelt werden können.

Es ist möglich, eine zusätzliche Komponente zu entwickeln, die die Reihenfolge der Operationen beim Zusammenführen der Änderungshistorien mit Hilfe einer Grammatik prüft. Die Grammatik muss dabei die Standardabläufe des Systems, d. h. die Operationsabfolgen beschreiben, um möglichst viele Fälle zu berücksichtigen. Sie würde einerseits den Suchraum für die korrekten Reihenfolgen der Operationen eingrenzen, andererseits würde sie die Korrektheit der Zusammenführung sicherstellen. Somit wird eine Effizienzsteigerung erzielt und die Korrektheit bestätigt. Es muss untersucht werden, welche Vorteile die Erweiterung bringt und welche Nachteile müssen in Kauf genommen werden. Dies kann in Form einer weiterführenden Arbeit geschehen.

Diese Arbeit bietet Freiraum, um weitere Forschungen für eine verbesserte Konfliktauflösung oder Datenerfassung zu betreiben. Das System bietet eine flexible Basis, um neue Algorithmen, die im Laufe der Forschung entwickelt werden, einfach anzuschließen und in der Praxis zu testen. Somit stellt die Lösung in der Praxis ein nützliches System dar und in der Forschung eine Basis für weitere Arbeiten.

Literatur

[ALO00] Adya, A. ; Liskov, B. ; O'Neil, P.: Generalized isolation level definitions. In: *Data Engineering, 2000. Proceedings. 16th International Conference on* IEEE, 2000, S. 67–78

[Ama] Amazon: *Amazon EC2 Service Level Agreement*. http://aws.amazon.com/de/ec2-sla/. – [Online; accessed 26-November-2012]

[BC+08] Brodkin, B.J. ; Computing, C. u. a.: Gartner: Seven cloud-computing security risks. In: *Infoworld* (2008), S. 2–3

[BGL+06] Baldoni, R. ; Guerraoui, R. ; Levy, R. ; Quéma, V. ; Piergiovanni, S.: Unconscious eventual consistency with gossips. In: *Stabilization, Safety, and Security of Distributed Systems* (2006), S. 65–81

[BKNT10] Baun, C. ; Kunze, M. ; Nimis, J. ; Tai, S.: *Cloud computing: Webbasierte dynamische IT-Services*. Springer, 2010

[BTN00] Barton, J.J. ; Thatte, S. ; Nielsen, H.F.: SOAP messages with attachments. In: *W3C note* 11 (2000)

[CCM+01] Christensen, E. ; Curbera, F. ; Meredith, G. ; Weerawarana, S. u. a.: *Web services description language (WSDL) 1.1*. 2001

[CLMR04] Crucitti, P. ; Latora, V. ; Marchiori, M. ; Rapisarda, A.: Error and attack tolerance of complex networks. In: *Physica A: Statistical Mechanics and its Applications* 340 (2004), Nr. 1, S. 388–394

[Cor91] Corkill, D.D.: Blackboard systems. In: *AI expert* 6 (1991), Nr. 9, S. 40–47

[Die08] Dierks, T.: The transport layer security (TLS) protocol version 1.2. (2008)

[DPS+94] Demers, A. ; Petersen, K. ; Spreitzer, M. ; Ferry, D. ; Theimer, M. ; Welch, B.: The Bayou architecture: Support for data sharing among mobile users. In: *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on* IEEE, 1994, S. 2–7

[Fos95] Foster, I.: *Designing and building parallel programs*. Bd. 95. Addison-Wesley Reading, MA, 1995

[FZRL08] Foster, I. ; Zhao, Y. ; Raicu, I. ; Lu, S.: Cloud computing and grid computing 360-degree compared. In: *Grid Computing Environments Workshop, 2008. GCE'08* Ieee, 2008, S. 1–10

[GHJV10] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Pearson Education, 2010

[GHOS96] Gray, J. ; Helland, P. ; O'Neil, P. ; Shasha, D.: The dangers of replication and a solution. In: *ACM SIGMOD Record* Bd. 25 ACM, 1996, S. 173–182

[GL02] Gilbert, S. ; Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In: *ACM SIGACT News* 33 (2002), Nr. 2, S. 51–59

[GRR+99] Guy, R. ; Reiher, P. ; Rather, D. ; Gunter, M. ; Ma, W. ; Popek, G.: Rumor: Mobile data access through optimistic peer-to-peer replication. In: *Lecture notes in computer science* (1999), S. 254–265

[HMPT03] Hansmann, U. ; Mettala, R.M. ; Purakayastha, A. ; Thompson, P.: *SyncML: Synchronizing and managing your mobile data*. Prentice Hall, 2003

[JN01] Jonsson, A. ; Novak, L.: SyncML- Getting the mobile Internet in sync. In: *ERICSSON REV(ENGL ED)* 78 (2001), Nr. 3, S. 110–115

[KRSD01] Kermarrec, A.M. ; Rowstron, A. ; Shapiro, M. ; Druschel, P.: The IceCube approach to the reconciliation of divergent replicas. In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing* ACM, 2001, S. 210–218

[KS93] Kumar, P. ; Satyanarayanan, M.: Log-based directory resolution in the Coda file system. In: *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on* IEEE, 1993, S. 202–213

[KSS94] Kumar, P. ; Satyanarayanan, M. ; SCIENCE., CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF C.: *Flexible and safe resolution of file conflicts*. Defense Technical Information Center, 1994

[LHSH05] Loo, B. ; Huebsch, R. ; Stoica, I. ; Hellerstein, J.: The case for a hybrid P2P search infrastructure. In: *Peer-to-Peer Systems III* (2005), S. 141–150

[Mic] Microsoft: *Vereinbarungen zum Servicelevel (SLAs)*. http://www.windowsazure.com/de-de/support/legal/sla/. – [Online; accessed 26-November-2012]

[PB99] Phatak, S.H. ; Badrinath, BR: Conflict resolution and reconciliation in disconnected databases. In: *Database and Expert Systems Applications, 1999. Proceedings. Tenth International Workshop on* IEEE, 1999, S. 76–81

[Pos81] Postel, J.: Request for Comments (RFC): 793. In: *Transmission Control Protocol-DARPA Internet Program Protocol Specification, abrufbar unter: http://www. ibiblio. org/pub/docs/rfc/rfc793. txt (letzter Abruf: 21.08. 2006)* (1981)

[Pri08] Pritchett, D.: Base: An acid alternative. In: *Queue* 6 (2008), Nr. 3, S. 48–55

[Rat98] Ratner, D.H.: *Roam: A scalable replication system for mobile and distributed computing*, Citeseer, Diss., 1998

[Rei] Reiher, Peter: *Rumor 1.0 User's Manual*. ftp://ftp.cs.ucla.edu/pub/rumor/rumor_users_manual.ps. – [Online; accessed 18-July-2012]

[Rei01] Reitz, Rainer: *Verschluesselung fuer Dropbox leicht gemacht*. http://www.computerwoche.de/software/office-collaboration/1934388/. Version: 2001. – [Online; accessed 19-September-2012]

[RHR+94] Reiher, P. ; Heidemann, J. ; Ratner, D. ; Skinner, G. ; Popek, G. u. a.: *Resolving file conflicts in the Ficus file system*. UCLA Computer Science Department, 1994

[RRPK01] Ratner, D. ; Reiher, P. ; Popek, G.J. ; Kuenning, G.H.: Replication requirements in mobile environments. In: *Mobile Networks and Applications* 6 (2001), Nr. 6, S. 525–533

[SKK+90] Satyanarayanan, M. ; Kistler, J.J. ; Kumar, P. ; Okasaki, M.E. ; Siegel, E.H. ; Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. In: *Computers, IEEE Transactions on* 39 (1990), Nr. 4, S. 447–459

[SNS88] Steiner, J.G. ; Neuman, C. ; Schiller, J.I.: Kerberos: An authentication service for open network systems. In: *USENIX conference proceedings*, 1988, S. 191–200

[Spr] Springer_Fachmedien_Wiesbaden_GmbH: *Gabler Wirtschaftslexikon*. http://wirtschaftslexikon.gabler.de/. – [Online; accessed 10-December-2012]

[TDP+94] Terry, D.B. ; Demers, A.J. ; Petersen, K. ; Spreitzer, M.J. ; Theimer, M.M. ; Welch, B.B.: Session guarantees for weakly consistent replicated data. In: *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on* IEEE, 1994, S. 140–149

[TM96] Tridgell, A. ; Mackerras, P.: *The rsync algorithm*. 1996

[TS08] Tanenbaum, A.S. ; Steen, M.: Verteilte Systeme: Prinzipien und Paradigmen. 2 (2008)

[WTK+08] Wang, L. ; Tao, J. ; Kunze, M. ; Castellanos, A.C. ; Kramer, D. ; Karl, W.: Scientific cloud computing: Early definition and experience. In: *High Performance Computing and Communications, 2008. HPCC'08. 10th IEEE International Conference on* IEEE, 2008, S. 825–830

[Yen05] Yeniay, O.: Penalty function methods for constrained optimization with genetic algorithms. In: *Mathematical and Computational Applications* 10 (2005), Nr. 1, S. 45–56

[YYLW08] Yang, H. ; Yang, P. ; Lu, P. ; Wang, Z.: A SyncML Middleware-Based Solution for Pervasive Relational Data Synchronization. In: *Network and Parallel Computing* (2008), S. 308–319