Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Marian Triebe

Optimistische Datenreplikation mit CRDTs im C++ Actor Framework

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Marian Triebe

Optimistische Datenreplikation mit CRDTs im C++ Actor Framework

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Thomas C. Schmidt
Zweitgutachter: Prof. Dr. Martin Becke

Eingereicht am: 15. Mai 2017

Marian Triebe

Thema der Arbeit

Optimistische Datenreplikation mit CRDTs im C++ Actor Framework

Stichworte

Conflict-free Replicated Datatypes (CRDTs), Datenreplikation, CAP-Theorem, Konsistenzmodelle, Aktoren, Verteilte Systeme, Logische Uhren

Kurzzusammenfassung

Das Aktormodell beschreibt isolierte, nebenläufige Entitäten, die mit transparenter Nachrichtenübermittlung kommunizieren. Es erlaubt, lokale, nebenläufige Kerne auszunutzen sowie skalierbare verteilte Systeme zu implementieren. Diese Bachelorarbeit vereint konfliktfrei replizierbare Datensturkturen, sogenannte „Conflict-free Replicated Datatypes“ (CRDTs), mit einem Aktor-Programmierkonzept. CRDTs können nebenläufig und unabhängig voneinander geändert werden, ohne dass eine Zugriffskoordination benötigt wird. In dieser Arbeit wird ein Modul für das C++ Actor Framework (CAF) zur Replikation von Daten mit Hilfe von CRDTs entworfen und implementiert. Hierfür wurde zwischen verschiedenen Zugriffskonzepten auf Replikate abgewogen. Außerdem wurden verschiedene Klassen von CRDTs gegenübergestellt. Weiterhin wurde ein Verfahren für die Verteilung von CRDTs im verteilten System entworfen, welches Konvergenz der Replikate zusichert. Der entstandene Entwurf wurde implementiert und schließlich evaluiert. Hierbei wurden insbesondere Kriterien wie vorhersagbare Konvergenzgeschwindigkeit sowie Datenaufkommen im Netzwerk berücksichtigt.

Marian Triebe

Title of the paper

Optimistic data replication with CRDTs in the C++ Actor Framework

Keywords

Conflict-free Replicated Datatypes (CRDTs), Data Replication, CAP-Theorem, Consistency Models, Actors, Distributed Systems, Logical Clocks

Abstract

The actor model describes isolated concurrent entities that communicate with transparent message passing. It allows local concurrent cores to be utilized as well as to implement scalable distributed systems. This thesis combines conflict-free replicated datatypes (CRDTs) with an actor programming concept. CRDTs can be changed concurrently and independently, without the need for access coordination. In this work, a module for the C ++ Actor Framework (CAF) is designed and implemented for the replication of data using CRDTs. For this purpose, different access concepts on replicas where compared. In addition, different classes of CRDTs were compared. Further, a method for the distribution of CRDTs in distributed systems has been designed which ensures convergence of replicas. The resulting design was implemented and finally evaluated. In particular, criterions such as predictable convergence speed as well as data volumes in the network were taken into account.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Zielsetzung . . . 1

1.3 Aufbau der Arbeit . . . 1

2 Hintergrund & Verwandte Arbeiten **3**

2.1 Das Aktormodell . . . 3

2.1.1 Erlang . . . 4

2.1.2 Akka . . . 4

2.1.3 Das C++ Actor Framework (CAF) . . . 7

2.2 Das CAP-Theorem . . . 10

2.3 Logische Uhren . . . 11

2.3.1 Lamport-Uhr . . . 12

2.3.2 Vektoruhr . . . 13

2.3.3 Plausible Clocks . . . 14

2.4 Datenreplikation . . . 15

2.4.1 Symmetrische Replikation . . . 16

2.4.2 Asymmetrische Replikation . . . 17

2.4.3 Synchrone Replikation . . . 17

2.4.4 Asynchrone Replikation . . . 18

2.4.5 Multi- & Single- Master Systeme . . . 19

2.4.6 Zustandstransfer & Operationstransfer . . . 19

2.4.7 Scheduling von Operationen . . . 20

2.4.8 Verfahren zum Auflösen, Reduzieren und Vermeiden von Konflikten . 21

2.5 Konsistenzmodelle . . . 23

2.5.1 Starke Konsistenzen . . . 23

2.5.2 (Strong) Eventual Consistency . . . 24

3 Conflict-free Replicated Datatypes (CRDTs) **25**

3.1 Asynchrone Objekt-Replikation . . . 25

3.2 Zustandsbasiert (CvRDT) . . . 26

3.3 Operationsbasiert (CmRDT) . . . 28

3.4 δ-CRDTs . . . 29

3.5 Bekannte CRDTs . . . 30

3.5.1 Last writer wins Register (LWW-Register) . . . 30

3.5.2 Multi-Value Register (MV-Register) . . . 30

3.5.3 Flag . . . 31
3.5.4 Grow-only Counter (GCounter) . . . 32
3.5.5 PN-Counter . . . 33
3.5.6 Grow-only Set (GSet) . . . 34
3.5.7 Two-Phase Set (2PSet) . . . 34
3.5.8 Last-Writer-Wins-Element Set (LWW-Element-Set) . . . 35

4 Design des CAF-CRDT-Moduls 37
4.1 Design-Ziele . . . 37
4.2 Integration von CRDTs in CAF . . . 38
4.2.1 Zugriff auf CRDT-Instanzen . . . 38
4.2.2 Adressierung von Replikaten . . . 40
4.3 Auswahl einer geeigneten CRDT-Klasse für CAF . . . 41
4.3.1 Anforderungen an das Kommunikations-System . . . 41
4.3.2 Auswirkung der CRDT-Klasse auf die Verteilung . . . 43
4.3.3 Entscheidung über die CRDT-Klasse . . . 44
4.4 Austausch von Meta-Informationen . . . 45
4.5 Austausch von Replikat-Daten . . . 46
4.6 Robuste Ausführung von Operationen . . . 47
4.6.1 Diskussion . . . 49
4.7 Lebenszeit von Replikaten . . . 49
4.7.1 Initialisierung . . . 49
4.7.2 Entfernen von Replikaten . . . 50
4.8 Software-Design . . . 52
4.8.1 Klassendiagramm . . . 52
4.8.2 Nachrichtenfluss im CAF-CRDT-Modul . . . 54
4.8.3 Entwurf einer Programmierschnittstelle (API) . . . 55
4.8.4 Erweiterbarkeit um eigene CRDTs . . . 59

5 Implementierung 61
5.1 Modul Konfiguration . . . 62
5.2 Verwaltung von Replikat-Daten . . . 63
5.2.1 Zustand der *replica*-Klasse . . . 64
5.2.2 Verwalten von Abonnenten . . . 65
5.2.3 Empfangen von Updates . . . 65
5.2.4 Robustes Ausführen von Operationen . . . 68
5.3 Der Replicator . . . 73
5.3.1 Instanziierung und Stopp des CAF-CRDT-Moduls . . . 74
5.3.2 Typisiertes Interface des *replicator_actor* . . . 76
5.3.3 Erstellen von *replica*-Aktor Instanzen . . . 80
5.4 Distribution Layer . . . 82
5.4.1 Handshake mit neu entdeckten Knoten . . . 83
5.4.2 Periodisches Austauschen von Replikat-Bezeichnern . . . 84

5.4.3 Änderung an den lokalen Replikat-Bezeichnern 86
5.4.4 Buffern und Senden von δ-CRDT-Zuständen 86
5.5 Erweiterung um eigene CRDTs 87
5.6 Diskussion 90

6 Evaluation 91
6.1 Testumgebung 91
6.2 Unit-Tests 91
6.2.1 GSet 92
6.2.2 GCounter 94
6.2.3 MV-Register 95
6.3 Verifikation der Konvergenz 96
6.4 Datenaufkommen im Netzwerk 98
6.4.1 Vorgehen und Erwartung 98
6.4.2 Aufbau 98
6.4.3 Ergebnisse & Diskussion 99
6.5 Datenaufkommen im Multi-Writer-Fall 108
6.5.1 Vorgehen und Erwartung 108
6.5.2 Ergebnisse & Diskussion 109
6.6 Performance lokaler Verteilung 111
6.6.1 Vorgehen und Erwartung 111
6.6.2 Ergebnisse & Diskussion 112
6.7 Auswirkung der Systemgröße auf die Konvergenzgeschwindigkeit 115
6.7.1 Vorgehen und Erwartungen 115
6.7.2 Ergebnisse & Diskussion 116

7 Schlussbetrachtung 121
7.1 Fazit 121
7.2 Ausblick 121

Tabellenverzeichnis

4.1 Die verschiedenen Fehlersemantiken beim Versenden von Nachrichten 42

6.1 Testfälle mit Eingabe und erwarteter Ausgabe für die GSet-Klasse 92
6.2 Erwartetes Datenaufkommen bei Verwendung von *CvRDTs* in Bezug zu verschiedener Anzahl an Knoten und verschiedener Anzahl an Operationen . . . 99
6.3 Gegenüberstellung von erwarteten und gemessenen Werten des *GSet* als *CvRDT* 103
6.4 Erwartetes Datenaufkommen bei Verwendung von δ-CRDTs in Bezug zu verschiedener Anzahl an Knoten und verschiedener Anzahl an Operationen . . . 104
6.5 Gegenüberstellung von erwarteten und gemessenen Werten des *GSet* als δ-CRDT 107

Abbildungsverzeichnis

2.1 Zustände der Mitgliedschaft eines Knotens in Akka Cluster (vgl.[1, p. 338]) . . 6
2.2 Architektur des C++ Actor Frameworks 7
2.3 Routing von Nachrichten über andere CAF-Knoten 9
2.4 Das CAP-Theorem als Venn-Diagramm 10
2.5 „Clock Condition“ (vgl. [2, p. 560]) 12
2.6 Mapping von Knoten auf Slots im Vektor 15
2.7 „Update anywhere“, jedes Replikat kann geändert werden 16
2.8 Beispiel des Update-Ablaufs bei Primary-Copy 17
2.9 Möglichkeiten im Umgang mit Konflikten (vgl. [3, p. 11]) 21

3.1 Hasse-Diagramm eines Join-Halbverbands 27
3.2 Zustandsbasierte Replikation [4, p. 6] 28
3.3 Zustandsbasiertes Last Writer Wins Register (LWW-Register) 30
3.4 Zustandsbasiertes Multi-Value Register (MV-Register) (vgl. [4, p. 20]) 31
3.5 Zustandsbasierte Flags 31
3.6 Zustandsbasierter Grow-only Counter (CvRDT GCounter) (vgl. [5, p. 4]) . . . 32
3.7 Zustandsbasierter Positiv-Negativ Counter (CvRDT PNCounter) 33
3.8 Zustandsbasiertes Grow-Only Set (CvRDT GSet) 34
3.9 Zustandsbasiertes Two-Phase Set (CvRDT 2PSet) 35
3.10 Zustandsbasiertes Last Writer Wins Set (CvRDT LWWSet) 35

4.1 Möglichkeiten für den konkurrierenden lokalen Zugriff auf Replikate 39
4.2 Kodierung zweier Replikat-Bezeichners 41
4.3 Periodischer Austausch von Meta-Informationen zwischen zwei CAF-Knoten 45
4.4 Initialisieren und Entfernen von lokalen Replikaten 46
4.5 Wichtige Klasse im CAF-CRDT-Modul 52
4.6 Kommunikationswege zwischen verschiedenen Software-Entitäten 54
4.7 UML-Klassendiagramm von *base_datatype* 60

6.1 Topologie des mit Mininet erstellten Netzwerks 98
6.2 Gemessenes Datenaufkommen im Netzwerk bei Verwendung von *CvRDTs*, dabei wurden 1000 Operationen auf dem *GSet* durchgeführt 100
6.3 Gemessenes Datenaufkommen im Netzwerk bei Verwendung von *CvRDTs*, dabei wurden 10000 Operationen auf dem *GSet* durchgeführt 102
6.4 Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 1000 Operationen auf dem *GSet* durchgeführt 105

6.5 Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 10000 Operationen auf dem *GSet* durchgeführt 106
6.6 Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 1000 Operationen auf einem *GCounter* ausgeführt 109
6.7 Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 10000 Operationen auf einem *GCounter* ausgeführt 110
6.8 Total 1000 Operationen, variable Anzahl an Abonnenten 112
6.9 Total 10000 Operationen, variable Anzahl an Abonnenten 113
6.10 Jeder Abonnent mit 1000 eigenen Operationen, variable Anzahl an Abonnenten 114
6.11 Verteilung der Zeitdifferenz bei unterschiedlicher Anzahl an Abonnenten auf nur einen Knoten 116
6.12 Verteilung Zeitdifferenz bei unterschiedlicher Anzahl an Knoten bei einem Flush-Intervall von einer Sekunde. 118
6.13 Verteilung Zeitdifferenz bei unterschiedlicher Anzahl an Knoten bei einem Flush-Intervall von zwei Sekunden 119
6.14 Verteilung Zeitdifferenz bei unterschiedlicher Anzahl an Knoten bei einem Flush-Intervall von vier Sekunden 120

Auflistungsverzeichnis

4.1 Konfiguration eines Aktor-Systems, welches das CAF-CRDT-Modul verwendet 55
4.2 Verwendung eines Replikats innerhalb eines nicht typisierten Aktors 56
4.3 Beispiel eines typisierten Aktors, welcher ein Replikat innerhalb seines Zustands hält 57
4.4 Operation auf einem Replikat ausführen, ohne Zustand im Aktor zu halten . . 58

5.1 Konfiguration des CRDT-Moduls 62
5.2 Zustand eines Replica-Aktors 64
5.3 An- und Abmelden an einem Replikat 65
5.4 Entpacken eines δ-CRDT-Zustands aus einer Nachricht 66
5.5 Empfang eines Updates als Nachricht 66
5.6 Empfang von Updates als Nachricht 67
5.7 Verteilung von Updates an Abonnenten 68
5.8 Operation auf einem Replikat ausführen, ohne Zustand im Aktor zu halten . . 69
5.9 Zustand von Write-Read-Aktoren 69
5.10 Aktor für schreibende Operationen 70
5.11 Aktor für lesende Operationen 72
5.12 Timeout-Handler von Write-Read-Aktoren 73
5.13 Initialisierung und Start des CAF-CRDT-Moduls 74
5.14 Callbacks für die Hooks des I/O-Moduls 75
5.15 Herunterfahren des CAF-CRDT-Moduls 76
5.16 Öffentlich zugängliches Interface des *replicator_actor* 77
5.17 Privates Interface des *replicator_actor* 79
5.18 Instanziieren neuer Replikate 81
5.19 Zustand der *distribution_layer*-Klasse 82
5.20 Aktion nach Entdecken eines neuen Knotens 83
5.21 Replikat-Bezeichner von bekannten Knoten anfordern 84
5.22 Knoten empfängt Anfrage nach Replikat-Bezeichnern 84
5.23 Knoten empfängt die Replikat-Bezeichner eines anderen Knotens 85
5.24 Änderungen am lokalen (eigenen) *node_data* Datensatz 86
5.25 Einfügen von δ-CRDT-Zuständen in den Sendebuffer 86
5.26 Flushen der Sendebuffer 87
5.27 Implementierung eines Benutzer-Definierten CRDT 88

6.1 Unit-Test der GSet-Klasse 93
6.2 Unit-Test der GCounter-Klasse 94

6.3 Unit-Test der MV-Register-Klasse 95
6.4 Aktor für den Konvergenztest 97

1 Einleitung

1.1 Motivation

Replikation von Daten auf verschiedene Knoten in verteilten Systemen ist ein allgemeines Problem der Informatik. Dabei sind die Gründe für Datenreplikation vielfältig. Zum einen werden Daten auf verschiedene Knoten repliziert, um eine Sicherheit gegenüber Datenverlust zu gewährleisten, andererseits um die Datenlokalität und somit Zugriffszeiten niedrig zu halten. Verschiedene Arten und Vorgehensweisen der Datenreplikation erlauben es, anwendungsfall-spezifisch zu optimieren und eine passende Lösung zu implementieren.
Conflict-free Replicated Datatypes (CRDTs) bieten eine Möglichkeit, Daten zwischen Knoten zu replizieren und dabei geringe Lese- und Schreib-Latenzen zu gewährleisten. Ein Vorteil von CRDTs ist, dass keine zentrale Zugriffskoordination zwischen Knoten benötigt wird. Durch ihre Eigenschaften eignen sich CRDTs außerdem besonders für asynchrone und nebenläufige Abläufe. CRDTs beruhen auf mathematischen Gesetzen. Deshalb ist es nicht möglich, jeden beliebigen Datentyp als CRDT zu implementieren.

1.2 Zielsetzung

Diese Arbeit stellt die Grundlagen und verschiedene Konzepte der Replikation von Daten vor. Dabei werden im späteren Verlauf sogenannte „Conflict-free Replicated Datatypes" (CRDTs) vorgestellt. Ziel dieser Arbeit ist es, Anforderungen, Design sowie Implementierung und Evaluation eines CRDT-Moduls für das C++ Actor Framework (CAF) zu erarbeiten, welches im Rahmen dieser Arbeit entwickelt wird.

1.3 Aufbau der Arbeit

In Kapitel 2 wird das zugrundeliegende Aktormodell sowie Implementierungen davon erläutert, insbesondere das C++ Actor Framework (CAF). Weiterhin werden Grundlagen von verteilten Systemen sowie der Replikation von Daten vorgestellt. In Kapitel 3 werden verschiedene Klassen von CRDTs inklusive des notwendigen Grundstocks präsentiert. Des Weiteren sind

bekannte CRDTs erklärt, welche auch von der Implementierung unterstützt werden sollen. In Kapitel 4 wird ein Entwurf für ein CRDT-Modul des C++ Actor Frameworks erstellt und diskutiert. Kapitel 5 stellt die Implementierung dieses Entwurfs vor, welche in Kapitel 6 evaluiert wird. Abschließend wird in Kapitel 7 ein Fazit gezogen.

2 Hintergrund & Verwandte Arbeiten

2.1 Das Aktormodell

In heutigen Computern werden überwiegend multi- und many-core CPUs verwendet. Diese stark nebenläufige Hardware lässt sich mit üblichen Programmiermodellen nur umständlich verwenden und nur schwer optimal ausnutzen. In üblichen Programmiermodellen werden zur lokalen Verteilung und Synchronisation Primitiven wie *Semaphore*, *Mutexe* und *Threads* verwendet. Durch die Verwendung dieser Primitiven steigt die Komplexität der Software. Werden diese Primitive falsch verwendet, kommt es zu schlechter Performance der Software oder sogar zu Fehler wie *Race-Conditions* oder *Deadlocks*.
Ein alternatives Programmiermodell ist das Aktormodell [6]. Dabei handelt es sich um isolierte Software-Entitäten, welche ausschließlich über den Austausch von Nachrichten kommunizieren. Aktoren kapseln ihren Zustand und erlauben so keinen direkten Zugriff auf vom Aktor gekapselte Daten. Aktoren senden Nachrichten entweder an sich selbst oder an andere Aktoren. Die versendeten Nachrichten kommen dabei in der Mailbox des Ziel-Aktors an. Jeder Aktor besitzt eine Mailbox, auf die nur der besitzende Aktor Zugriff hat. Sogenannte *behaviors* (Verhalten) beschreiben, wie und auf welche Nachrichten ein Aktor reagiert. Da ein Aktor seine *behaviors* zur Laufzeit verändern kann, eignen sich Aktoren zum Modellieren von Zustandsautomaten (FSM).
Da Aktoren keinen geteilten Zustand haben und exklusiv auf Speicherregionen arbeiten, ist keine Synchronisation der Zugriffe auf den Speicher notwendig. Die einzige Ausnahme bildet die Mailbox. Sie muss synchronisiert werden, da andere Aktoren Nachrichten in fremde Mailboxen schreiben können.
Zusammenfassend lässt sich sagen, dass sich Aktoren deshalb für die nebenläufige Verarbeitung eignen, da diese unabhängig von einander ausgeführt werden können. Außerdem adressiert das Aktormodell nicht nur lokale Nebenläufigkeit, sondern auch transparente Verteilung von Aktoren in einem verteilten System.

2.1.1 Erlang

Erlang [7] ist die erste Implementierung des Aktormodells, jedoch ist die Entwicklung parallel zur formalen Definition unabhängig davon verlaufen. Erlang wurde 1987 von Joe Armstrong bei Ericsson mit dem Ziel entwickelt, eine Laufzeit und Sprache bereitzustellen, welche den Anforderungen in (1) Parallelität, (2) Verfügbarkeit und (3) Fehlertoleranz in verteilten Systemen gerecht wird. Armstrong fasst in seiner Doktorarbeit [8] die Anforderungen, die eine verteilte Umwelt stellt, zusammen und zeigt, wie Erlang diesen gerecht wird.

2.1.2 Akka

Akka [1] ist eine Implementierung des Aktormodells und ist unter Java und Scala verfügbar. Die erste Version von Akka wurde 2009 veröffentlicht und ist Teil der Standard-Bibliothek von Scala. Akka stellt ein experimentelles Cluster-Modul bereit, welches unter anderem Datenreplikation per Conflict-Free-Replicated-Datatypes (CRDTs) bereitstellt. Im Folgenden sind die für die Datenreplikation wichtigen Funktionen des Akka Cluster-Moduls erläutert.

Akka Cluster

Mit Akka Cluster wird nicht nur die Replikation von Daten über Knotengrenzen hinweg angeboten, sondern auch eine Zugehörigkeitskontrolle zu einem „Cluster“. Ein logischer Cluster ist hierbei ein Verbund von Knoten. Knoten sind Akka-Laufzeitinstanzen, wobei eine Maschine mehrere Akka-Laufzeitinstanzen beinhalten kann. In diesem logischen Cluster werden unter anderem Daten repliziert und konvergent gehalten. CRDTs werden vom Akka Cluster-Modul mit einer Key/Value-Store API bereitgestellt.
Das Verfahren für die Mitgliedschaft des Clusters (das so genannte *Membership*) basiert auf dem Verfahren von Amazons Dynamo [9] und Bashos Riak [10]. Änderungen am Cluster werden mit Hilfe eines *Gossip*-Protokolls verteilt. Dabei wird der Zustand des Clusters an zufällig gewählte Knoten des Clusters propagiert. Knoten, welche noch nicht den aktuellen Zustand des Clusters kennen, werden jedoch bevorzugt ausgewählt.

Die Datenstruktur, welche den Zustand des Clusters beschreibt, konvergiert bei jedem Knoten und ist damit ein CRDT. Ein Knoten des Clusters kann erkennen, wenn seine Sicht des Zustands auch von allen anderen Knoten im Cluster gesehen wurde. Um das zu erreichen, wird eine Menge von Knoten (das sogenannte *seen-set*) verwendet. Das *seen-set* wird im Gossip verteilt. Jeder Knoten der den aktuellen Zustand gesehen hat, trägt sich dort ein. Sind alle Knoten im *seen-set* enthalten, ist der Zustand auf allen Knoten konvergiert (vgl. [1]).

Erkennen von ausgefallenen Knoten Akka Cluster verwendet zum Erkennen von ausgefallenen oder nicht erreichbaren Knoten eine Implementierung des φ-*Failure Detector* [11]. In der Implementierung wird jeder Knoten von bis zu fünf Knoten (voreingestellt) beobachtet. Sobald einer der Beobachter einen Ausfall eines zu beobachtenden Knotens feststellt, reicht dies aus, um den Knoten als nicht erreichbar zu identifizieren. Der Zustand des Clusters wird dadurch entsprechend geändert und per Gossip verbreitet. Ein Knoten kann wieder als erreichbar gelten, wenn alle Beobachter den ausgefallenen Knoten wieder als erreichbar sehen. Die Beobachter senden zum Erkennen eines Ausfalls *Heartbeats*-Nachrichten. Dabei werden die Ankunftszeiten dieser Nachrichten ausgewertet. Die Wahrscheinlichkeit, dass ein Knoten ausgefallen ist, wird durch einen Wert φ beschrieben.

Die Idee hinter dem φ-*Failure Detector* ist es, den aktuellen Zustand des verteilten Systems zu beschreiben. Der Wert für φ berechnet sich wie folgt:

$$\varphi = -log_{10}(1 - F(\mathit{timeSinceLastHearthbeat})) \tag{2.1}$$

F stellt in Formel (2.1) die Verteilungsfunktion der Normalverteilung mit Mean μ und Varianz σ^2 dar. Erreicht φ einen konfigurierbaren Grenzwert, so gilt ein Knoten als nicht erreichbar beziehungsweise ausgefallen (vgl. [1, 11]).

Das Gossip-Protokoll Das implementierte Gossip Protokoll ist eine Variation aus einem *Push/Pull* Gossip mit dem Ziel, die Menge an versendeten Daten zu reduzieren. Um das zu erreichen, werden nicht sofort Zustände per Gossip versendet, sondern nur Zeitstempel (Vector Clock). Daten werden dann bei Bedarf angefordert oder versendet. Zustände sind immer mit einem Zeitstempel versioniert. Es ergeben sich damit drei Möglichkeiten:

1. Der Empfänger hat einen neueren Zeitstempel und antwortet mit einem aktuellen Zustand.
2. Der Empfänger hat einen älteren Zeitstempel und sendet einen *Request* an den *Gossiper*. Der *Gossiper* sendet daraufhin seinen aktuellen Zustand.
3. Der empfangene Zeitstempel ist nebenläufig zum eigenen Zeitstempel und markiert damit einen Konflikt. Der Empfänger führt beide Zustände zusammen und versendet den dabei neu entstandenen Zustand an den *Gossiper*.

Akka sendet periodisch im Sekundentakt (konfigurierbar) Gossip nach dem oben genannten Verfahren. Sobald mehr als die Hälfte aller Knoten einen Zustand gesehen haben (Im *seen-set*

sind), so wird die Gossip-Rate verdreifacht. Durch die Erhöhung der Gossip-Rate wird die Konvergenzgeschwindigkeit erhöht (vgl. [1]).

Für die Serialisierung des Gossips wird Google Protobuf [12] verwendet. Protobuf wurde mit dem Ziel entworfen, eine möglichst schnelle und einfache Serialisierung von strukturierten Daten im Binärformat zu ermöglichen. Zusätzlich verwendet Akka GZIP [13] um die mit Protobuf serialisierten Daten zu komprimieren.

Zum Verwalten der Mitgliedschaften eines Akka-Cluster-Verbunds ist ein *Leader* notwendig. Jeder Knoten, der eine konvergierte Sicht auf den Zustand des Clusters hat, kann den Leader bestimmen. Dabei ist die Auswahl des Leaders auf jedem Knoten deterministisch und hat bei konvergentem Zustand das gleiche Ergebnis. Somit gibt es kein Wahlverfahren. Zum Bestimmen eines *Leaders* wird eine sortierte Liste mit allen Knoten des Clusters erstellt, welche im Zustand *Up* oder *Leaving* sind. Die entstehende Sortierung ist auf allen Knoten identisch. Der *Leader* ist der erste Knoten in der sortierten Liste (vgl. [1]).

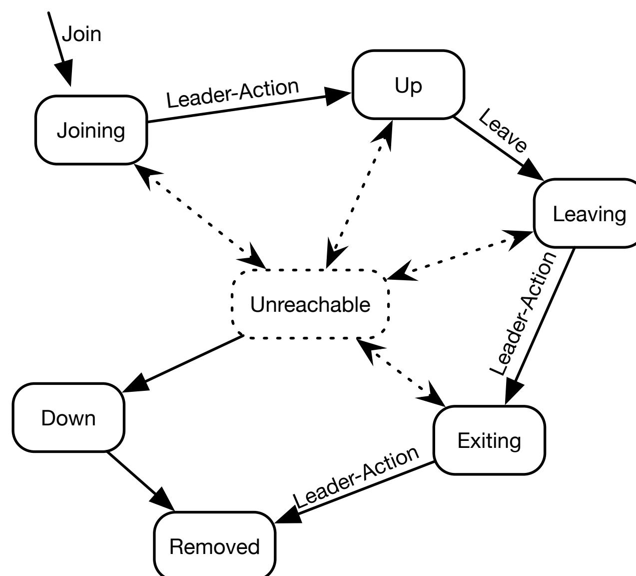

Abbildung 2.1: Zustände der Mitgliedschaft eines Knotens in Akka Cluster (vgl.[1, p. 338])

Abbildung 2.1 zeigt die Zustände der Mitgliedschaft eines Knotens innerhalb eines Akka Clusters. Ein neuer Knoten beginnt mit dem Zustand *Joining*. Nachdem alle anderen Knoten des Clusters den neuen Knoten im Zustand *Joining* gesehen haben und somit konvergiert sind, ändert der *Leader* des Clusters den Zustand von *Joining* zu *Up*. Ein normaler Ablauf ohne Ausfall ist wie folgt: $\xrightarrow{Join}$ *Joining* $\xrightarrow{LA}$ *Up* $\xrightarrow{Leave}$ *Leaving* $\xrightarrow{LA}$ *Exiting* $\xrightarrow{LA}$ *Removed*, wobei „LA“ für „Leader-Action“ steht. Die Transition der Zustände wird also vom *Leader*

vorgenommen. *Unreachable* ist kein Zustand, sondern ein Flag, der alte Zustand wird also beibehalten. Die Flag kann entfernt werden, wenn der *Failure Detector* einen Knoten wieder als erreichbar erkannt hat. Ist ein Knoten einmal im Zustand *Down*, so kann dieser Knoten nicht mehr aktiver Teil des Clusters werden. Sollte ein Knoten wieder erreichbar werden, so muss der Knoten dem Cluster neu beitreten (vgl. [1]).

2.1.3 Das C++ Actor Framework (CAF)

Das C++ Actor Framework (CAF) [14] ist eine Implementierung des Aktormodells auf Basis von C++11. Es wird seit 2011 an der HAW-Hamburg entwickelt. Es stellt eine Laufzeitumgebung zur Verfügung, welche Nachrichtenversand, lokales Scheduling von Aktoren auf Threads sowie Queue-Management bietet. Die Kommunikation zwischen lokalen und entfernten Aktoren wird durch die Laufzeit transparent bereitgestellt.

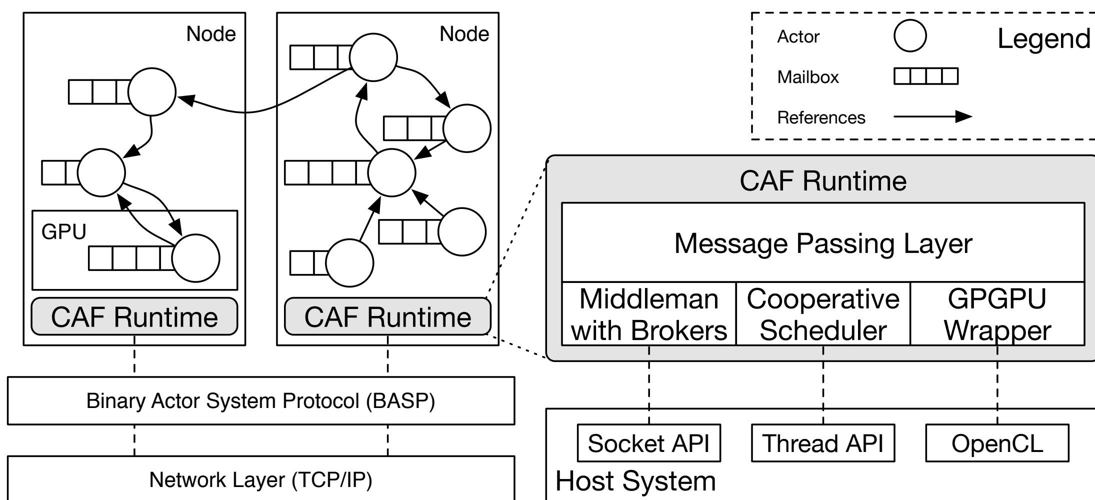

Abbildung 2.2: Architektur des C++ Actor Frameworks

Abbildung 2.2 zeigt die Architektur von CAF. Zu sehen sind zwei Knoten (Nodes), welche verschiedene Aktoren (Kreise) instanziieren. Aktoren haben jeweils eine Mailbox (Quadrate an den Kreisen). Der Kommunikationsfluss zwischen den Aktoren ist durch Pfeile gekennzeichnet. Es ist außerdem zu sehen, dass ein Aktor auf einer GPU ausgeführt wird.

Aktoren sind sich über ihre physische Verteilung nicht bewusst und formen ein logisches Kommunikationsnetzwerk, welches beliebig auf physische Knoten verteilt werden kann. Diese flexible Topologie wird durch das "Binary Actor System Procotol" (BASP) ermöglicht. Verteilte Laufzeitumgebungen kommunizieren über Middleman-Instanzen, welche das CAF-I/O-Modul

bereitstellt. Die Aufgabe der Middleman-Instanzen ist es, die Socket-API des Hostsystems zu abstrahieren und in eine nachrichtenbasierte Schnittstelle abzubilden. Pakete- und Byte-Streams der Netzwerkschicht werden als Nachrichten sogenannten Brokern zugestellt. Broker sind Aktoren, welche asynchrones I/O ausführen. Aktoren auf entfernten Laufzeitumgebungen werden durch den "BASP Broker" kontaktiert, welcher interne Aktor-Nachrichten an das Netzwerk weiterreicht. Das Scheduling von Aktoren auf Threads wird durch einen kooperativen Scheduler organisiert. Der Scheduler organisiert paralleles sowie faires Ausführen von Aktoren auf einem lokalen Knoten und abstrahiert dabei die Threading-API der C++ Standard-Bibliothek. Ein GPGPU-Wrapper des OpenCL-Moduls spricht heterogene Hardwarekomponenten wie Grafikkarten an. Es werden Aktoren erstellt, welche Aufgaben an einen OpenCL-Kernel weiterleiten. Nachrichten werden zur Laufzeit in OpenCL-kompatibel Datentypen gewandelt und umgekehrt.
Die folgenden Abschnitte erläutern Details der CAF-Laufzeit, welche verantwortlich für die Effizienz der Laufzeit sind. Weiterhin erklären sie Begriffe, welche im Terminus dieser Arbeit verwendet werden.

Actor-System CAF kapselt alle für die Laufzeit notwendigen Software-Entitäten in einem so genannten "Actor-System". Das Actor-System beinhaltet unter anderem alle geladenen Module (I/O, OpenCL) sowie Scheduler und das Typensystem. Eine lokale Anwendung kann mehrere Actor-Systeme beinhalten, welche dann über unabhängige Scheduler und Typensysteme verfügen. Es ist möglich, diese Actor-Systeme verschieden zu priorisieren, was es erlaubt, zeitkritische Aktoren im Actor-System mit einem aggressiv konfigurierten Scheduler zu betreiben. Andere "unwichtigere" Aktoren können in einem zweiten Actor-System mit einem passiver konfigurierten Scheduler ausgeführt werden.

Mailbox Jeder Aktor besitzt eine eigene Mailbox für den Eingang von Nachrichten. Eine effiziente Mailbox-Implementierung ist essentiell für ein Aktorsystem. Empfängt ein Aktor von mehreren Sendern gleichzeitig Nachrichten, so kann die Mailbox zu einem Flaschenhals werden und somit die gesamte Systemperformance negativ beeinflussen. Bei der Wahl des Mailbox-Algorithmus ist es wichtig, dass nebenläufige Lese- und Schreib-Zugriffe keinen signifikanten Synchronisationsaufwand verursachen. Ein Algorithmus, welcher auf klassischen Mutexen und Condition-Variables basiert, besitzt viel Synchronisationsaufwand, weshalb dieser Ansatz sich nicht für CAF eignet. CAF implementiert eine "Single-Reader-Many-Writer-Queue". Diese Implementierung erlaubt paralleles Schreiben in die Mailbox, wobei aber nur der Besitzer der Mailbox lesenden Zugriff hat. Die Implementierung basiert auf einem nicht blockierenden Stack,

welcher nach dem Last-In-First-Out (LIFO) Prinzip funktioniert. Alle Zugriffe auf diesen Stack sind mit einer compare-and-swap (CAS)-Operation realisiert. Um Nachrichten in der korrekten Reihenfolge First-In-First-Out (FIFO) verarbeiten zu können, ist der Stack mit einem internen Cache kombiniert, der Nachrichten in FIFO-Reihenfolge übersetzt und nur für den Besitzer der Mailbox verfügbar ist. Die Mailbox leidet nicht unter dem ABA-Problem bei parallelem Zugriff in CAS-basierten Algorithmen [15], da durch die Entkopplung von Cache und Stack jedes Element nur maximal einem Thread zur Zeit zur Verfügung steht. Beim Einfügen von neuen Elementen besitzt die Mailbox eine Komplexität von $\mathcal{O}(1)$. Bei der Entnahme von Elementen liegt die durchschnittliche Komplexität bei $\mathcal{O}(1)$, im schlechtesten Fall jedoch durch Umsortieren der Elemente von LIFO nach FIFO bei $\mathcal{O}(n)$, wobei n die Anzahl der Nachrichten auf dem Stack ist.

Middleman & Multiplexer CAF abstrahiert die Socket-API des Hostsystems durch das I/O-Modul. Pro Laufzeit-Instanz gibt es einen Middleman sowie einen Multiplexer. Die Aufgabe des Middlemans ist es, die Socket-API-Funktionen zu abstrahieren und beispielsweise Verbindungen zu anderen Knoten zu organisieren. Ein Multiplexer verwaltet und überwacht bestehende Sockets. Events auf verschiedenen Sockets können so effizient verarbeitet werden.
Verteilte CAF Knoten brauchen keine direkte Verbindung zu jedem Knoten. Kommt ein neuer Knoten hinzu, beispielsweise durch Erhalt eines Aktor-Handles aus einem fremden (noch) unbekannten Knoten, so wird die Verbindung über den Knoten aufgebaut, über welchen die Informationen gesendet wurden.

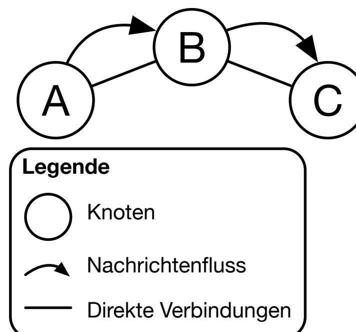

Abbildung 2.3: Routing von Nachrichten über andere CAF-Knoten

Abbildung 2.3 zeigt drei CAF-Knoten (Kreise), welche über direkte Verbindungen miteinander verbunden sind (Geraden). Die Knoten haben sich bereits alle kennengelernt und können miteinander kommunizieren. Sendet ein Aktor aus Knoten A nun eine Nachricht an einen Aktor

in Knoten C, so wird die Nachricht über Knoten B geroutet. Knoten A und B sowie Knoten B und C sind direkt miteinander verbunden. Da Knoten A und C aber auch kommunizieren, haben sich diese beiden Knoten über Knoten B kennengelernt.

2.2 Das CAP-Theorem

Verteilte Systeme können verschiedene Grade von drei Eigenschaften sicherstellen: (1) Konsistenz, (2) Verfügbarkeit und (3) Partitionstoleranz. Das CAP-Theorem [16] beschreibt, dass verteilte Systeme maximal zwei Eigenschaften zugleich garantieren können.

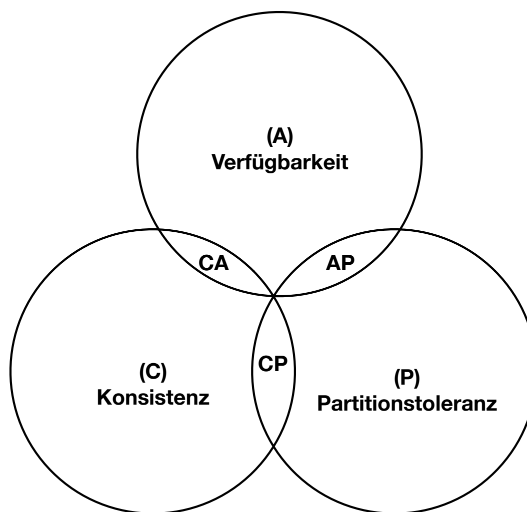

Abbildung 2.4: Das CAP-Theorem als Venn-Diagramm

Abbildung 2.4 zeigt die möglichen Eigenschaften, welche ein verteiltes System zur Zeit garantieren kann. Die Eigenschaft der Konsistenz sagt dabei, dass alle Knoten, welche auf geteilte Daten zugreifen, dieselbe Sicht auf die Daten haben. Systeme, die hohe Verfügbarkeit bieten, haben beim Schreiben auf den oder Lesen von den Daten eine geringe Latenz. Partitionstolerante Systeme können Dienste weiter anbieten und arbeiten, auch wenn Teile des Netzwerks ausfallen. Die Eigenschaften lassen sich verschieden kombinieren, dabei ist immer ein gradienter Anteil der Eigenschaft vorhanden.

Verfügbarkeit und Partitionstoleranz (AP) Systeme, die Verfügbarkeit und Partitionstoleranz sicherstellen, haben eine geringe Garantie der Konsistenz. Der Internet Domain Name Service (DNS) ist der verteilte Namensdienst des Internets und fällt in die Kategorie der AP-Systeme. DNS ist hoch verfügbar und ausfallsicher. Eine starke Konsistenz ist jedoch nicht

zugesichert, da es eine unbestimmt lange Zeit dauern kann, bis eine Änderung an alle DNS-Hierarchien propagiert wurde und damit alle Clients die gleichen Einträge sehen (vgl. [17, p. 233]).
Mit Cassandra [18] hat Facebook eine verteilte Datenbank implementiert, welche auch unter der Kategorie der AP-Systeme einzuordnen ist. Cassandra erlaubt es, Daten auf Hunderten von Knoten zu halten und trotzdem geringe Latenzen beim Lesen zu gewährleisten. Außerdem wird sichergestellt, dass selbst Millionen von schreibenden Änderungen effizient zwischen den Knoten propagiert werden. Dabei ist es möglich, dass Knoten geografisch weit entfernt liegen (vgl. [18]). Weitere Beispiele für AP-Systeme sind CouchDB [19], Riak [10] und Dynamo [9]. Systeme, welche auf Conflict-free Replicated Datatypes (CRDTs) setzen, fallen unter die Kategorie der AP-Systeme.

Konsistenz und Verfügbarkeit (CA) Klassische relationale Datenbanken (RDBMS) wie MySQL [20], Postgres [21] oder OracleSQL finden sich im CAP-Theorem bei den CA-Systemen wieder. Bei diesen Datenbanken ist eine konsistente Sicht auf Daten sowie eine geringe Latenz bei Schreibe- und Lese-Operationen wichtig. Replikation von Daten findet hier hauptsächlich zur Erhöhung der Verfügbarkeit statt.

Konsistenz und Partitionstoleranz (CP) CP-Systeme bieten ein hohes Maß an konsistenter Sicht und Partitionstoleranz. Ein Beispiel für diese Art von Systemen ist Hadoop HBase [22]. Weitere Beispiele für CP-Systeme sind Googles BigTable [23] sowie MongoDB [24] und Redis [25].

2.3 Logische Uhren

Verteilte Systeme benötigen oft Zeitstempel, um Zeitpunkte im System darzustellen. Dabei kann ein Zeitpunkt mit einem Datenstand, einer Operation, Events, oder anderen Dingen verknüpft werden. Das Problem an Uhren und Zeiten in verteilten Systemen ist, dass verschiedene Rechner keine eindeutig synchronisierte Uhr haben und sich diese auch nicht komplett synchronisieren lassen. Gerade wenn Rechner weit verteilt oder mit einem Netzwerk verbunden sind, welches nicht zuverlässig ist, können Uhren von Rechnern immer weiter auseinander laufen. Es ist ständig notwendig, die verschiedenen Uhren der Rechner im verteilten System zu synchronisieren, was gerade bei großen Systemen mit einer hohen Anzahl an Rechnern zu viel Aufwand führt.
In den meisten Fällen reicht es aus, wenn sich alle Rechner im Verbund über die Uhrzeit einig

sind. Das muss dabei nicht bedeuten, dass diese Uhrzeit auch die korrekte Uhrzeit ist. Es ist also nur wichtig, dass die interne Konsistenz der Uhren gegeben ist. Diese Uhren werden auch logische Uhren genannt (vgl. [17, p. 288]). Dieser Abschnitt stellt verschiedene Arten und Funktionsweisen von logischen Uhren vor.

2.3.1 Lamport-Uhr

Die Lamport-Uhr [2] ist die einfachste logische Uhr. Sie besteht aus einem einzelnen Zeitstempel-Wert. Mit dieser Uhr generierte Zeitstempel lassen sich mit Hilfe der „Happens-Before"-Relation auswerten. Die „Happens-Before"-Relation $a \rightarrow b$ wird gelesen als „a passiert vor b". Die Happens-Before Relation kann in zwei Situationen direkt beobachtet werden:

1. „Wenn a und b Ereignisse im selben Prozess sind, und a vor b auftritt, gilt $a \rightarrow b$." [17, p. 289]

2. „Wenn a das Ereignis darstellt, dass eine Nachricht von einem Prozess gesendet wird, und b ist das Ergebnis, dass die Nachricht von einem anderen Prozess empfangen wird, gilt ebenfalls $a \rightarrow b$. Eine Nachricht kann nicht empfangen werden, bevor sie gesendet wird, und auch nicht zum gleichen Zeitpunkt, zu dem sie gesendet wird, weil es eine endliche Zeit ungleich null dauert, bis sie ankommt." [17, p. 289]

Die „Happens-Before"-Relation ist transitiv. Das bedeutet, dass wenn $a \rightarrow b \wedge b \rightarrow c$ gilt, auch $a \rightarrow c$ gilt. Angenommen zwei Ereignisse x und y passieren unabhängig voneinander in verschiedenen Systemen, so kann weder $x \rightarrow y$ noch $y \rightarrow x$ gelten. In einer solchen Situation spricht man von nebenläufigen Ereignissen, da diese nicht in Relation gestellt werden können oder müssen (vgl. [17, p. 289]). Angenommen, Ereignisse a und b sind nebenläufig, so wird häufig $a||b$ angegeben (siehe Formel (2.2)).

$$a||b \equiv \neg(a \rightarrow b) \wedge \neg(b \rightarrow a) \tag{2.2}$$

Es ist möglich, zu jedem Ereignis einen Zeitstempel repräsentiert zu bekommen. Die Lamport-Uhr definiert eine Uhr-Bedingung („Clock Consistency Condition"):

$$\text{Für alle Events } a, b \text{ gilt}: \quad a \rightarrow b \implies \mathcal{C}(a) < \mathcal{C}(b)$$

Abbildung 2.5: „Clock Condition" (vgl. [2, p. 560])

Die in Abbildung 2.5 verwendete Notation von $\mathcal{C}(a)$ und $\mathcal{C}(b)$ beschreibt die Zeitpunkte, an welchen Event a beziehungsweise b passiert sind. Die Bedingung besagt, dass wenn die

„Happens-Before"-Relation für a und b gilt, dann ist Zeitpunkt $\mathcal{C}(a)$ auch kleiner als Zeitpunkt $\mathcal{C}(b)$.
Die Lamport-Uhr ordnet alle Ereignisse in einem verteilten System vollständig, dabei kann jedoch nichts über die Beziehung zwischen zwei Ereignissen a und b ausgesagt werden. Es werden lediglich die Zeitstempel $\mathcal{C}(a)$ und $\mathcal{C}(b)$ verglichen, was nicht impliziert, dass a vor b stattgefunden hat. Das Problem bei der Lamport-Uhr ist die Kausalität. Kausale Beziehungen können mit der Lamport-Uhr nicht abgebildet werden [17, p. 293]. Gerade in Replikations-Systemen ist es sinnvoll, eine kausale Beziehung zwischen Ereignissen feststellen zu können, da somit echte Konflikte zuverlässiger erkannt werden können.

2.3.2 Vektoruhr

Vektoruhren können kausale Zusammenhänge zwischen Ereignissen beschreiben. Dazu wird ein Vektor-Zeitstempel $VT(a)$ von Ereignis a verwendet. Der Vektor-Zeitstempel hat die Eigenschaft, dass sicher ist, dass Ereignis b kausal hinter a passiert ist, wenn $VT(a) < VT(b)$ gilt. Ein Vektor-Zeitstempel besteht aus einem Vektor V_i, welcher zu Prozess P_i gehört (vgl. [17, p. 293]). Jeder Prozess hat seinen eigenen Eintrag im Vektor-Zeitstempel. Es gelten folgende Eigenschaften eines Vektor-Zeitstempels:

1. „$V_i[i]$ ist die Anzahl der Ereignisse, die bisher in P_i aufgetreten sind." [17, p. 294]

2. „Wenn $V_i[j] = k$ gilt, erkennt P_i, dass auf P_j k Ereignisse aufgetreten sind." [17, p. 294]

Die erste Eigenschaft ist einfach zu gewährleisten. Dazu muss lediglich bei jedem neuen Ereignis eines Prozesses P_i der eigene Zähler von P_i inkrementiert werden. Die zweite Eigenschaft wird sichergestellt, indem beim Versenden von Ereignis m auch der Vektor-Zeitstempel des Ereignisses mitgesendet wird. So kann ein Empfänger darüber informiert werden, wie viele Ereignisse in P_i aufgetreten sind. Dabei ist jedoch wichtiger, dass der Zeitstempel von m die Information enthält, wie viele Ereignisse in anderen Prozessen vorausgegangen sind (vgl. [17, p. 294]).
Vektor-Uhren lassen sich somit gut verwenden, um kausale Zusammenhänge festzustellen. Kann für zwei Ereignisse a und b kein kausaler Zusammenhang festgestellt werden, so gilt $a||b$. Vektor-Uhren kommen oft in Systemen vor, welche Daten replizieren. In den Beispielen aus diesem Abschnitt wurde von Vektor-Zeitstempeln im Bezug auf Prozesse gesprochen. In verteilten Systemen zur Datenreplikation wird hier meist ein Replikat oder Knoten mit einem Vektor-Zeitstempel-Slot in Verbindung gebracht anstelle eines Prozesses.

Das Problem mit Vektor-Uhren Stellen wir uns ein verteiltes System vor, in welchem immer wieder Knoten den Verbund verlassen oder ihm beitreten. Jeder dieser neuen Knoten benötigt also einen eigenen Eintrag in der Vektoruhr. Eine große Anzahl an Knoten n führt zu mehreren Problemen. Ein Problem sind die Anforderungen an den Speicher, welcher erforderlich wird bei hohen n. Umso größer die Vektoruhr, desto mehr Daten müssen an andere Knoten versendet werden, was zu Netzwerk-Overhead führt. Außerdem ist die Auswertung der Ordnung von Events bei großen Vektoruhren ein Problem, da dadurch viel Overhead entsteht. Demnach skalieren Vektoruhren schlecht (vgl. [26]).
Oft kann nur schwer entschieden werden, ob ein Eintrag E_i aus einer Vektoruhr entfernt werden kann, da man nicht sicher feststellen kann, ob der Knoten K_i noch weiter Ereignisse generiert oder nun „fertig“ ist.
Eine Lösung dazu können Plausible Clocks sein, welche der folgende Abschnitt 2.3.3 beschreibt.

2.3.3 Plausible Clocks

Plausible Clocks [26] sind eine Klasse von Uhren, welche einen berechenbaren Grad an Ungenauigkeit zulassen und dabei konstante Größe bieten. Kausalität kann mit dieser Art von Uhren nicht komplett erfasst werden. Sie skalieren jedoch durch die konstante Größe besser in großen Systemen oder Systemen, welche unter hoher Fluktuation von Teilnehmern leiden.
Plausible Clocks können nicht alle Fälle von Nebenläufigkeit erkennen. Deshalb eignen sich diese Arten von Uhren nur für Systeme, in welchen die Ordnung durch die Uhr nur Einfluss auf die Performance und nicht auf die Korrektheit hat. In diesem Abschnitt ist eine Variation der Vektor-Uhr vorgestellt, welcher zur Klasse der Plausible Clocks gehört.

R-Entries-Vectorclock Die R-Entries-Vectorclock (REV) ist eine Abwandlung der Vektoruhr. Dabei besitzt nicht jeder Teilnehmer beziehungsweise Knoten der Vektoruhr einen eigenen Slot, sondern teilt sich diesen mit anderen. Angenommen es gibt N Teilnehmer an der Vektoruhr, und es gibt nur R Slots im Vektor (es gilt: $R < N$). Da nicht für jeden Teilnehmer einen Slot in der Vektoruhr zur verfügung steht, werden Teilnehmer auf einen Slot abgebildet. Die simpelste Abbildung ist ein Modulo-R-Mapping.
Die Funktionsweise der REV unterscheidet sich nur im Zugriff auf den Vektor von der Funktionsweise der Vektoruhr. Ein Beispiel: Jeder Teilnehmer braucht einen eindeutigen Bezeichner i, wobei $0 \leq i \leq N - 1$. Der Zugriff auf den zugrunde liegenden Vektor V wird beschrieben als $V[i \bmod R]$ (vgl. [26]).

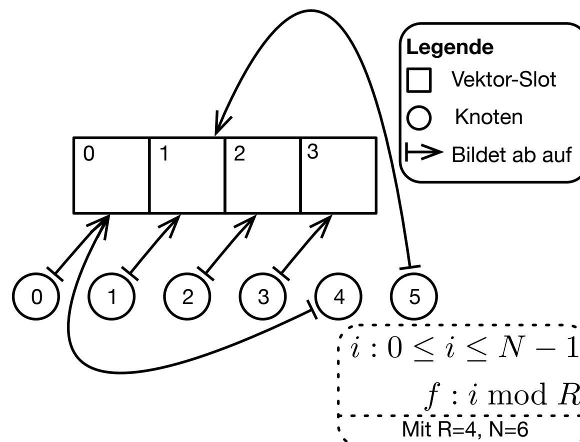

Abbildung 2.6: Mapping von Knoten auf Slots im Vektor

Abbildung 2.6 beschreibt das Mapping von Knoten auf Slots in der REV. Zu sehen sind nummerierte Vektor-Slots (Vierecke) sowie nummerierte Knoten (Kreise). Der Pfeil beschreibt die Abbildung von Knoten auf einen Vektor-Slot. Dabei wird die Anzahl der Slots im Vektor mit $R := 4$ und die Anzahl der Knoten $N := 6$ verwendet. Die verwendete Abbildung ist dabei eine einfache Modulo-Operation. Es ist zu erkennen, dass Vektor-Slot null von zwei Knoten verwendet wird (0 und 4). Der Vektor-Slot eins wird ebenfalls doppelt verwendet (Knoten 1 und 5). Vektor-Slots zwei und drei sind allein an Knoten 2 beziehungsweise 3 gebunden.

2.4 Datenreplikation

In verteilten Systemen wird die Replikation von Daten dazu verwendet, die Zuverlässigkeit zu steigern oder die Leistung zu verbessern. Eines der größten Probleme dabei ist es, die Replikate konsistent zu halten. Bei Änderung einer Kopie muss sichergestellt sein, dass alle anderen Kopien ebenfalls diese Änderung bekommen (vgl. [17, p. 333]). Bei manchen Änderungen ist es sogar notwendig, dass diese in bestimmter Reihenfolge eingespielt werden, da sonst verschiedene Sichten entstehen und die Replikate somit nicht mehr den gleichen Zustand haben.

Zuverlässigkeit kann durch das Replizieren von Daten auf mehrere Rechner im verteilten System erreicht werden. Bei Ausfall einzelner Teile des Netzwerks oder ganzer Systeme kann ein Verlust von Daten verhindert werden. Außerdem kann durch lokale Haltung von Daten die Zugriffszeit reduziert und dadurch die Leistung gesteigert werden.

Um Daten in einem verteilten System zu replizieren gibt es verschiedene Ansätze. Klassifikationen können sein, ob etwa Daten an allen Replikaten geändert werden können, oder nur manche Replikate im System schreibende Rechte gewähren. Außerdem hat das Kommunikationsmodell (synchron/asynchron) Auswirkungen auf die Replikation. Des Weiteren gibt es Unterschiede darin, welche Art von Daten zwischen Knoten propagiert wird (Operationen oder Zustände). Sobald mehrere Knoten mit geteilten Daten arbeiten, kann es zu Konflikten kommen, deshalb sind Verfahren zum Umgang mit Konflikten erläutert. Der folgende Abschnitt verschafft einen Überblick über die gebräuchlichen Verfahren zur Replikation von Daten in verteilten Systemen.

2.4.1 Symmetrische Replikation

Bei symmetrischer Replikation wird ein Update auf allen Replikaten des Replikationsverbundes durchgeführt, es wird auch von „Update anywhere“ oder „Update everywhere“ gesprochen (vgl. [27, p. 174]). Datenbank-Systeme, welche nach diesem Prinzip arbeiten, zeichnen sich dadurch aus, dass jeder Knoten auch bei Ausfall des Netzwerks oder anderer Knoten funktionsfähig ist. Da jeder Knoten Änderungen an seinen lokalen Replikaten vornehmen kann und diese in Konflikt stehen können, ist es bei Systemen, die nach diesem Muster arbeiten, vonnöten, einen Algorithmus zur Konfliktauflösung (Conflict-Resolution) zu verwenden. Diese Art von Systemen lassen sich zu den AP-Systemen des CAP-Theorems zählen.

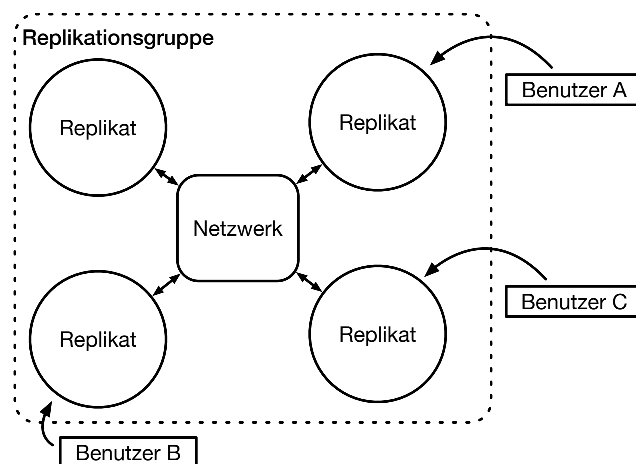

Abbildung 2.7: „Update anywhere“, jedes Replikat kann geändert werden

Abbildung 2.7 zeigt vier Replikate in einer Replikationsgruppe. Mehrere Benutzer (A, B, C) nehmen Änderungen an einem ihrer lokalen Replikate vor. Dabei werden Änderungen auch

auf anderen Replikaten vorgenommen. Änderungen, die global sichtbar werden, sind also auf jedem Replikat möglich. Bei Wiedereintritt eines Knotens nach Ausfall muss ein Verfahren verwendet werden, was eventuelle Konflikte zwischen den Replikaten auflösen kann.

2.4.2 Asymmetrische Replikation

Bei der asymmetrischen Replikation gibt es einen Master oder einen Verbund von Rechnern, welche auch als „Update-Master“ bekannt sind. Bei einer Änderung wird nur die „Primary-Copy“ geändert. Bei erfolgreicher Änderung der Primary-Copy verteilt sich das Update an Rechner im Replikationsverbund. Alle Änderungen finden also zuerst auf der Primary-Copy statt.

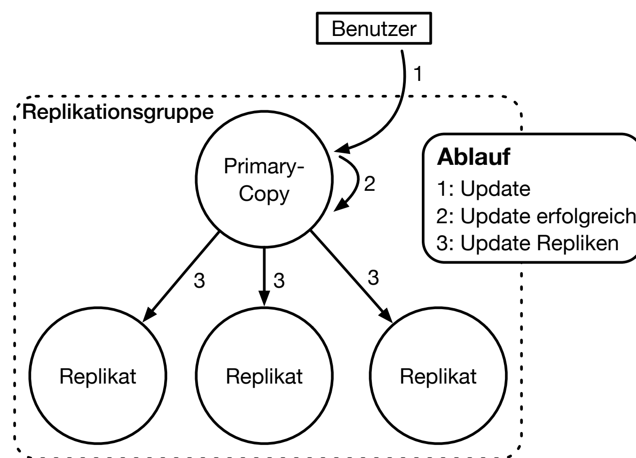

Abbildung 2.8: Beispiel des Update-Ablaufs bei Primary-Copy

Abbildung 2.8 zeigt eine Primary-Copy sowie drei Replikate dieses Dateneintrags, welche zusammen eine Replikationsgruppe bilden. In diesem Beispiel ändert ein Nutzer die Daten an der Primary-Copy (1). Sollte die Änderung an der Primary-Copy (2) erfolgreich sein, wird die Änderung auch an den Repliken der Gruppe ausgeführt (3).
Fällt ein Replikat aus und wird später wieder verfügbar, fordert es eine Kopie der Primary-Copy an.

2.4.3 Synchrone Replikation

Synchrone Replikation wird auch „Pessemistic-“ oder „Eager-“ Replikation genannt. Dabei werden Operationen auf allen Repliken des Replikationsverbundes ausgeführt. Schlägt eine Operation lokal auf einem Replikat fehl, so scheitert die Operation im globalen Kontext.

Replikate sind damit immer exakt gleich.(vgl. [27]). Synchrone Replikation lässt sich nicht gut in Umgebungen mit unzuverlässigen Netzwerken oder gar Netzwerk-Ausfällen betreiben. Als Beispiel für eine solche Umgebung können Mobil-Netze genannt werden.

Read-One-Write-All (ROWA) Read-One-Write-All-Verfahren werden verwendet, um die Performance und Verfügbarkeit bei lesenden Operationen zu erhöhen (vgl. [28]). Eine Änderung auf einem Replikat wird auf allen Replikaten durchgeführt. Um dies zu erreichen, müssen alle Replikate für die Operation gesperrt werden. Sind alle Replikate gesperrt, kann die Operation ausgeführt werden. ROWA-Verfahren sollten deshalb nur in Systemen verwendet werden, in denen mehrheitlich lesender Zugriff auf Daten benötigt wird, da schreibende Zugriffe ein hohes Maß an Synchronisation benötigen.
Ein verteiltes System, welches bei der Replikation von Daten auf ein ROWA-Verfahren setzt, wird am besten so strukturiert, dass Replikate nah an den Clients verfügbar sind. Ein lesender Zugriff auf Daten kann bei einem solchen System auf jedem Replikat ausgeführt werden. Dadurch ist eine hohe Verfügbarkeit zugesichert. Im besten Fall sind die Zugriffszeiten bei Lese-Operationen kurz, da ein beliebiges und damit im besten Fall nahes Replikat zum Lesen verwendet werden kann. Ein schreibender Zugriff auf eines der Replikate wird jedoch auf allen Replikaten des Replikationsverbundes ausgeführt. Schreibende Zugriffe auf die Replikate sind dadurch teuer und verringern die Verfügbarkeit, da andere Operationen erst einmal blockiert sind. Eines der Hauptprobleme bei ROWA-Verfahren ist somit die gleichzeitige Verfügbarkeit aller Replikate (vgl. [28]).

Read-One-Write-All-Available (ROWAA) Das ROWAA-Verfahren ist eine Optimierung des ROWA. Lesende Zugriffe können wie bei ROWA auf jedem Replikat durchgeführt werden. Schreibende Zugriffe werden jedoch nur an verfügbare Replikate propagiert. Damit löst ROWAA das Problem der Verfügbarkeit, jedoch kann es sein, dass Daten nicht mehr konsistent sind. Sollte ein Replikat, welches nicht aktualisiert wurde, wieder verfügbar werden („recovern“), so wird jede zukünftige Lese-Operation auf dieses Replikat einen falschen Wert wiedergeben. Die Daten sind damit nicht mehr konsistent (vgl. [29]). Es muss zusätzlicher Aufwand betrieben werden, um alle Replikate wieder in den selben Zustand zu versetzen.

2.4.4 Asynchrone Replikation

Eine Alternative zur synchronen Replikation bietet die asynchrone Replikation, welche auch „Lazy-Replication“ oder „Optimistic-Replication“ genannt wird. Bei dieser Art von Replikation werden Updates asynchron an andere Replikate des Replikationsverbunds verteilt. Manche

verteilte Systeme mit einem zuverlässigen Netzwerk verwenden diese Art der Replikation, um Zugriffszeiten zu verringern (vgl. [27]). Das Hauptproblem in asynchronen Verfahren ist die Konfliktauflösung. Asynchrone Replikationsverfahren sind entweder zustandsbasiert oder operationsbasiert. Abschnitt 2.4.6 geht auf Zustands- und Operationstransfer ein.

2.4.5 Multi- & Single- Master Systeme

Single-Master Systeme bestehen aus Knoten, in dem nur ein Knoten berechtigt ist, Änderungen an Daten vorzunehmen, welche von dem Master dann zu den Replikaten oder auch „Slaves" gesendet werden. Da nur der Master schreibenden Zugriff hat, ist es für die anderen Teilnehmer im Replikationsverbund nicht möglich, Änderungen an den Daten vorzunehmen. Die Slaves können deshalb auch Caches genannt werden, bei schreib-intensiven Anwendungen ist die Verfügbarkeit jedoch eingeschränkt (vgl. [3, p. 9,10]).
Multi-Master-Systeme bestehen aus mehreren (mindestens zwei) Knoten mit der Berechtigung zum Ändern von Daten. Die Verteilung von Updates geschieht im Hintergrund und kann unabhängig voneinander geschehen. Die Komplexität von Multi-Master Systemen ist höher als bei Single-Master Systemen, da meist ein Operation-Scheduling und/oder eine Konfliktbehandlung benötigt wird. Ein weiteres Problem bei Multi-Master-Systemen ist, dass sich mit steigender Komplexität auch die Anzahl an potenziell zu lösenden Konflikten erhöht (vgl. [3, p. 9,10]). In dem Fall, dass mehrere Master ($M > 2$) in einem Multi-Master System in einem konstanten Intervall Updates erstellen und verschicken, würden bei einer naiven Implementierung Konflikte in der Größenordnung $\mathcal{O}(M^2)$ entstehen, welche aufgelöst werden müssen (vgl. [27]).

2.4.6 Zustandstransfer & Operationstransfer

Eine der grundlegenden Entscheidungen bei dem Design von verteilten Systemen, welche Daten replizieren, ist die Art des Transfers. Dabei ist es möglich, Operationen oder Zustände zwischen den Teilnehmern des Systems auszutauschen. Systeme, welche Zustände transferieren, sind in ihren Operationen auf replizierten Objekten limitiert. Entweder kann ein Objekt gelesen oder das komplette Objekt überschrieben werden. Operationsbasierte Systeme sind in der Lage, den Zugriff semantisch genauer zu beschreiben (vgl. [3, p. 10]). Stellen wir uns eine Datei vor, welche in einem verteilten Datensystem geändert wird. Bei einem Zustandstransfer-Systems wird nun die gesamte Datei neu übertragen. Bei Operationstransfer-Systemen wird hingegen nur eine Operation übertragen, welche beim Ziel eingespielt werden kann (vgl. [3, p. 10]).
Zustandstransfer-Systeme sind generell einfacher, da zum Erhalten der Konsistenz lediglich der neueste Datensatz transferiert werden muss. Bei Operationstransfer-Systemen ist das

Übertragen wesentlich aufwendiger, da eine History von Operationen geführt werden muss. Außerdem kann es sein, dass die Reihenfolge von Operationen korrekt sein muss. Dies führt zu zusätzlichem Aufwand, jedoch können Operationstransfer-Systeme im Bezug zur Netzwerklast effizienter sein, gerade wenn das replizierte Objekt komplex ist (vgl. [3, p. 10]). Systeme wie CVS (Concurrent Versions System), welche δ-Zustände transferieren, befinden sich zwischen operationsbasierten und zustandsbasierten Systemen. Konfliktbehebung in Operationstransfer-Systemen ist flexibler, da hier auf semantischer Basis gearbeitet werden kann, wie beispielsweise eine Bibliotheksdatenbank, in welcher ein Autor von zwei verschiedenen Büchern geändert wird. Schwieriger wird es hier, wenn ein System die komplette Datenbank überträgt, wie es bei Zustandstransfer-Systemen der Fall wäre (vgl. [3, p. 10]).

2.4.7 Scheduling von Operationen

In verteilten Systemen und Anwendungen, welche Daten replizieren, kann es vorkommen, dass Operationen „gescheduled“ werden müssen. Das bedeutet auch, dass sie in eine Ordnung gebracht werden, damit alle Replikate den selben Zustand erreichen. Man kann diese Scheduling-Verfahren in zwei Kategorien unterteilen, (1) Syntaktisches Scheduling und (2) Semantisches Scheduling. Beim syntaktischen Scheduling werden Operationen danach bewertet, wann, wo oder von wem diese getätigt wurden. Generell sind Systeme, welche syntaktisches Scheduling durchführen, einfacher, generieren dabei aber mehr Konflikte (vgl. [3, p. 11]).
Semantisches Scheduling kann Operationen nach ihrer Art sortieren. Systeme, welche ein semantisches Scheduling durchführen, gelten als flexibler und können die Anzahl an Konflikten reduzieren. Der Preis dafür ist jedoch, dass das Scheduling wesentlich komplexer ist und mit in die Anwendungslogik greift. Der Scheduling-Algorithmus muss also semantisch den Prozess der Anwendungslogik verstehen (vgl. [3, p. 11]).
Als Beispiel dient folgendes Szenario: In einem Pool von Ressourcen, sind drei Operationen nebenläufig. Operation eins und zwei entnehmen jeweils eine Ressource, Operation drei gibt eine zurück. In einem syntaktischen System, welche die Operationen nun in der Reihenfolge der Nummerierung durchführt (also 1, 2, 3), können nicht alle Operationen erfüllt werden, wenn Operation eins die letzte Ressource des Pools entnimmt. Ein System, welches semantisches Scheduling verwendet, kann die Operationen anders schedulen, sodass alle Operationen erfolgreich sind (1, 3, 2) (vgl. [3, p. 11]).

2.4.8 Verfahren zum Auflösen, Reduzieren und Vermeiden von Konflikten

In verteilten Systemen, welche Daten replizieren, kann es zu Konflikten kommen. Gerade in Systemen, in denen von verschiedenen Orten Daten geändert werden können (beispielsweise Multi-Master-Systeme), ist es möglich, dass Änderungen nebenläufig passieren oder sich gegenseitig widersprechende Ergebnisse liefern. Um die Sicht auf Daten auf jedem Replikat konsistent zu halten, werden bei Konflikten Algorithmen verwendet, welche diese auflösen, die Wahrscheinlichkeit dieser verringern oder diese ganz verhindern.
Es gibt verschiedene Verfahren zur Konfliktlösung, welche meist anwendungsspezifisch implementiert werden. Simple Verfahren sind beispielsweise Zeitstempel-Verfahren, wobei es die Möglichkeit gibt, entweder den kleinsten oder den größten Zeitstempel gewinnen zu lassen. Außerdem gibt es Verfahren, in welchen immer eine bestimmte Kopie gewinnt (beispielsweise „Mastercopy Wins“). Sollten die im System oder der Anwendung vorhandenen Verfahren zur Konfliktbehebung nicht erfolgreich oder anwendbar sein, so gibt es die Möglichkeit, den Benutzer den Konflikt per Hand lösen zu lassen („Notice User“). Ein Beispiel für ein solches Verfahren ist in Git [30] verwendet. Wenn alle Verfahren zum Lösen von Konflikten fehlschlagen, muss der Benutzer den Konflikt manuell lösen. Im Folgenden werden mehrere Verfahren zum Behandeln, Reduzieren und Lösen von Konflikten vorgestellt.

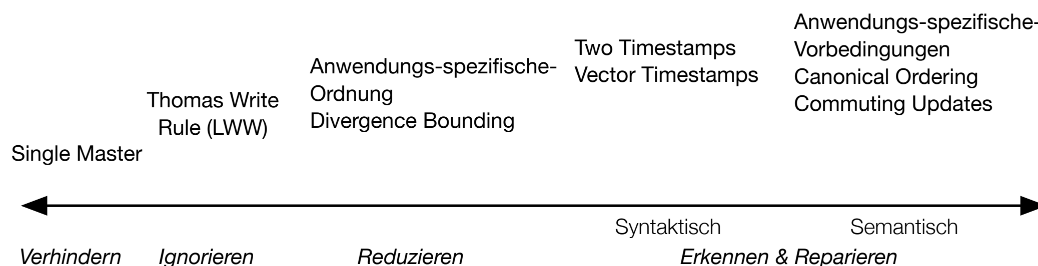

Abbildung 2.9: Möglichkeiten im Umgang mit Konflikten (vgl. [3, p. 11])

Abbildung 2.9 zeigt die verschiedenen Möglichkeiten, wie in verteilten Systemen mit Konflikten umgegangen werden kann. Der Spielraum reicht von Verbieten über Ignorieren und Reduzieren & Erkennen bis zu Reparieren. Die Abbildung zeigt verschiedene Verfahren und Techniken hierzu, welche im Folgenden erklärt sind.

Single Master Single-Master-Systeme verhindern Konflikte, da Operationen, welche Zustände ändern, nur auf einem System zugelassen sind. Konflikte können so nicht aufkommen, da nur der Master den Zustand ändern darf, welcher dann von den Slave-Replikaten übernommen wird. Single-Master-Systeme sind bei Verbieten von Konflikten in Abbildung 2.9 einsortiert.

Thomas Write Rule Bei der Thomas Write Rule handelt es sich um ein „Last-Writer-Wins“-Verfahren. Dabei wird mit dem Replikats-Zustand auch ein Zeitstempel gespeichert, welcher die Aktualität („newness“) des Zustands beschreibt. Empfängt das Replikat i nun ein Update von Replikat j, wird geprüft ob i's Zeitstempel kleiner als j's Zeitstempel ist. Wenn j den größeren (neueren) Zeitstempel hat, übernimmt i den Zustand samt Zeitstempel von j (vgl. [3, p. 22]). Die Thomas Write Rule wird meistens in Zustandstransfer-Systemen verwendet, da Zustände komplett übernommen werden. Die Thomas-Write-Rule ist unter Ignorieren von Konflikten in Abbildung 2.9 zu finden.

Divergence Bounding Verfahren, welche „Divergence Bounding“ verwenden, verhindern, dass Replikate zu weit auseinander laufen und damit ein zu hohes Maß an Inkonsistenz aufweisen. Eine gewisse Inkonsistenz zwischen den Replikaten ist damit zugelassen. In Abbildung 2.9 ist das „Divergence Bounding“ unter Reduzieren von Konflikten zu finden. Ein einfaches Verfahren ist das sogenannte „Order Bounding“. Dabei wird die Anzahl an Operationen, welche noch nicht im System propagiert sind, limitiert. Ein anderes Beispiel ist TACT [31], ein Multi-Master-Operationstransfer-System. Sollte hierbei die Differenz zwischen akzeptierten Updates und noch ausstehenden Updates zu groß werden, so werden keine neuen Updates angenommen (vgl. [3, p. 35]). TACT stellt außerdem „Numeric Bounding“ bereit, wobei die Differenz von Replikat-Werten begrenzt wird (vgl. [3, p. 35]). Mit „Numeric Bounding“ lässt sich die tatsächliche Divergenz von Replikaten besser begrenzen als bei „Order Bounding“, dafür ist es aber komplexer (vgl. [3, p. 35]). „Numeric Bounding“ ist außerdem stärker an die Anwendungslogik gekoppelt, da Wissen über die Semantik vorliegen muss.

Two Timestamps Der Two-Timestamps-Algorithmus ist eine Erweiterung der Thomas Write Rule und damit auch ein Last-Writer-Wins-Verfahren. Bei Two Timestamps gibt es zusätzlich zur „newness“ einen zweiten Zeitstempel (*previous*), welcher beschreibt, wann das letzte Update durch Synchronisation übernommen wurde. Ein Konflikt wird dadurch erkannt, dass sich die *previous* Zeitstempel von zwei Knoten, welche sich synchronisieren, unterscheiden. Two-Timestamps-Algorithmen kommen auch in Operationstransfer-Systemen vor, um Konflikte zu erkennen. Bei mehr als zwei Knoten steigt dabei die Chance auf fehlerhaftes Erkennen von Konflikten (vgl. [3, p. 24]). Two Timestamps findet sich in Abbildung 2.9 unter Erkennen und Reparieren. Dabei kann ein Konflikt nur mit syntaktischen Mitteln erkannt werden.

Vector Timestamps Verfahren, welche auf Vektor-Zeitstempel setzen, finden sich in Abbildung 2.9 unter Erkennen und Reparieren. Dabei werden Vektor-Uhren verwendet, welche Kolli-

sionen beziehungsweise Nebenläufigkeit erfassen können. Vektor-Uhren sind in Abschnitt 2.3.2 genauer erläutert.

Canonical Ordering Bei „Canonical Ordering“-Verfahren werden alle möglichen Kombinationen von Operationen miteinander verknüpft. Es wird dabei festgelegt, in welcher Reihenfolge Operationen ausgeführt werden dürfen, ähnlich der Operatorrangfolge aus Programmiersprachen oder der „Punkt-vor-Strich“-Rechnung. Ein Beispiel für dieses Verfahren ist die Arbeit von Ramsey und Csirmaz et al. [32], welche dieses Verfahren zur Beschreibung eines limitierten verteilten Datensystems anwendet. In dem formalen Datensystem können auf allen Replikaten Änderungen gemacht werden, jedoch sollen final alle Zustände der Replikate gleich sein. Das Datensystem unterstützt *create*, *remove* und *edit*, nicht jedoch *move\rename* da dies die Komplexität zu sehr erhöhen würde (vgl. [3, p. 17]). Es handelt sich bei diesem Verfahren um eine Operations-Scheduling-Technik, welche semantisch arbeitet (siehe Abbildung 2.9).

Commuting Updates Ein weiteres Operations-Scheduling-Verfahren ist das Ausnutzen der Kommutativität von Operationen. Dabei handelt es sich wieder um ein semantisches Verfahren (siehe 2.9). Sind zwei Operationen α und β kommutativ, so können diese in beliebiger Reihenfolge ausgeführt werden, auch wenn eine Happens-Before-Relation verletzt werden würde (Abschnitt 2.3.1). Durch ein solches Verfahren lässt sich die Anzahl an Rollbacks reduzieren, da dieser Konflikt einfach zu beheben ist (vgl. [3, p. 17]).

2.5 Konsistenzmodelle

Konsistenzmodelle beschreiben einen Vertrag zwischen Prozess und Datenspeicher. Der Datenspeicher verspricht, korrekt zu arbeiten, wenn sich der Prozess an gewisse Regeln hält. Ein Prozess hat normalerweise die Annahme, dass eine Lese-Operation auf Speicher die letzte Schreibe-Operation reflektiert. In verteilten Systemen, in welchen es keine globale Uhr gibt, ist es oft schwierig zu definieren, welche Schreiboperation die letzte war. Alternativ lassen sich andere Definitionen finden, welche zu verschiedenen Konsistenzmodellen führen (vgl. [17, p. 340]).

2.5.1 Starke Konsistenzen

Starke oder strenge Konsistenzen sind Modelle mit dem höchsten Konsistenzanspruch. Diese sind durch folgende Bedingung definiert:

„Jede Leseoperation für ein Datenelement x gibt einen Wert zurück, der dem Ergebnis der letzten Schreibe-Operation für x entspricht.“ [17, p. 341]

Diese Definition ist natürlich, jedoch wird durch diese Definition die Existenz einer absoluten globalen Zeit eingefordert, welche in nebenläufigen Systemen oder verteilten Systemen nicht existiert. Die starke Konsistenz kann schnell gebrochen werden. Stellen wir uns zwei Maschinen A und B vor, welche per Netzwerk verbunden sind. Zum Zeitpunkt T_1 versucht ein Prozess auf Maschine A den Wert x zu lesen, wobei x nur auf Maschine B gespeichert ist. Nun versucht ein Prozess auf Maschine B zum Zeitpunkt T_2 den Wert x zu schreiben. Das Problem ist dabei jedoch, sollte $T_2 - T_1 \approx 1ns$ sein und die Maschinen ca. drei Meter voneinander entfernt stehen, dann müsste das Signal schneller als Lichtgeschwindigkeit übertragen werden. Für einen Datenspeicher, welcher streng konsistent ist, muss also gelten, dass alle Schreibe-Operationen unmittelbar für alle Lese-Operationen sichtbar sind und eine globale absolute Zeitreihenfolge herrscht (vgl. [17, p. 340-342]).

2.5.2 (Strong) Eventual Consistency

Eventual Consistency ist ein schwaches Konsistenzmodell. Es besagt, dass alle Replikate auf den selben Zustand des letzten Updates konvergieren, wenn lange genug keine Änderung vorgenommen wurde. Wenn keine Fehler im verteilten System auftreten, lässt sich das maximale Maß an maximaler Inkonsistenz anhand von Faktoren wie Round-Trip-Time (RTT), Last auf den Knoten oder Anzahl an Replikaten bestimmen (vgl. [33]).
Strong Eventual Consistency (SEC) ist eine Erweiterung von Eventual Consistency. Dabei gilt zusätzlich, dass Knoten, die dieselben ungeordneten Updates erhalten haben, in demselben Zustand sind. Die Reihenfolge der Verarbeitung der Updates spielt dabei keine Rolle. Die in dieser Arbeit behandelten CRDTs sichern SEC zu.

3 Conflict-free Replicated Datatypes (CRDTs)

Conflict-free Replicated Datatyped (CRDTs) [34, 4] sind Datentypen, welche sich konfliktfrei replizieren lassen. Operationen, welche von CRDTs unterstützt werden, unterliegen generell Einschränkungen, damit eine Konfliktfreiheit garantiert werden kann. Es gibt einfache CRDTs, welche sich wie Zähler, Container oder Register verhalten. Es lassen sich auch komplexere Strukturen wie zum Beispiel Bäume und Graphen mithilfe von CRDTs darstellen. Bekannte CRDTs werden in ihrer Semantik und Funktionsweise in Abschnitt 3.5 erläutert. CRDTs unterliegen dem Modell der asynchronen Objekt-Replikation und lassen sich in zwei verschiedene Klassen unterteilen: (1) Operationsbasiert und (2) Zustandsbasiert. Im Folgenden werden beide Klassen vorgestellt.

3.1 Asynchrone Objekt-Replikation

Es wird angenommen, dass ein System zu Grunde liegt, welches anfallende Updates an alle anderen Replikate im Replikationsverbund verteilt. Nicht weiter spezifizierte Clients verwenden dabei ein Replikat ihrer Wahl, welches das sogenannte *Source*-Replikat ist. Eine Änderung läuft dabei in zwei Phasen ab.
Die erste Phase „at-source" führt den Aufruf auf dem *Source*-Replikat aus, wenn datentypspezifische Vorbedingungen erfüllt sind.
Die zweite Phase ist die „at-downstream"-Phase, dabei werden Änderungen an allen anderen Replikaten des Replikationsverbundes eingespielt.
Y.Saito und M.Shapiro unterscheiden dabei zwischen zwei Ansätzen der *downstream*-Phase [3]. Der erste Ansatz ist eine zustandsbasierte Herangehensweise, wobei der neue Zustand des *Source*-Replikats an die anderen Replikate im Verbund verteilt wird. Die Änderung wird mit einem Zusammenführen der Zustände in den Replikaten sichtbar.
Die zweite Herangehensweise ist die operationsbasierte. Dabei hat die *at-source*-Phase keine Effekte, die Änderung wird direkt an alle Replikate des Verbunds inklusive des *source*-Replikats verteilt. In der *downstream*-Phase wird nun bei jedem Replikat die datentypspezifische Vorbe-

dingung der *downstream*-Phase geprüft. Ist die Vorbedingung erfüllt, wird die Änderung am jeweiligen Replikat vorgenommen [4, p. 5].

3.2 Zustandsbasiert (CvRDT)

Zustandsbasierte CRDTs werden „State-based Convergent Replicated Data Type“ (CvRDT) genannt. Die Grundlage von CvRDTs sind Halbverbände. Halbverbände sind spezielle Halbgruppen, welche nicht nur assoziativ und kommutativ, sondern auch idempotent sind. Definieren wir zuerst eine Halbgruppe $(S, \bullet)$, welche aus einer Menge S und einer zweistelligen Verknüpfung $\bullet$ besteht:

$$\begin{aligned} \bullet : S \times S \to S \\ (a, b) \mapsto a \bullet b \end{aligned} \tag{3.1}$$

Die zweistellige Verknüpfung $\bullet$ einer Halbgruppe ist assoziativ, da $\forall a, b, c \in S : a \bullet (b \bullet c) = (a \bullet b) \bullet c$. Eine Halbgruppe ist kommutativ oder abelsch, wenn für die Verknüpfung $\bullet$ gilt: $\forall a, b \in S : a \bullet b = b \bullet a$ [35, p. 85]. Die Idempotenz gilt, wenn $\forall a \in S : a \bullet a = a$, dabei spricht man von einem Halbverband. Zusätzlich lässt sich auf einen Halbverband S eine Halbordnung definieren:

$$(S, \leq) \tag{3.2}$$

Für Halbverbände gilt [35, p. 14,15,17]:

1. $\forall a \in S : a \leq a$ (*Reflexivität*)
2. $\forall a, b \in S : a \leq b \wedge b \leq a \implies a = b$ (*Antisymmetrie*)
3. $\forall a, b, c \in S : a \leq b \wedge b \leq c \implies a \leq c$ (*Transitivität*)

Nun betrachten wir $S' \subset S$. S' heißt nach oben beschränkt, wenn gilt: $\exists x \in S, \forall y \in S' : x \geq y$. Das Element x heißt dann obere Schranke von S'. $s \in S$ heißt kleinste obere Schranke von S', wenn für jede andere obere Schranke x von S' gilt: $s \leq x$. Das Element s heißt *Supremum* von S' [35, p. 255].
Die zweistellige Verknüpfung $\bullet$ unseres Halbverbandes S wird auch „Join“ genannt, für die Elemente $a, b \in S$ lässt sich die kleinste obere Schranke (*Supremum*) feststellen:

$$a \bullet b = sup\{a, b\} \tag{3.3}$$

Weiterführend wird in dieser Arbeit die Notation $a \sqcup b$ für *sup*$\{a, b\}$ verwendet, da diese von Shapiro et. al. [4] bezüglich CRDTs ebenfalls verwendet wird.

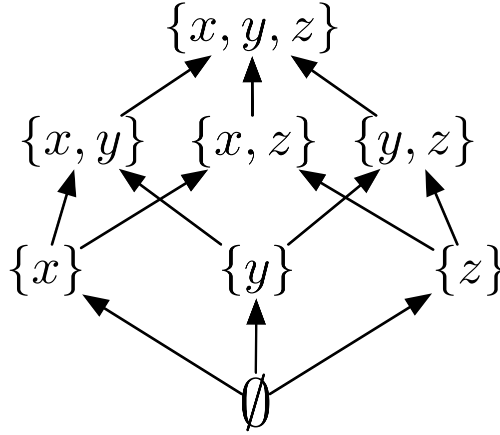

Abbildung 3.1: Hasse-Diagramm eines Join-Halbverbands

Abbildung 3.1 zeigt ein Hasse-Diagramm eines *Join*-Halbverbands. Dabei handelt es sich um die Potenzmenge $\mathcal{P}(S)$ der Menge $S := \{x, y, z\}$. Das Hasse-Diagramm ist nach der Inklusion geordnet.

Mithilfe der vorgestellten Eigenschaften lässt sich also eine Causal History [36] $\mathcal{C}$ für ein zustandsbasiertes Replikat x_i definieren [4, p. 7]:

(a) Initial gilt: $\mathcal{C}(x_i) = \emptyset$

(b) Nach ausgeführter Update-Operation f: $\mathcal{C}(f(x_i)) = \mathcal{C}(x_i) \cup \{f\}$

(c) Nach Zusammenführen von x_i und x_j: $\mathcal{C}(merge(x_i, x_j)) = \mathcal{C}(x_i) \cup \mathcal{C}(x_j)$

Dadurch lässt sich die Happens-Before Relation [2] wie folgt definieren [4, p. 7]: $f \rightarrow g \Leftrightarrow \mathcal{C}(f) \subset \mathcal{C}(g)$.

Nun stellen wir uns einen Datentypen vor, welcher zustandsbasiert arbeitet, also seinen Zustand als Nutzdaten versendet. Die Datenstruktur ist ein Join-Halbverband, wobei eine Funktion *merge(x, y)* bereitgestellt wird. Die Funktion *merge(x,y)* wertet dabei $x \sqcup y$ aus. Dieser monoton wachsende Zustand in Kombination mit der Ordnung $\leq$ führt dazu, dass der Zustand aller Replikate konvergiert. Dies bedeutet, dass nach einer Zusammenführung der resultierende Zustand größer oder gleich dem vorherigen Zustand ist.

Ein System, welches auf CvRDTs setzt, muss keine starken Eigenschaften im Bezug auf die Kommunikation zwischen den Knoten haben, da Updates alle Konten eventuell erreichen

werden. Dabei spielt es keine Rolle, ob einzelne Updates zwischen den Knoten verloren gehen, da Updates Knoten direkt oder indirekt erreichen können. Durch die Assoziativität und Kommutativität der Datenstruktur ist es egal, in welcher Reihenfolge Updates zusammengeführt werden. Die Idempotenz führt zu Toleranz gegen mehrfaches Zusammenführen des gleichen Zustands (vgl. [4, p. 10]).

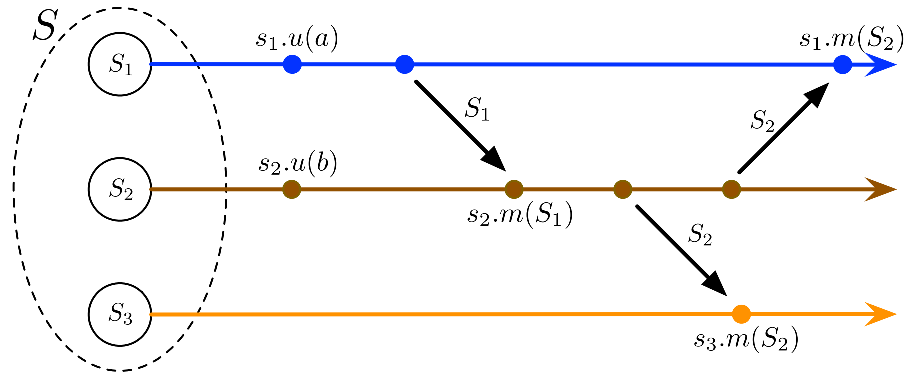

Abbildung 3.2: Zustandsbasierte Replikation [4, p. 6]

Abbildung 3.2 zeigt drei Replikate S_1, S_2, S_3. Zu sehen ist, dass S_1 und S_2 ihren Zustand nebenläufig verändern ($S_1.u(a)$ und $S_2.u(b)$). S_1 versendet nun seinen lokalen Zustand zu S_2, welcher diesen dann mit $S_2.merge(S_1)$ zusammenführt. Ab diesem Zeitpunkt gilt $S_2 \supset S_1$. Nach der Zusammenführung sendet S_2 seinen Zustand an S_3, was auf S_3 zu einer weiteren Zusammenführung führt. Nun gilt $S_3 \equiv S_2$, da S_3 keine für S_2 unbekannten Zustände besitzt. S_2 sendet seinen Zustand außerdem an S_1, was ein Zusammenführen auf S_1 zufolge hat. Nachdem alle Zustände zusammengeführt worden sind, gilt: $S_1 \equiv S_2 \equiv S_3$.

3.3 Operationsbasiert (CmRDT)

Operationsbasierte CRDTs werden auch „Op-based Commutative Replicated Data Type“ (CmRDT) genannt. CmRDTs versenden ausschließlich Operationen. Bei CmRDT ist die Ordnung der Operationen wichtig, da eine spätere Konsistenz sonst nicht gewährleistet werden kann. CmRDT sind deshalb mit Vorbedingungen ausgestattet, welche erfüllt sein müssen, bevor eine Operation angewendet werden kann.

Stellt ein System *Causal Delivery* bereit, so ist die Vorbedingung für viele *CmRDTs* immer erfüllt [4, p. 11, 23, 29, 33]. Im Folgenden ist ein Beispiel gegeben, weshalb die Ordnung unbedingt eingehalten werden muss (entweder durch die Vorbedingungen oder *Causal Delivery*):

Sei S eine Menge aus Elementen, es existieren drei Knoten i, j, k welche jeweils an der Replikation von S teilnehmen und somit eine eigene Kopie des Zustands halten. An Zeitpunkt t_0 fügt Knoten i ein Element x_1 in die Menge S_i hinzu. Die daraus resultierende Operation „o_1 *insert* x_1 *into* S“ wird an alle Replikate im Verbund verteilt. An Zeitpunkt t_1 empfängt Knoten j die Operation o_1 und verändert damit seinen lokalen Zustand (danach gilt $S_i \equiv S_j \wedge S_j \not\equiv S_k$). An Zeitpunkt t_2 entfernt Knoten j das Element x_1 aus der Menge S_j. Daraus resultiert die Operation „o_2 *remove* x_1 *from* S“, welche an alle Knoten verteilt wird. An Zeitpunkt t_3 empfängt Knoten k die Operation o_2 und entfernt x_1 aus seiner Menge. Das Problem ist jedoch, Operation o_2 hat Operation o_1 überholt. An Zeitpunkt $t4$ empfängt Knoten k die Operation o_1 und fügt Element x_1 hinzu. An Zeitpunkt $t5$ empfängt Knoten u die Operation o_2 und entfernt x_1. Alle Replikate sollten nun äquivalent sein, was jedoch nicht zutrifft, da $S_i = S_j = \emptyset \wedge S_k = \{x_1\} \implies S_{i,j} \not\equiv S_k$.

CmRDTs gelten als leichter zu implementieren und erzeugen weniger Verkehr im Netzwerk, dafür muss jedoch vom unterliegenden System, welches die Daten verteilt, mehr zugesichert werden, so zum Beispiel *exactly-once* Delivery um Idempotenz zu sichern. Um dies bereitzustellen ist also zusätzlicher Aufwand im System notwendig, da ein großer Log gehalten werden muss, um die Bedingungen sicherzustellen.

3.4 δ-CRDTs

δ-CRDTs [5] sind eine Optimierung von CvRDTs. Ändernde Zugriffe auf den CRDT generieren dabei einen sogenannten δ-Zustand, welcher typischerweise kleiner ist, als der vollständige Zustand. Durch den kleineren δ-Zustand, sind versendete Nachrichten kleiner als bei herkömmlichen CvRDTs, trotzdem eignen sich δ-CRDTs weiterhin für die Verwendung von nicht zuverlässiger Kommunikation, da insbesondere die *Idempotenz* von CvRDTs gegeben ist.
Jede ändernde Operation auf einem δ-CRDT gibt einen neuen δ-Zustand zurück, diese können sofort an andere Replikate im System propagiert werden. Aus Effizientsgründen lassen sich δ-Zustände auch lokal sammeln und zusammenfassen. Diese zusammengefassten δ-Zustände werden auch δ-Gruppe genannt [5, p. 5].

Da ein δ-Zustand kein vollständiger Zustand ist, kann beim Verteilen von δ-CRDTs periodisch der vollständige Zustand versendet werden. Dadurch können δ-Zustände, welche durch nicht zuverlässige Kommunikation eventuell verloren wurden, kompensiert werden [5, p. 8].

3.5 Bekannte CRDTs

In diesem Abschnitt werden ausgewählte und wohlbekannte CRDTs vorgestellt. Verschiedene Semantiken werden vorgestellt, welche sich fast beliebig mit verschiedenen Arten von Datentypen kombinieren lassen. Die hier vorgestellten Datentypen reichen von Registern, welche einen einzelnen Wert aufnehmen können, über Zähler bis hin zu Containern.

3.5.1 Last writer wins Register (LWW-Register)

Das *Last writer wins* Register (*LWW-Register*) [37] kann einen einzelnen Wert halten. Dabei enthält das Register ein Tupel $(\mathbb{S}, \mathbb{T}, \mathbb{I})$, der Wert des Registers ist mit $\mathbb{S}$ definiert, $\mathbb{T}$ ist ein Zeitstempel und $\mathbb{I}$ ein eindeutiger Bezeichner eines Knotens oder Prozesses.
Das Register hält immer den aktuellen Wert, das heißt, den Wert mit dem höchsten Zeitstempel aus $\mathbb{T}$. Sind zwei Zeitstempel nebenläufig, so gewinnt der Eintrag mit dem höchsten Bezeichner aus $\mathbb{I}$.

$$
\begin{aligned}
LWWRegister &= \mathbb{S} \times \mathbb{T} \times \mathbb{I} \\
set(v) &= (v, now(), id()) \\
M \sqcup M' &= \begin{cases} M' & \text{wenn } T < T' \vee (T || T' \wedge I < I') \\ M & \text{sonst} \end{cases}
\end{aligned}
$$

Abbildung 3.3: Zustandsbasiertes Last Writer Wins Register (LWW-Register)

Abbildung 3.3 zeigt die Definition eines *LWW-Registers*. Mithilfe der *set*-Funktion kann ein neuer Wert v gesetzt werden. Dabei wird ein Tupel mit v gebildet, *now()* beschreibt den zu v gehörigen Zeitstempel und $id()$ den Bezeichner des Knotens oder Prozesses, in dem die Operation durchgeführt wird.
Das Bilden des *supremums* von M und M' ist dabei abhängig von den Zeitstempeln oder Bezeichnern. Ist T kleiner als T', so gewinnt M'. Sind die Zeitstempel T, T' nebenläufig, so gewinnt der Tupel mit dem höheren Bezeichner I.

3.5.2 Multi-Value Register (MV-Register)

Ein alternativer Ansatz zum *LWW-Register* ist das *Multi-Value Register* (*MV-Register*). Nebenläufige Zuweisungen werden dabei vereint, es entsteht dabei eine Menge von nebenläufigen Zuweisungen. Ein Beispiel für die Verwendung von *MV-Register* ist das Datensystem Coda [38] oder Amazons shopping card Dynamo [9]. Ein neues Zuweisen eines Wertes dominiert die

aktuelle Menge von nebenläufigen Zuweisungen. Das verteilte Dateisystem Ficus [39] verwendet dabei einen generelleren Ansatz, nebenläufige Zuweisungen werden selbst noch einmal zusammengeführt zu einem einzigen Eintrag (vgl. [4, p. 20]).
Das *MV-Register* fasst nebenläufige Zuweisungen in einer Menge aus Tupeln von $(\mathbb{V}, \mathbb{T})$ zusammen. Der zugewiesene Wert wird in $\mathbb{V}$ dargestellt, als Zeitstempel dient ein Vektor-Uhr Zeitstempel aus $\mathbb{T}$.

$$
\begin{aligned}
MVRegister &= \mathcal{P}(\mathbb{V} \times \mathbb{T}) \\
set(v) &= \{(v, now())\} \\
M \sqcup M' &= \{(x,T) \in M | \forall (y,T') \in M' : T || T' \vee T \geq T'\} \cup \{(y,T') \in M' | \forall (x,T) : T' || T \vee T' \geq T\}
\end{aligned}
$$

Abbildung 3.4: Zustandsbasiertes Multi-Value Register (MV-Register) (vgl. [4, p. 20])

Abbildung 3.4 zeigt die Definition eines *MV-Registers*. Die *set*-Funktion zum Setzen eines neuen Wertes v generiert mithilfe der Funktion *now* einen neuen Zeitstempel. Dieser ist automatisch höher als der Zeitstempel des zu überschreibenden Wertes und dominiert damit die Menge.
Beim bilden des *Supremum* von M und M' werden alle Elemente vereinigt, die nicht von einem Element in der anderen Mengen dominiert werden [4, p. 20].
Eine *get*-Funktion ist hier nicht weiter definiert, würde aber einfach die aktuelle Menge S des Registers zurückgeben.

3.5.3 Flag

Flags sind Boolsche Register, welche entweder den Wert *true* oder *false* haben können. Dabei gibt es verschiedenste Semantiken, von *true-wins* oder *false-wins*, bis hin zu *Last-Writer-Wins*-Semantiken.

$$
\begin{aligned}
Flag_{\vee} &= \mathbb{B} \\
\bot &= \mathit{false} \\
set(v) &= B := v \vee B \\
get &= B \\
B \sqcup B' &= B := B \vee B'
\end{aligned}
$$

(a) True Wins Flag

$$
\begin{aligned}
Flag_{\wedge} &= \mathbb{B} \\
\bot &= \mathit{true} \\
set(v) &= B := v \wedge B \\
get &= B \\
B \sqcup B' &= B := B \wedge B'
\end{aligned}
$$

(b) False Wins Flag

Abbildung 3.5: Zustandsbasierte Flags

Abbildung 3.5 zeigt zwei mögliche Flag Semantiken. Zum einen das *True-Wins*-Flag (*TWFlag*) in Abbildung 3.5a. Dabei handelt es sich um ein Flag, welches nach dem setzen auf *true* nicht wieder in den *false* Zustand versetzt werden kann. Bei der Initialisierung dieses Datentypes ist der Wert *false*. Bei einem *merge* beziehungsweise bilden des *Supremums* gewinnt *true*. In Abbildung 3.5b ist das Gegenstück zu sehen, hierbei handelt es sich um das *False-Wins*-Flag (*FWFlag*). Hier sind die Bedingungen zum setzen des Wertes genau umgekehrt und der Initialwert ist *true*. Bei der Bildung des *Supremum* gewinnt außerdem *false*.

3.5.4 Grow-only Counter (GCounter)

Bei den Zählern beginnen wir mit dem *Grow-Only-Counter* (*GCounter*). Der *GCounter* unterstützt somit nur die veränderliche Operation des Hochzählens. Fälle wie das Hochzählen um negative Werte muss dabei abgefangen werden. Bei dem *GCounter* handelt es sich eigentlich um eine Map, welche Knoten Bezeichner auf einen Integer abbildet.

$$
\begin{aligned}
GCounter &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\bot &= \{\} \\
inc_i &= m\{i \mapsto m(i) + 1\} \\
inc_by_i(v) &= m\{i \mapsto m(i) + |v|\} \\
value(m) &= \sum_{i \in \mathbb{I}} m(j) \\
m \sqcup m' &= \{j \mapsto max(m(j), m'(j)) \mid j \in \operatorname{dom} m \cup \operatorname{dom} m'\}
\end{aligned}
$$

Abbildung 3.6: Zustandsbasierter Grow-only Counter (CvRDT GCounter) (vgl. [5, p. 4])

Abbildung 3.6 zeigt die Definition eines *GCounters* als CvRDT. Wir sehen eine *inc*-Funktion, welche für die Inkrementierung zuständig ist. Die *inc_by*-Funktion benötigt zusätzlich einen Parameter v. Um zu verhindern, dass ein negativer Wert für v angegeben wird, so wird $|v|$ verwendet. Der totale Wert des *GCounters* wird mit der *value*-Funktion bestimmt, dabei wird über die Map iteriert und alle Werte werden aufsummiert. Der totale Wert ist also die Summe der Werte aller Knoten. Die *Merge*-Funktion, hier als *Supremum* $m \sqcup m'$ ist, der maximale Wert jedes Key/Value Paares der Maps von m und m', dabei muss j in der Definitionsmenge von m oder m' sein.
Diese Datenstruktur entspricht einer Vector Clock, wobei jedoch andere Funktionen auf der Datenstruktur definiert sind. Die Vector Clock gilt ebenfalls als CRDT.

3.5.5 PN-Counter

Der *PN-Counter* ist ein Zähler, welcher nicht nur hochgezählt werden kann wie der *GCounters*, sondern auch heruntergezählt werden kann. Diese Art von Zähler kann aus der Kombination von zwei *GCountern* erreicht werden. Dabei sollte der Zähler nie für globale Bedingungen verwendet werden. Ein Beispiel für eine solche Bedingung ist „der Zähler darf nicht unter -2 Fallen". So eine Bedingung kann lokal nicht zugesichert werden und darf deshalb nicht verwendet werden.

$$
\begin{aligned}
PNCounter &= \mathbb{I} \hookrightarrow \mathbb{N} \times \mathbb{N} \\
\bot &= \{\} \\
inc_i &= p\{i \mapsto p(i) + 1\} \\
dec_i &= n\{i \mapsto n(i) + 1\} \\
modify_by(v) &= \begin{cases} p\{i \mapsto p(i) + v\} & \text{wenn } v > 0 \\ n\{i \mapsto n(i) + |v|\} & \text{sonst} \end{cases} \\
value(p, n) &= \sum_{i \in \mathbb{I}} p(j) - \sum_{i \in \mathbb{I}} n(j) \\
p \sqcup p' &= \{j \mapsto max(p(j), p'(j)) \mid j \in \operatorname{dom} p \cup \operatorname{dom} p'\} \\
n \sqcup n' &= \{j \mapsto max(n(j), n'(j)) \mid j \in \operatorname{dom} n \cup \operatorname{dom} n'\} \\
(p, n) \sqcup (p', n') &= (p \sqcup p', n \sqcup n')
\end{aligned}
$$

Abbildung 3.7: Zustandsbasierter Positiv-Negativ Counter (CvRDT PNCounter)

Abbildung 3.7 zeigt die Definition eines *PN-Counters* als CvRDT. Es ist zu sehen, wie ein Bezeichner aus $\mathbb{I}$ auf ein Zweiertupel aus $\mathbb{N} \times \mathbb{N}$ abbildet. Der erste Index des Tupel zählt dabei die Inkrementierungen und ist mit p beschrieben, der zweite zählt die Dekrementierungen und ist mit n beschrieben.
Die *modify_by*-Funktion zählt entweder den Wert von p oder n hoch, in Abhängigkeit ob der Wert v, um welchen modifziert werden soll, positiv oder negativ ist. Die *value*-Funktion, welche den totalen Wert des Zählers wiedergibt, wird dabei gebildet, indem beide Maps aufsummiert werden und die Summen voneinander abgezogen werden. Die *merge*-Funktion hat dabei das selbe Verhalten wie bei dem *GCounter*, siehe *Supremum* von n und p.

3.5.6 Grow-only Set (GSet)

Die erste Menge oder Container, welcher vorgestellt wird, ist das *Grow-Only-Set* (*GSet*). Das *GSet* kann dabei als Menge betrachtet werden, welche als Operation nur die Vereinigung ($\cup$) sowie $\in$ unterstützt. Ein Element kann nur genau einmal enthalten sein.

$$
\begin{aligned}
GSet &= \mathcal{P}(\mathbb{M}) \\
\bot &= \{\} \\
insert(e) &= \{e\} \cup M \\
subset_insert(S) &= S \cup M \\
contains(e) &= e \in M \\
M \sqcup M' &= M \cup M'
\end{aligned}
$$

Abbildung 3.8: Zustandsbasiertes Grow-Only Set (CvRDT GSet)

Abbildung 3.8 zeigt diesmal die Definition eines *GSet*. Initial ($\bot$) ist die Menge leer. Grundlegende Funktionen wie *insert* und *lookup* werden bereitgestellt. Die erste Funktion *insert* mit einem Argument e fügt dabei ein Element in die Menge ein. Es ist außerdem möglich, mit *subset_insert* eine ganze Menge S einzufügen ($S \cup M$). Die *lookup*-Funktion nimmt dabei ein Element e entgegen und prüft, ob dieses Element in M enthalten ist ($e \in M$). Es ist außerdem möglich, Funktionen zu definieren, welche eine Teilmengenbeziehung prüft. Die *merge*-Funktion oder *Supremum* $M \sqcup M'$ ist dabei die einfache Vereinigung beider Mengen $M \cup M'$.

3.5.7 Two-Phase Set (2PSet)

Das *Two-Phase Set* (*2PSet*) ist eine Menge, aus welcher Elemente auch herausgenommen werden können. Elemente können, nachdem diese entfernt wurden, nicht wieder eingefügt werden. Aufgrund dieser Semantik kann auch von „remove-wins“-Semantik gesprochen werden. Das *2PSet* besteht diesmal aus zwei Mengen A, R. Dabei sind Elemente welche sich im *2PSet* befinden nur in A und entfernte Elemente in R.

$$
\begin{aligned}
2PSet &= \mathcal{P}(\mathbb{M}) \times \mathcal{P}(\mathbb{M}) \\
\bot &= (\{\}, \{\}) \\
insert(e) &= \{e\} \cup A \\
remove(e) &= \begin{cases} \{e\} \cup R & \text{wenn } e \in A \\ \emptyset \cup R & \text{sonst} \end{cases} \\
contains(e) &= e \in A \vee e \notin R \\
(A, R) \sqcup (A', R') &= (A \cup A', R \cup R')
\end{aligned}
$$

Abbildung 3.9: Zustandsbasiertes Two-Phase Set (CvRDT 2PSet)

In Abbildung 3.9 ist die Definiton eines 2PSet erläutert. Initial sind beide Mengen A, R leer. Wie beim *GSet* wird ein *insert* von einzelnen Elementen unterstützt, *subset_insert* ist generell genau so möglich. Eine *remove*-Funktion wird bereitgestellt, welche Element e aus dem *2PSet* entfernt. Es muss jedoch sichergestellt sein, dass e bereits zu dem *2PSet* hinzugefügt wurde. Um dies sicherzustellen wird geprüft, ob e bereits in A ist ($e \in A$). Wenn diese Bedingung erfüllt ist, so wird e in die Menge R hinzugefügt, ansonsten wird R mit der leeren Menge $\emptyset$ vereinigt. Um zu prüfen, ob ein Element e in dem *2PSet* vorhanden ist, wird die *contains*-Funktion verwendet. Diese prüft, ob das Element e in A existiert und nicht in R ($e \in A \wedge e \notin R$).
Zur Optimierung dieses Sets ist es möglich, nach Entfernen eines Eintrags die Menge A zu bereinigen, dazu kann lokal die Operation $A \leftarrow A \setminus R$ ausgeführt werden. Dies ist optional und optimiert den Speicherverbrauch.

3.5.8 Last-Writer-Wins-Element Set (LWW-Element-Set)

Das *Last-Writer-Wins-Element Set* (*LWW-Element-Set*) ist ein Set, welches im Gegensatz zum *2Pset* das einfügen von bereits entfernten Elementen erlaubt.

$$
\begin{aligned}
LWWSet &= \mathcal{P}(\mathbb{M} \times \mathbb{T}) \times \mathcal{P}(\mathbb{M} \times \mathbb{T}) \\
\bot &= (\{\}, \{\}) \\
insert(e) &= \{(e, now())\} \cup A \\
remove(e) &= \{(e, now())\} \cup R \\
contains(e) &= \exists t, \forall t' > t : (e, t) \in A \wedge (e, t') \notin R \\
(A, R) \sqcup (A', R') &= (A \cup A', R \cup R')
\end{aligned}
$$

Abbildung 3.10: Zustandsbasiertes Last Writer Wins Set (CvRDT LWWSet)

Abbildung 3.10 zeigt die Definition eines *Last-Writer-Wins-Element-Set*. Es besteht aus zwei Mengen A und R, welche jeweils über das Tupel aus einem Element e und einem Zeitstempel t bestehen. Initial sind beide Mengen leer.
Wird ein Element e mit *insert(e)* eingefügt, wird ein Tupel $(e,$ *now()*$)$ mit der Menge für hinzugefügte Elemente A vereinigt. Die Funktion *now()* generiert dabei einen Zeitstempel, welcher den Zeitpunkt des Hinzufügens darstellt. Zum Entfernen eines Elements e wird die *remove(e)* verwendet. Dabei wird ein neues Tupel $(e,$ *now*$)$ mit der Menge für entferne Elemente R vereinigt.
Um zu prüfen, ob ein Element im Set enthalten ist, wird die Funktion *contains(e)* verwendet. Dabei wird geprüft, ob es kein Tupel mit dem entsprechenden Element e in R gibt, welches einen höheren Zeitstempel hat, als ein Tupel welches e enthält und sich in A befindet [4, p. 24].
Zur Optimierung des Sets ist es möglich, lokal veraltete Einträge (Tupel) zu entfernen.

4 Design des CAF-CRDT-Moduls

4.1 Design-Ziele

Dieser Abschnitt stellt die Design-Ziele des Replikationsmoduls vor und erklärt, wie diese erreicht werden.

Effiziente Kommunikation Um einen geringen Kommunikationsaufwand des Moduls zu gewährleisten, sollen so wenig Daten wie möglich zwischen den Knoten gesendet werden. Die zu versendenden Nutzdaten müssen pro änderndem Zugriff auf ein Replikat so gering wie möglich sein.

Skalierbarkeit Die Performance einzelner Knoten sowie des verteilten Systems soll nicht stark an die (a) Anzahl der Rechner im Replikationsverbund und (b) Anzahl der zu replizierenden Objekte gebunden sein.

Konfigurierbarkeit Das Design muss Parametrisierbarkeit im Bezug zu anwendungsspezifischen Parametern bieten. Dazu zählt, wie schnell Änderungen im System propagiert werden (lokal sowie verteilt). Denkbar ist, dass sich Parameter, die den Austausch von Meta-Daten betreffen, anders einstellen lassen als Parameter für den Austausch von Nutzdaten (Replikat-Daten). Eine häufige Übertragung von Nutzdaten kann zum Beispiel das Netzwerk im Übermaß belasten, weshalb die Flush-Intervalle der verschiedenen Buffer im CRDT-Modul pro Laufzeit-Instanz konfigurierbar sein sollten.

Benutzerfreundliche API Wichtig hierbei ist es, die allgemeine Benutzerfreundlichkeit von CAF auch in dem neuen Modul bereitzustellen. Eine benutzerfreundliche API ist wenig komplex in der Verwendung. Dies wird dadurch erreicht, dass beispielsweise Funktionsaufrufe keine komplexen Signaturen und eindeutige Namen der einzelnen Übergabeparameter besitzen.

Erweiterbarkeit Das Modul stellt eine Sammlung von einfachen CRDTs bereit. Diese Sammlung besteht aus Grundtypen wie Registern, Zählern und Containern. Es muss möglich sein,

diese Grundtypen mit eigenen, vom Benutzer definierten Typen zu spezialisieren, ohne dafür bestehenden Code ändern zu müssen. Des Weiteren müssen vom Benutzer eigene CRDTs implementierbar sein, ohne den Code am eigentlichen Modul ändern zu müssen.

4.2 Integration von CRDTs in CAF

Dieser Abschnitt diskutiert die Darstellung von CRDTs in CAF. Im Folgenden wird das Wort Replikat verwendet, ein Replikat ist eine lokale Instanz einer verteilten CRDT. Pro Knoten kann es ein oder mehrere verschiedene Replikate geben. Abschnitt 4.2.1 geht dabei auf die Art des Zugriffs auf CRDTs beziehungsweise Replikate ein. Es wird gezeigt, wie CAF die Konvergenz aller Replikate im verteilten System sicherstellt. Abschnitt 4.2.2 diskutiert, wie verschiedene Replikate voneinander unterschieden werden können und was dafür notwendig ist.

4.2.1 Zugriff auf CRDT-Instanzen

Bei der Wahl der Art des Zugriffs auf Replikate ist es wichtig, dass das zugrunde liegende Programmiermodell nicht verletzt wird. Unterschiedliche Zugriffsarten können Auswirkungen auf die Laufzeit und den Speicherverbrauch haben. Im folgenden werden verschiedene Varianten des Zugriffs auf Replikate diskutiert.
Es ist möglich, dass in einem CAF-Knoten mehrere Aktoren existieren, die mit einem Replikat derselben Art arbeiten. Konkurrierender Zugriff auf ein Replikat von mehreren lokalen Aktoren kann dabei zum Problem werden.

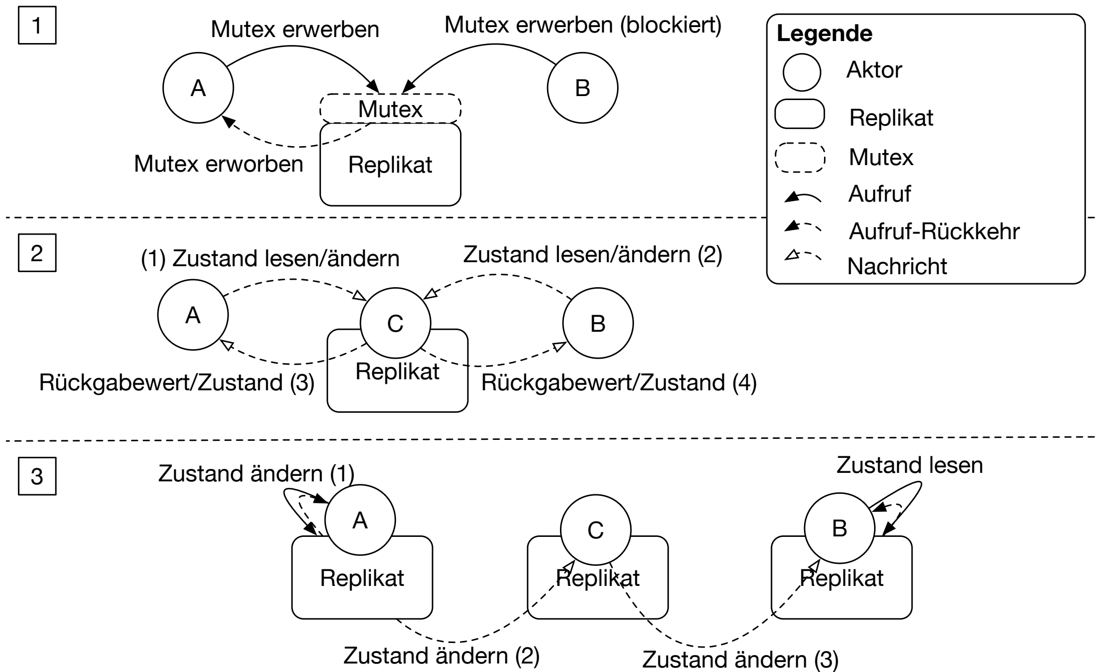

Abbildung 4.1: Möglichkeiten für den konkurrierenden lokalen Zugriff auf Replikate

Abbildung 4.1 zeigt drei Möglichkeiten zum Zugriff auf (lokale) Replikate. Zu sehen sind Aktoren (Kreise), Replikate (Vierecke), Nachrichten (gestrichelte, nicht ausgefüllte Pfeile), Funktionsaufrufe (Pfeile) und Rückkehr von Funktionsaufrufen (gestrichelte ausgefüllte Pfeile). In Variante Eins ist zudem ein Mutex zu sehen (gestricheltes Viereck). Es werden drei verschiedene Arten des Zugriffs gezeigt, welche von Eins bis Drei nummeriert sind und durch gestrichelte Linien getrennt sind.
Die erste Variante zeigt die klassische Synchronisation von Zugriffen auf eine Ressource mit Hilfe eines Mutexes, welche aus der Objekt-Ortientierten-Programmierung (OOP) bekannt ist. Die zu schützende Ressource ist hier ein Replikat. Soll auf die Ressource zugegriffen werden, muss zuvor das Mutex erworben werden. Dabei kann das Mutex nur von einer Entität (in diesem Fall Aktoren) zur Zeit erworben sein. Sollte ein weiterer Aktor zeitgleich mit dem Replikat arbeiten wollen, so wird der Zugriff blockiert, bis der aktuelle Besitzer des Mutex dieses wieder freigegeben hat. Diese Variante ist für CAF nicht geeignet, da Aktoren kooperativ gescheduled sind, dabei werden Aktoren von Worker-Threads ausgeführt. Sollte ein Aktor blockieren, wird der Worker-Thread blockiert und kann nicht zum Ausführen anderer Aktoren genutzt werden.
Variante Zwei zeigt, wie ein zusätzlicher System-Aktor (C) das Replikat verwaltet. Dabei

werden Anfragen zum Lesen oder Ändern des Replikats per Nachricht an den verwaltenden Aktor (C) gestellt. Dieser arbeitet die Nachrichten in First-In-First-Out (FIFO) Ordnung ab und gewährleistet dabei asynchrones Arbeiten. Dieses Verfahren ist von der Synchronisation über Nachrichten kompatibel mit dem Aktormodell. Jedoch können große Latenzen auftreten, da die Nachrichten in Aktor C nur sequenziell verarbeitet werden. Problematisch kann dies werden, wenn viele lokale Aktoren mit der Ressource arbeiten.
Bei der dritten Variante hält jeder Aktor ein eigenes unabhängiges Replikat in seinem Zustand. Da die Replikate unabhängig voneinander geschrieben und gelesen werden können, ist es nicht notwendig, Zugriffe in irgendeiner Art zu synchronisieren. Ändernde Zugriffe werden im Hintergrund an andere Kopien des Replikats repliziert. Dazu ist es notwendig, dass ein zusätzlicher System-Aktor eine Kopie des Replikats hält und alle anderen Aktoren kennt, welche mit Kopien des Replikats arbeiten. Von dort aus werden Änderungen an den Replikaten weitergereicht. Da in dieser Variante potenziell mehr Replikate existieren, ist der Speicherverbrauch bei dieser Art des Zugriffs erhöht. Es müssen mehr Replikate konsistent gehalten werden, außerdem sind mehr Nachrichten notwendig, da jeder Aktor, welcher ein Replikat besitzt, informiert werden muss. Durch das Halten im Zustand der Aktoren ist es jedoch möglich, das Replikat zu cachen. Dadurch kann direkt und ohne zusätzliche Latenz auf ein Replikat zugegriffen werden.

Die Verwendung von Replikaten innerhalb des Zustands von Aktoren passt am besten zu CRDTs (Variante Drei). Änderungen an einzelnen Replikaten können ohne Konflikte lokal zusammengeführt und im verteilten System propagiert werden. Es ist keine Synchronisation der Zugriffe notwendig. Lesende und schreibende Zugriffen sind möglichst kurz, da kein zusätzlicher Aktor dazwischen geschaltet ist, welcher das Replikat sequenziell verwaltet und so den Zugriff synchronisiert. Ein Aktor, der ein Replikat in seinem Zustand hält, kann direkt auf den Replikat-Daten arbeiten. Benutzer-Aktoren, welche sensitiv auf Änderungen eines Replikats sind, werden durch Benachrichtigungen (Nachrichten) informiert. Dadurch lässt sich „Polling“ auf dem Replikat vermeiden.

4.2.2 Adressierung von Replikaten

Replikate müssen einer Art zuordenbar sein. Dabei können Replikate von unterschiedlichen Typen sein. Es ist also notwendig, (a) den Typen anhand eines Bezeichners zu erkennen und (b) verschiedenartige Replikate vom dann selben Typ unterscheidbar zu machen. Dies ist notwendig, um zum Beispiel von anderen Knoten empfangene Änderungen, an die passenden Replikate verteilen zu können. Der verwendete Bezeichner sollte auch für den Anwendungsentwickler lesbar sein, da dieser zum Debugging oder Logging verwendet werden kann.

Die Art des Namensraumes, welcher vom Anwendungsentwickler kontrollierbar ist, sollte strukturiert sein. Flache Namensräume eignen sich nicht, um eine geeignete Struktur herzustellen. Der Namensraum sollte deshalb hierarchisch sein, um auch bei vielen Arten von Replikaten eine Übersicht über die im verteilten System verwendeten Replikate zu haben.

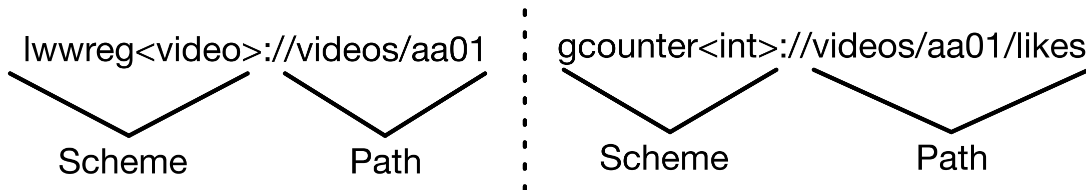

Abbildung 4.2: Kodierung zweier Replikat-Bezeichners

Abbildung 4.2 zeigt zwei Replikat-Bezeichner als URI [40] kodiert. Dabei wird der *Scheme* als Typ-Bezeichner verwendet. Der *Path* ist dabei als hierarchischer Namensraum verwendbar. Zu sehen sind zwei Beispiele, wie sich (a) der Typ-Bezeichner unterscheidet und (b) der *Path* eine weitere Tiefe bekommt.

4.3 Auswahl einer geeigneten CRDT-Klasse für CAF

In diesem Abschnitt wird die Wahl der verwendeten CRDT-Klasse mit ihren Auswirkungen auf das Design diskutiert. Diese lassen sich in zwei Auswirkungen unterteilen. Die erste Auswirkung bezieht sich auf die aus der CRDT-Klasse resultierenden Anforderungen an das unterliegende Kommunikations-System und ist in Abschnitt 4.3.1 diskutiert. Die zweite Auswirkung bezieht sich auf die aus der CRDT-Klasse entstehende Effizienz der Verteilung und ist in Abschnitt 4.3.3 diskutiert.

4.3.1 Anforderungen an das Kommunikations-System

Das Kommunikations-System ist dafür zuständig, dass lokal anfallende Änderungen an Replikaten alle Knoten im verteilen System erreichen. Damit ist das Design des Kommunikations-Systems verantwortlich für den entstehenden Kommunikationsaufwand und die Skalierbarkeit des Moduls. Um das Kommunikations-System skalierbar zu halten, soll es ohne großen Zustand auskommen, also möglichst *Stateless* gehalten sein. Zusätzlich soll es lose gekoppelt sein und kein komplexes Protokoll zur Kommunikation zwischen den Knoten verwenden.

Unzuverlässige Kommunikation in CAF

Fehlersemantiken sind wichtig, da verwendete Algorithmen auf die Fehlersemantik des Systems ausgelegt sein müssen. Die Fehlersemantik des Systems wirkt sich auch auf die Auswahl der CRDT-Klasse aus. Das CAF-I/O-Modul arbeitet mit TCP-Verbindungen, wodurch die Eigenschaften wie Deduplikation und erneutes Versenden von Nachrichten aus verlorenen Paketen geerbt wird. CAF-Knoten benötigen nicht zu jedem anderen CAF-Knoten eine direkte Verbindung, es ist möglich, dass Nachrichten über andere CAF-Knoten geroutet werden. Dies geschieht, wenn sich CAF-Knoten über einen anderen Knoten kennengelernt haben ohne eine direkte Verbindung zueinander. Nachrichten zwischen den nicht direkt miteinander verbundenen Knoten werden über die direkt verbundenen Knoten geroutet. Zwischen den Knoten stellt TCP sicher, dass Nachrichten nicht verloren gehen können. Es kann jedoch passieren, dass ein CAF-Knoten abstürzt, während dieser eine Nachricht weiterleitet. Dies kann durch die TCP-Verbindungen zwischen den CAF-Knoten nicht abgefangen werden. Nachrichten haben in CAF deshalb nur eine *at-most-once* Fehlersemantik. CAF selbst implementiert keine Funktionalität, um festzustellen, ob geroutete Nachrichten angekommen sind, es findet kein erneutes Versenden statt.

Semantik	**Wie oft können Nachrichten ankommen?**
maybe	Gar nicht, oder beliebig oft
at-least-once	Mindestens einmal
at-most-once	Gar nicht, oder einmal
exactly-once	Genau einmal

Tabelle 4.1: Die verschiedenen Fehlersemantiken beim Versenden von Nachrichten

Tabelle 4.1 zeigt die verschiedenen Fehlersemantiken im Bezug auf Nachrichten. Die Tabelle listet die verschiedenen Fehlersemantiken von schwach zu stark, wobei starke Fehlersemantiken aufwendiger zuzusichern sind.

CmRDTs Durch *CmRDTs* würden zusätzliche Anforderungen an das Kommunikations-System gestellt werden. *CmRDTs* haben keine idempotente Eigenschaft, weshalb höhere Anforderungen an den Nachrichtenversand gestellt werden. Die von CAF bereitgestellte Fehlersemantik reicht dabei nicht aus um diese CRDT-Klasse zu verwenden. Es müsste ein Algorithmus implementiert werden, welcher feststellt, ob Nachrichten verloren gegangen sind und diese erneut versenden kann. Dies würde zu viel Zustand innerhalb des Algorithmus führen, da Operationen in einem Log („History“) gehalten werden müssen, bis alle Knoten die Opera-

tion erhalten haben. Jedes Replikat würde zudem eine eigene History führen, welche bei Änderungen mit hoher Frequenz schnell wächst.

CvRDTs Diese Klasse der CRDTs stellt keine zusätzlichen Anforderungen an das Kommunikations-System. Durch die idempotente *merge*-Funktion können Nachrichten beliebig oft ankommen. Sollten Nachrichten verloren gehen und der Zustand damit nicht komplett sein, wird bei der nächsten erfolgreichen Übertragung die verlorene Information durch einen *merge* ergänzt. Damit reicht die von CAF gestellte Fehlersemantik aus, um diese Klasse von CRDTs zu verwenden.

δ-CRDTs δ-CRDTs sind eine Abwandlung von *CvRDTs*, wobei die zu versendenden Datenmengen generell kleiner sind als bei *CvRDTs*. Sie besitzen wie *CvRDTs* eine idempotente *merge*-Funktion. Auch hier werden keine weiteren Anforderungen an das Kommunikations-System gestellt, da die gleichen Eigenschaften wie bei *CvRDTs* gelten. Allerdings muss jeder Knoten periodisch, anstelle von δ-CRDTs, vollständige Zustände (wie bei *CvRDTs*) versenden, um eventuell verlorene Nachrichten auszugleichen.

CmRDTs eignen sich weniger für die Integration in CAF, da die Fehlersemantik, welche für diese Klasse am geeignetsten wäre (*exactly-once*), nicht gegeben ist. Um die Anforderungen zu erfüllen, welche durch *CmRDTs* gestellt werden, würden die Ziele eines möglichst schlanken Designs, was ohne großen Zustand auskommt verletzt werden. Außerdem müsste ein aufwendigeres Protokoll unter den Knoten verwendet werden, inklusive dem Buffern von Nachrichen, um Verluste von Nachrichten ausgleichen zu können.
Für die Integration in CAF eignen sich CvRDTs oder δ-CRDTs am besten, da diese eine idempotenten *merge*-Funktion bieten. Die von CAF gegebene Liefergarantie ist für diese beiden Klassen von CRDTs ausreichend.

4.3.2 Auswirkung der CRDT-Klasse auf die Verteilung

Eine effiziente Verteilung von Replikat-Daten ist notwendig, um Netzwerk-Ressourcen zu schonen und das verteilte System skalierbar zu halten. Die Effizienz der Verteilung misst sich anhand der versendeten Datengröße im Bezug zur Größe und Frequenz der Änderungen. Die Wahl der CRDT-Klasse hat Auswirkungen auf die Effizienz der Verteilung. Im Folgenden beschreibe ich die Auswirkungen der Wahl der CRDT-Klasse auf die Effizienz der Verteilung.

CmRDT CmRDTs eignen sich zur effizienten Verteilung, da nur Operationen versendet werden. Diese sind von ihrer Nutzlast klein und verursachen dadurch kaum Auslastung im Netzwerk.

CvRDT CvRDTs versenden immer den vollständigen Zustand, daher sind die zu versendeten Daten pro Änderung wesentlich größer als bei operationsbasierten Verfahren. CRDT-Instanzen, auf welchen bereits viele Operationen ausgeführt wurden, haben als CvRDT einen großen Zustand (monoton wachsend), was dazu führt, dass die übertragene Datenmenge mit der Zeit immer größer wird.

δ-CRDT Da δ-CRDTs bei einer Änderung nicht ihren vollständigen Zustand neu versenden, sondern nur den δ-Zustand zum vorherigen Zustand, eignen sich δ-CRDTs für eine effiziente Verteilung. Ein δ-Zustand ist dabei kleiner als der komplette CRDT-Zustand (vgl. [5, p. 6]).

4.3.3 Entscheidung über die CRDT-Klasse

Im Folgenden sind die Vor- und Nachteile der einzelnen CRDT-Klassen zusammengefasst aufgelistet.

CmRDTs eignen sich weniger für CAF, da die von CAF gegebene Fehlersemantik nicht ausreicht, um ein schlankes Kommunikations-System für *CmRDTs* zu designen. Es wäre notwendig, dass jedes Replikat eine History der getätigten Operationen führt. Es müsste außerdem zusätzlicher Aufwand betrieben werden, um Nachrichten, die verloren gegangen sind, erneut zu senden. Des Weiteren wäre es notwendig, Nachrichten auf Empfängerseite zwischenzuspeichern, welche aufgrund von noch nicht erfüllter (*CmRDT* spezifischer) Vorbedingungen (durch eventuell verlorene Nachrichten) noch nicht angewendet werden konnten. In Systemen, welche in kurzer Zeit viele Änderungen auf Replikaten haben, kann es so zu einem erhöhten Speicherverbrauch kommen. Die durch *CmRDTs* verursachte Netzwerk-Belastung ist gering, da Operationen schlanker sind als Zustände von *CvRDTs* oder *δ-CRDTs*.

CvRDTs eignen sich besser für CAF, da die von CAF gegebenen Fehlersemantik ausreicht, um ein schlankes Kommunikations-System zu designen. Um eventuell auftretende Nachrichtenverluste auszugleichen, reicht es, die nächste Nachricht einzuspielen. Das Hauptproblem von *CvRDTs* ist jedoch die Last, die auf dem Netzwerk erzeugt wird, da bei jeder Änderung der komplette Zustand übertragen wird. Eine Alternative bietet hier die Verwendung von δ-CRDTs, welche dieselben Anforderungen an die Liefergarantien von CAF stellen wie *CvRDTs*, jedoch

wesentlich weniger Last auf dem Netzwerk erzeugen.
Da mit *δ-CRDTs* ein simpler Verteil-Algorithmus realisierbar ist und die Last im Netzwerk geringer als bei *CvRDTs*, werden in dem CAF-CRDT-Modul *δ-CRDTs* verwendet.

4.4 Austausch von Meta-Informationen

Um Replikat-Daten gezielt an CAF-Knoten verteilen zu können, müssen sich Knoten darüber austauschen, welche Replikate von welchen Knoten repliziert werden. Die dabei zu verteilende Meta-Information („welcher Knoten instanziiert welche Replikate“) werden getrennt von den eigentlichen Replikat-Daten übertragen. Jeder CAF-Knoten, der das CRDT-Modul geladen hat, nimmt an der Übertragung von Meta-Informationen teil.
Jeder CAF-Knoten kennt die Replikat-Bezeichner, die die anderen CAF-Knoten replizieren. Damit weiß jeder Knoten, welche Replikate auf welchen Knoten repliziert sind.
Im Folgenden wird der Algorithmus zum Verteilen der Meta-Informationen erläutert, dazu wird für die Menge an eindeutigen Knoten-Bezeichnern $\mathbb{I}$ verwendet. Es gibt eine Menge an Replikat-Bezeichnern $\mathbb{R}$. Für Versionsnummern werden Elemente der Menge $\mathbb{Z}$ verwendet.
Jeder CAF-Knoten hat zugriff auf seinen eindeutigen Bezeichner $K_{id} \in \mathbb{I}$. Der Algorithmus besteht in jedem Knoten aus zwei Maps. Die erste Map V bildet von Knoten-Bezeichnern auf Versionsnummern ab ($V : \mathbb{I} \mapsto \mathbb{Z}$). Die zweite Map D bildet von Knoten-Bezeichnern auf eine Menge von Replikat-Bezeichnern ab ($D : \mathbb{I} \mapsto \mathcal{P}(\mathbb{R})$). Initial gilt, dass V auf 0 abbildet, D bildet initial auf die leere Menge ab.

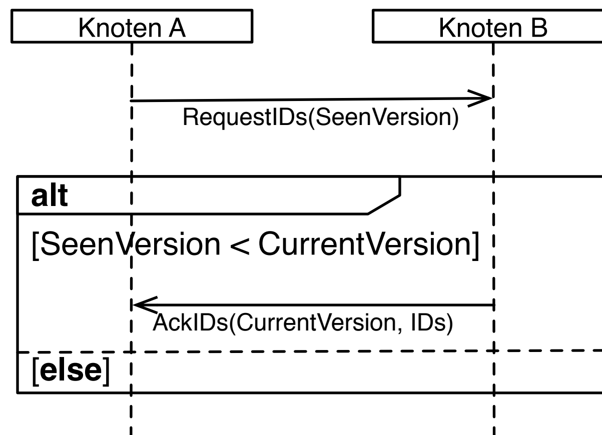

Abbildung 4.3: Periodischer Austausch von Meta-Informationen zwischen zwei CAF-Knoten

Abbildung 4.3 zeigt zwei CAF-Knoten (A, B). Es ist zu sehen, wie Knoten A eine Nachricht an Knoten B versendet, dabei wird die letzte bekannte Versionsnummer (*SeenVersion*: $V_A[id_B]$) von Knoten B mit versendet. Knoten B empfängt die *SeenVersion* und vergleicht diese mit der lokalen Versionsnummer (*CurrentVersion*: $V_B[id_B]$). Ist die von Knoten B empfangene Versionsnummer kleiner als die lokale Versionsnummer, bedeutet dies, dass der anfragende Knoten A nicht die aktuelle Version der Replikat-Bezeichner von Knoten B hat. Knoten B sendet nun als Antwort eine neue Nachricht, welche die aktuelle Versionsnummer sowie die aktuellen Replikat-Bezeichner von Knoten B enthält. Knoten A überschreibt die Einträge für Knoten B in seinen lokalen Maps V_A und D_B mit den empfangenen Daten. Sollte Knoten B die aktuelle Versionsnummer empfangen haben, wird keine Antwort gesendet und die Nachricht ignoriert. Dieser Ablauf wird periodisch von allen Knoten ausgeführt.

$$InitReplikat(R_{id}) = D[K_{id}] \cup R_{id} \wedge V[K_{id}] + 1$$
$$RemReplikat(R_{id}) = D[K_{id}] \setminus R_{id} \wedge V[K_{id}] + 1$$

Abbildung 4.4: Initialisieren und Entfernen von lokalen Replikaten

Beim lokalen Instanziieren oder Entfernen von Replikaten werden die lokalen Meta-Informationen geändert. Abbildung 4.4 zeigt, wie die Maps geändert werden. Beim Neuinstanziieren von einem Replikat mit dem Bezeichner R_{id} wird der Bezeichner einfach in die Menge der lokal instanziieren Replikat-Bezeichner eingefügt. Die lokale Versionsnummer wird um Eins inkrementiert.
Beim Entfernen eines Replikats mit dem Bezeichner R_{id} wird der Eintrag aus der lokalen Menge entfernt. Die lokale Versionsnummer wird um Eins inkrementiert.

Der gezeigte Algorithmus zum Synchronisieren der Meta-Informationen versendet im Falle, dass keine Änderungen an den Meta-Informationen vorgenommen wurden, nur Versionsnummern. Wurde erkannt, dass ein Knoten nicht aktuell ist, werden die Replikat-Bezeichner übertragen. Replikat-Bezeichner werden nur übertragen, wenn dies tatsächlich notwendig ist.

4.5 Austausch von Replikat-Daten

Um Replikate konvergieren zu lassen, wird ein Algorithmus benötigt, welcher entstehende Operationen beziehungsweise δ-CRDT-Zustände an andere Knoten propagiert. Der Algorithmus muss tolerant gegen unzuverlässige Kommunikation sein. Ferner soll der Algorithmus möglichst wenig Nachrichten senden und somit den Kommunikationsaufwand gering halten.

Zur Synchronisation von Replikat-Daten wird ein Push-Verfahren verwendet. Wenn lokal Operationen (δ-CRDT-Zustände) anfallen, werden diese in einen lokalen Sendebuffer geschrieben und in einem konfigurierbaren Intervall an andere interessierte Knoten propagiert. Der Sendebuffer erlaubt es, Operationen zusammenzufassen. Knoten sind dann interessiert, wenn sie ebenfalls eine lokale Instanz des Replikats halten. Weiterhin senden die Knoten periodisch vollständige Zustände anstelle von δ-Zuständen, damit ein Ausgleich von eventuell verlorenen Informationen, bedingt durch unzuverlässige Kommunikation, stattfinden kann.
Bevor tatsächlich Replikat-Daten zwischen Knoten ausgetauscht werden, werden diese gebuffert. Dadurch werden zwischen zwei Knoten in einem Buffer-Intervall nur maximal einmal Daten versendet.

4.6 Robuste Ausführung von Operationen

Erfolgreiche Operationen auf einem Replikat bedeuten nicht automatisch, dass diese auch im verteilten System propagiert sind. Es ist denkbar, dass eine lokale Operation auf einem Replikat erfolgreich durchgeführt wurde, der Knoten aber abstürzt, bevor die Operation an andere Knoten propagiert werden konnte. Die Operation galt als erfolgreich, ist aber trotzdem verloren gegangen. Um dieses Problem zu reduzieren ist es möglich, verschieden starke Lese-/Schreib- Zugriffe zu ermöglichen. Eine Operation wird dabei nicht nur lokal angewendet, sondern auch auf anderen Knoten. Damit ist die Operation erst erfolgreich, wenn die Operation auch auf einer bestimmten Anzahl an anderen Knoten erfolgreich war. Die Wahl eines starken Zugriffs hat dabei Auswirkungen auf die Latenz der Operation und damit auf die Verfügbarkeit. Da ein asynchrones, unzuverlässiges Kommunikationsmodell diskutiert wird, werden zur Fehlererkennung Timeouts benötigt.
Es ist außerdem möglich, sich überlappende Lese- und Schreib- Zugriffe zu wählen. Dies führt dazu, dass garantiert aktuelle Daten gelesen werden.
Im Folgenden werden verschieden starke Lese- und Schreib- Zugriffe und ihre Auswirkungen beschrieben.

Write-/Read-Local Die schwächste Form des Zugriffs ist das lokale Schreiben oder Lesen von einem Replik. Es wird keine Synchronisation zwischen Knoten benötigt, damit Operationen als erfolgreich gelten. Die Zugriffslatenz ist bei dieser Form sehr kurz, da nur die lokale Sicht auf die Daten gegeben ist. Die Wahrscheinlichkeit, dass veraltete Daten gelesen werden, ist relativ hoch (was jedoch durch das Konsistenzmodell erlaubt ist). Beim lokalen Schreiben auf ein Replikat ist nicht gewährleistet, dass eine lokal erfolgreiche Operation an den Rest des verteilten

Systems propagiert wird. Ein Beispiel für das Verlieren von Operationen ist ein Szenario, indem ein Knoten kurz nach einer lokal erfolgreichen Operation seine Netzwerkverbindung verliert oder abstürzt.

Write-/Read-All Die stärkste Form ist das Schreiben auf allen Replikaten im verteilten System. Eine Operation wird dabei auf allen Knoten ausgeführt und ist erst erfolgreich, wenn alle Knoten die Operation ausgeführt haben. Diese starke Form des Zugriffs eliminiert die Wahrscheinlichkeit, dass eine Operation verloren geht. Die Zugriffslatenzen sind allerdings im Vergleich zu denen der anderen Zugriffsformen eher schlecht, da Operationen länger benötigen. Lesende Operationen beinhalten die Sichten aller Knoten. Damit ist die Chance auf das Lesen von veralteten Daten reduziert. Die Zugriffszeit hängt stark von der Größe und der tatsächlichen Entfernung einzelner Knoten des verteilten Systems ab.

Write-/Read-Majority Der Kompromiss zwischen *Write-/Read-All* und *Write-/Read-Local* ist *Write-/Read-Majority*. Hierbei wird eine Operation auf einer Mehrheit von Knoten des verteilten Systems aufgeführt, bevor sie als erfolgreich gilt. Diese Form des Zugriffs bietet hohe Sicherheit gegen das Verlieren von Operationen. Damit es bei *Write-/Read-Majortiy* zu Verlust von Operationen kommt, müssten nach einer erfolgreichen Operation alle Knoten, die die Operation ausgeführt haben, abstürzen. Lesende Operationen bieten eine geringere Chance, veraltete Daten zu lesen, da die Sichten von einer Mehrheit an Knoten eingeholt wird. Die Mehrheit bestimmt sich dabei durch $N/2 + 1$, wobei N die Anzahl an Knoten im verteilten System ist. Die Zugriffslatenz wird im direkten Vergleich zu *Write-/Read-All* reduziert.

Write-/Read-K Mit *Write-/Read-K* ist es möglich, von einer beliebigen Anzahl an Knoten zu lesen, beziehungsweise auf diese zu schreiben. Der Anwendungsentwickler hat dabei die volle Kontrolle darüber, auf wie vielen Knoten im verteilten System eine Operation ausgeführt werden soll.

Timeout von Operationen Damit Operationen, welche auch auf anderen Knoten ausgeführt werden, als fehlgeschlagen erkannt werden können, ist es notwendig, Timeouts zu verwenden. Ein Timeout beschreibt die Zeit, die vergehen darf, bis die Operation beendet sein muss. Damit Operationen zuverlässig als fehlgeschlagen erkannt werden können, sollte ein Timeout so kurz wie möglich gewählt sein. Dabei muss er aber so groß wie notwendig sein, damit nur wirklich fehlgeschlagene Operationen erkannt werden. Es gibt verschiedene Verfahren zum Festlegen von Timeouts, darunter statische und dynamische Verfahren. Ein statisches Verfahren zum Bestimmen von Timeouts ist das Festlegen von festen Timeouts, beispielsweise

zwei Sekunden. Statische Timeouts erfordern Wissen über den Anwendungsfall und die Gegebenheiten des Systems, sowohl lokal (beispielsweise Auslastung des lokalen Knotens) als auch verteilt (beispielsweise Round-Trip-Time). Alleine durch diese beiden Variablen ist es nicht möglich, einen zuverlässigen statischen Timeout zu definieren, welcher möglichst kurz und dabei so lange wie notwendig ist. Dynamische Verfahren beziehen mögliche Variablen wie Round-Trip-Time (RTT) und Auslastung der Knoten mit ein. Dynamische Verfahren können jedoch nicht Eigenheiten, welche durch den Anwendungsfall bedingt werden, mit einbeziehen.

Reagieren auf Timeouts Wird ein Timeout ausgelöst, so gilt eine Operation als fehlgeschlagen. In diesem Fall wird der Anwendungsentwickler benachrichtigt und kann den Fehler behandeln. Es ist denkbar, dass der Fehler einfach geloggt oder ein anwendungsspezifisches Verfahren gewählt wird. Es ist nicht notwendig, die Operation erneut auszuführen, da sie lokal durchgeführt wurde und somit nicht fehlschlagen konnte und deshalb später an alle anderen Knoten propagiert wird.

4.6.1 Diskussion

Da CRDTs im Bereich der AP-Systeme (Abschnitt 2.2) verwendet werden, ist der standardmäßige Zugriff auf Replikate *Write-/Read-Local*. Eine stärkere Form des Zugriffs beschneidet CRDTs in ihren Eigenschaften (Verfügbarkeit & Partitionstoleranz). Für den lokalen Zugriff ist es nicht notwendig, einen Timeout zu spezifizieren. Soll für einzelne lesende Operationen explizit verhindert werden, dass veraltete Daten gelesen werden, lässt sich ein starkes Zugriffsmodell wählen. Dabei muss vom Anwendungsentwickler ein Timeout angegeben werden, welcher zum Anwendungsfall passt. Im Fehlerfall kann der Anwendungsentwickler auf Timeouts reagieren.

4.7 Lebenszeit von Replikaten

Die Lebenszeit von Replikaten bezieht sich auf den Zeitabschnitt beginnend mit der Initialisierung bis zum Entfernen von nicht mehr verwendeten Replikaten. Abschnitt 4.7.1 diskutiert die Initialisierung und die damit verbundenen Implikationen. Abschnitt 4.7.2 behandelt das Entfernen von Replikaten.

4.7.1 Initialisierung

Es ist möglich, ein Replikat auf allen (bekannten) Knoten zu initialisieren, sobald das Replikat auf irgendeinem (bekannten) Knoten zum ersten Mal verwendet wurde. Alternativ wäre eine Lazy-Initialisierung möglich. Hierbei wird ein Replikat nur auf einem Knoten initialisiert, sobald

es in dem Knoten verwendet wird. Dies hat bereits Auswirkungen auf die Ausfallsicherheit und damit auf die Datensicherheit. Das Ressourcen-Management innerhalb der Knoten wird durch die Entscheidung ebenfalls beeinflusst.

Initialisierung auf allen Knoten Eine Initialisierung auf allen bekannten Knoten führt zu einer hohe Datensicherheit, da eine hohe Redundanz von Replikaten vorliegt. Für das Ressourcen-Management innerhalb der einzelnen Knoten kann dies jedoch von Nachteil sein, da ein Knoten alle Replikate halten muss. Außerdem entsteht mehr Last auf Netzwerk-Ebene, da Zustände zu allen Knoten propagiert werden müssen.

Lazy-Initialisierung Bei dieser Methode findet eine Initialisierung auf Knoten erst dann statt, wenn ein Replikat lokal verwendet wird. Dies hat den Vorteil, dass nur Ressourcen für Replikate verwendet werden, welche tatsächlich gebraucht werden. Es reduziert jedoch die Datensicherheit, da weniger Replikate redundant gehalten werden.

Diskussion

Das Modul verwendet Lazy-Initialisierung der Replikate, da es so auch auf Knoten verwendet werden kann, die in ihren Ressourcen eher limitiert sind. Es wäre jedoch denkbar, einzelne CAF-Knoten so zu konfigurieren, sodass diese alle Replikate im verteilten System replizieren.

4.7.2 Entfernen von Replikaten

Wird ein bestimmtes Replikat lokal von keinem Aktor mehr verwendet, könnte das Replikat von dem Knoten gelöscht werden. Dieser Knoten muss damit keine Nachrichten mehr verarbeiten, die zu diesem Replikat gehören. Andere Knoten müssen Änderungen nicht mehr an den Knoten verteilen. Ein Problem dabei stellt jedoch der eventuelle Verlust an Datensicherheit dar. Wenn alle Aktoren im verteilten System das Replikat nicht mehr verwenden und es somit von allen Knoten entfernt wird, kann dies zu einem problematischen Datenverlust führen. In verteilten Systemen mit einer hohen Fluktuation an Replikaten sollten nicht mehr benötigte Replikate entfernt werden, da sich sonst ein immer größerer Zustand akkumuliert. Es muss dabei jedoch sichergestellt sein, dass nur Replikate entfernt werden, welche für die Logik der Anwendung nicht mehr benötigt werden und deren Entfernung somit sicher ist.

Automatisches Entfernen

Das automatische Löschen von Replikaten führt dazu, dass Ressourcen, welche nicht mehr verwendet werden, wieder freigegeben werden. Mit dem Löschen von Replikaten von Knoten geht jedoch ein Teil der Redundanz und damit der Datensicherheit verloren. Ein Replikat wird demnach automatisch entfernt, wenn kein lokaler Benutzer-Aktor mehr an dem Replikat interessiert ist (Das Replikat hat keine Abonnenten mehr).

Sofortiges Entfernen Das sofortige Entfernen von lokal nicht mehr verwendeten Replikaten stellt Ressourcen in aggressiver Weise wieder zur Verfügung. Als Konsequenz ergibt sich der Nachteil, dass der Zustand verloren ist, sollten alle Knoten das Replikat entfernt haben. Wird nun erneut auf ein Replikat mit diesem Bezeichner zugegriffen, ist das Replikat leer (frisch initialisiert).

Entfernen nach Timeout Eine Alternative zum sofortigen Entfernen stellt das Entfernen nach Timeout dar. Wenn lokal kein Aktor mehr ein bestimmtes Replikat verwendet, startet ein Timer, welcher nach Ablauf das Replikat löscht, sofern es in der Zwischenzeit nicht wieder verwendet wird. Die Wahrscheinlichkeit von zu früh entfernten Replikaten wird somit reduziert. Das Problem des globalen Entfernen von Replikaten, was Datenverlust zur Folge hat, bleibt bestehen.
Das Problem mit Timeout-Verfahren ist, dass sich kein Anwendungsfall unabhängiger Timeout definieren lässt. Selbst dem Anwendungsentwickler ist es oft nicht möglich einen sinnvollen Timeout festzulegen.

Manuelles Löschen

Das manuelle Löschen von Replikaten durch den Anwendungsentwickler löst das Problem des falschen beziehungsweise zu frühen Löschens von Replikaten. Grund hierfür ist, dass der Entwickler nur Replikate freigibt, die die Anwendungslogik nicht mehr benötigt.

Diskussion

Es lässt sich nicht für alle Anwendungsfälle sagen, dass es sinnvoll wäre, Daten zu löschen, die nicht mehr verwendet werden. Deshalb muss der Anwendungsentwickler das Entfernen von Replikaten in Knoten übernehmen. Dabei kann er abhängig von dem Anwendungsfall entscheiden, wann ein Replikat nicht mehr benötigt wird und es löschen. Der Entwickler muss

falsches beziehungsweise zu frühes Entfernen von Replikaten ausschließen können, da sonst ein Datenverlust droht.

4.8 Software-Design

Dieser Abschnitt stellt die aus dem Design resultierenden Software-Entitäten vor. In Abschnitt 4.8.1 findet sich ein UML-Klassen-Diagramm mit den wichtigsten Klassen. Abschnitt 4.8.2 zeigt den internen Ablauf und das Zusammenspiel der einzelnen Software-Entitäten. Ein API-Entwurf findet sich in Abschnitt 4.8.3. Abschließend zeigt Abschnitt 4.8.4 die Schnittstelle, mit deren Hilfe Entwickler eigene CRDTs mit dem CAF-Modul verwenden können.

4.8.1 Klassendiagramm

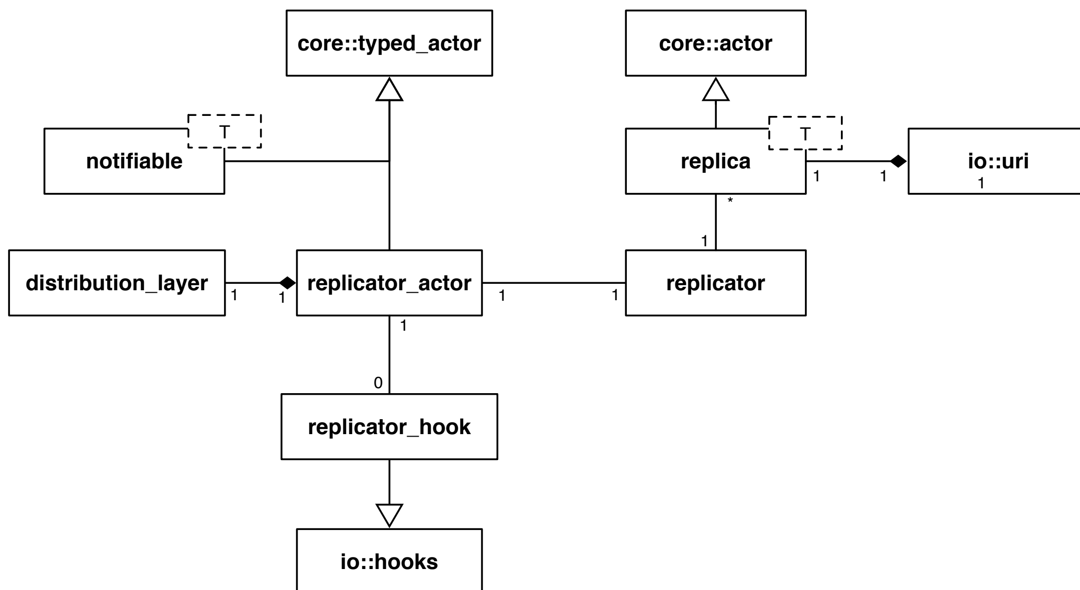

Abbildung 4.5: Wichtige Klasse im CAF-CRDT-Modul

Abbildung 4.5 zeigt ein UML-Klassendiagramm mit den Hauptklassen und ihren Beziehungen. Klassen, die mit einem Präfix annotiert sind, kommen entweder aus dem CAF-Core-Modul oder aus dem CAF-IO-Modul. Klassen ohne Präfix sind im CAF-CRDT-Modul definiert. Im Folgenden werden die Klassen sowie ihre Zusammenhänge beschrieben.

typed_actor Typisierte Aktoren erlauben es, die Verwendung von Aktoren zur Übersetzungszeit zu prüfen. Dabei wird geprüft, ob auch nur Nachrichten an den *typed_actor* gesendet werden, welche auch von ihm unterstützt werden.

actor Die Klasse Aktor repräsentiert dynamische (nicht typisierte) Aktoren. Diese Art von Aktoren benötigen keine Interface-Definition und werden daher nicht zur Übersetzungszeit überprüft. Diese Aktoren können jede Art von Nachricht empfangen.

hooks Das CAF-IO-Modul stellt verschiedene Hooks bereit. Hooks erlauben es, Funktionen aufzurufen, wenn bestimmte Ereignisse passieren. Das CAF-IO-Modul stellt Hooks bereit, welche ausgeführt werden, wenn neue Verbindungen zu Knoten hergestellt oder Verbindungen zu Knoten verloren gegangen sind.

uri Die URI-Klasse aus dem CAF-IO-Modul implementiert und parst URIs. Diese Klasse wird benötigt, da Replikat-Bezeichner als URIs repräsentiert werden.

notifiable Typisierte Aktor-Interfaces sind erweiterbar, deshalb wird mit *notifiable* ein typisiertes Interface bereitgestellt, welches es erlaubt, Replikate in vom Anwendungsentwickler definierten *typed_actors* zu verwenden. Um ein Interface mit *notifiable* zu erweitern, ist ein Template-Parameter T notwendig, welcher ein CRDT sein muss. Ein typisiertes Interface, welches mit notifiable erweitert wurde, fordert einen Nachrichten-Handler mit der Signatur (*notify_atom,* T) ein. Dieser Nachrichten-Handler wird ausgeführt, sobald der Aktor ein Update zu einem Replikat des Typs T erhält.

replica Die Klasse *replica* ist ein Aktor, welcher eine Instanz T hält (T ist ein CRDT). Sie ist dafür zuständig, alle Replikate derselben Art, welche in Zuständen von Aktoren gehalten werden, zu synchronisieren. Dies geschieht über eine Publish/Subscribe-API. Aktoren, welche ein Replikat in ihrem Zustand halten, melden sich bei der passenden *replica*-Instanz an und bekommen von dort an alle Änderungen auf dem Zustand per Nachricht (*notify_atom*, T).

replicator_actor Der Replikator-Aktor stellt die asynchrone, nachrichtenbasierte API für Aktoren bereit. Er empfängt Zustandsänderungen von fremden Knoten sowie von Aktoren, welche ein Replikat in ihrem Zustand halten. Des Weiteren ist er dafür zuständig, Zustandsänderungen an die passende *replica*-Instanz zu delegieren. *replica*-Instanzen sind dem *replicator_actor* als *childs* bekannt und werden ausschließlich von ihm instanziiert. Außerdem ist er dafür zuständig, alle Knoten konvergent zu halten. Dabei reagiert er auch auf Ereignisse

wie zum Beispiel das Verlieren von Verbindungen zu Knoten oder dem Neuentdecken von Knoten. Über die Ereignisse wird er von der Klasse *replicator_hook* informiert, welche spezielle Nachrichten versendet.

replicator Der Replicator beinhaltet Initialisierungs- und Destruktionscode für das Modul. Er stellt eine alternative blockierende API für Subscribe/Unsubscribe bereit, falls das nachrichtenbasierte Interface nicht verwendet werden kann. Dies ist zum Beispiel in nicht aktorbasierten Softwareteilen der Fall.

replicator_hook Ein Replicator-Hook reagiert auf Events des I/O-Moduls, darunter (1) das Entdecken von neuen Knoten und (2) das Verlieren von Verbindungen zu anderen Knoten. Sobald einer dieser Hooks ausgelöst wird, wird eine Nachricht an den *replicator_actor* versendet. Auf die Events reagiert er per Hook, was bedeutet, dass er Nachrichten an den *replicator_actor* sendet.

distribution_layer Diese Klasse implementiert den Algorithmus, um Knoten konvergent zu halten, was Abschnitt 4.5 bereits beschreibt. Außerdem implementiert die Klasse den austausch von Meta-Informationen, was bereits in Abschnitt 4.4 beschrieben ist. Innerhalb des *replicator_actor* wird eine Instanz dieser Klasse verwendet.

4.8.2 Nachrichtenfluss im CAF-CRDT-Modul

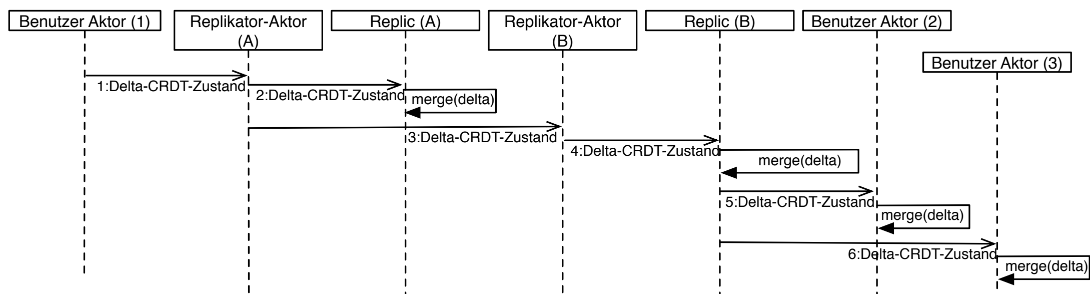

Abbildung 4.6: Kommunikationswege zwischen verschiedenen Software-Entitäten

Abbildung 4.6 zeigt die Kommunikationswege unter den einzelnen Software-Entitäten. Dabei sind zwei CAF-Knoten involviert, sowie drei Benutzer-Aktoren, die an den Replikat-Daten interessiert sind. Benutzer-Aktor (1) ändert das Replikat, welches sich in seinem lokalen Aktor-Zustand befindet. Daraufhin wird ein δ-CRDT-Zustand generiert, welche an den Replikator-Aktor (A) gesendet wird (1). Der Replikator-Aktor (A) leitet den δ-CRDT-Zustand an den lokal

zuständigen Replica-Aktor weiter (2). Der Replica-Aktor (A) empfängt den Zustand und führt ihn mit seinem Zustand zusammen. Der Replikator-Aktor (A) sendet den δ-CRDT-Zustand außerdem an den Replikator-Aktor (B), welcher in einem anderen interessierten Knoten läuft (3). Der Replikator-Aktor (B) sucht den lokal passenden Replic-Aktor (B) und leitet die Nachricht an diesen weiter (4). Der Replic-Aktor (B) führt den empfangenen Zustand mit seinem Zustand zusammen. Entsteht dabei ein δ-CRDT-Zustand, der nicht leer ist, so wird der empfangene δ-CRDT-Zustand an alle lokal interessierten Benutzer-Aktoren versendet (5 & 6).

4.8.3 Entwurf einer Programmierschnittstelle (API)

Damit Aktoren asynchron auf die Schnittstelle das CAF-Moduls zugreifen können, muss das Interface nachrichtenbasiert abgebildet sein. Zur Abbildung der asynchronen Interfaces ist der bereits vorgestellte *replicator_actor* zuständig.
Änderungen an Replikaten sollen keine zusätzlichen Aktionen von Aktoren erfordern. Das heißt, dass Änderungen im Hintergrund an die Schnittstelle des CAF-Moduls geleitet werden. Der folgende Abschnitt stellt verschiedene API-Beispiele vor. Zu Beginn wird die Konfiguration des Moduls gezeigt. Außerdem wird auf die Verwendung von typisierten und dynamischen Aktoren mit CRDTs eingegangen.
Eine Definition der öffentlich zugänglichen Schnittstelle des *replicator_actors* findet sich in Abschnitt 5.3.2.

Auflistung 4.1: Konfiguration eines Aktor-Systems, welches das CAF-CRDT-Modul verwendet

```
1  struct config : public crdt_config {
2    using time_resolution = std::chrono::seconds;
3    config() {
4      add_crdt<gcounter<int>>("gcounter<int>");
5      set_notify_interval(time_resolution(1));
6      set_flush_interval(time_resolution(10));
7      set_refresh_ids_interval(time_resolution(10));
8      set_state_interval(time_resolution(300));
9    }
10 };
```

Konfiguration des Aktor-Systems

Auflistung 4.1 zeigt die möglichen Konfigurations-Parameter des CAF-CRDT-Moduls. Um CRDTs der Laufzeit bekannt zu machen, wird die Funktion *add_crdt* verwendet (Zeile 4).

Diese nimmt als Template Parameter die CRDT-Klasse entgegen. Des Weiteren wird ein Typ-Bezeichner als String übergeben. Es ist außerdem möglich, ein Intervall für die Benachrichtigung von lokalen Aktoren (Abonnenten) einzustellen (Zeile 5). Die Zeit, welche verwendet wird, um anfallende Änderungen (Updates) zu buffern, kann mit *set_flush_interval* konfiguriert werden (Zeile 6). Das Intervall für die Synchronisation von Replikat-Bezeichnern kann mit `set_refresh_ids_interval` gesetzt werden (Zeile 7). Das periodische Versenden von vollständigen Replikaten (anstelle von δ-Zuständen) kann mit `set_state_interval` konfiguriert werden (Zeile 8). Da dies das Datenaufkommen wesentlich erhöht und nur selten benötigt wird, ist ein möglichst hoher Wert empfohlen.

Auflistung 4.2: Verwendung eines Replikats innerhalb eines nicht typisierten Aktors

```
1  struct state {
2    state(event_based_actor* self)
3        : crdt(self, "gset<int>://set") { /* nop */ }
4    gset<int> crdt;
5  };
6
7  void actor_fun(stateful_actor<state>* self) {
8    self->become(
9      [=](int value) {
10       self->state.crdt.insert(value);
11     },
12     [=](notify_atom, const gset<int>& other) {
13       self->state.crdt.merge(other);
14     }
15   );
16 }
```

Nicht Typisierte Aktoren

Auflistung 4.2 zeigt einen nicht typisierten funktionsbasierten Aktor. Dieser hält eine Replikat vom Typ `gset<int>` in seinem Zustand (Zeile 4). Mit der Initialisierung des Zustands (Zeile 3) meldet sich der Aktor implizit für Änderungen für dieses Replikat an. Dies bedeutet, dass der Aktor Nachrichten empfängt, welche Änderungen an dem Zustand enthalten, jedoch von anderen Aktoren erzeugt wurden. Der Aktor reagiert auf Nachrichten vom Typ `int` mit dem Hinzufügen des empfangenen *Integers* in das *GSet* (Zeile 9-11). Der Aufruf `state_->insert(value)` ändert nicht nur den lokalen im Aktor befindlichen Zustand, sondern sendet

auch ein bei der Operation entstehendes δ zum *replicator_actor*. Außerdem reagiert der Aktor auf Nachrichten, welche ein `notify_atom` und eine Replik-Instanz beinhalten (Zeile 12-14). Empfangene Replik-Instanzen werden mit dem lokal gehaltenen Zustand zusammengeführt (*merge*).

Auflistung 4.3: Beispiel eines typisierten Aktors, welcher ein Replikat innerhalb seines Zustands hält

```
1  using interface = typed_actor<...>::extends_with<
2    notifiable<gcounter<int>>>;
3
4  class class_actor : public interface::base {
5  public:
6    class_actor(actor_config& cfg)
7      : interface::base(cfg),
8        state_(this, "gcounter<int>://class_actor/count") {
9      state_.increment_by(1);
10   }
11
12 protected:
13   interface::behavior_type make_behavior() override {
14     return {
15       // ...
16       [&](notify_atom, const gcounter<int>& other) {
17         state_.merge(other);
18       },
19       // ...
20     };
21   }
22
23 private:
24   gcounter<int> state_;
25 };
```

Typisierte Aktoren

Das Beispiel aus Abbildung 4.3 zeigt, wie Instanziierungen von `class_actor` global gezählt werden können. Jede Instanz zählt dabei einen Counter-CRDT hoch. Zu sehen ist ein typisierter klassenbasierter Aktor. Der Aktor hält in seinem Zustand ein Replikat vom Typ `gcounter<int>` (Zeile 24). Es ist notwendig, den Replik-Zustand im Konstruktor der Klasse zu initialisieren

(Zeile 8). Dabei wird der Aktor automatisch als Abonnent zu diesem Replikat hinzugefügt. Im Konstruktor selbst wird der Zähler um eins erhöht (Zeile 9). Die Operation wird im Hintergrund propagiert. Der Aktor ist ein stark typisierter Aktor, dessen Interface in Zeile 1-2 definiert ist. Es ist ein bestehendes typisiertes Aktor-Interface zu sehen (`typed_actor<...>`), welches für die Verwendung von CRDTs erweitert wird. Dazu wird `extebds_with` verwendet. Die Implementierung des typisierten Aktors erbt von `interface::base`, was die erweiterte Interface-Definition ist.

Auflistung 4.4: Operation auf einem Replikat ausführen, ohne Zustand im Aktor zu halten

```
1   // ...
2   auto repl = system.replicator().actor_handle();
3   scoped_actor self{system};
4   gset<int> set;
5   set.insert(5);
6   auto req = self->request(repl, seconds(2), write_all_atom::value,
7                            "gset<int>://visitors",
8                            make_message(set));
9   req.receive(
10    [&](write_succeed_atom) { /* Operation erfolgreich */ },
11    [&](error&) { /* Fehlerbehandlung */ }
12  );
13  // ...
14  req = self->request(repl, seconds(2), read_all_atom::value,
15                      "gset<int>://visitors");
16  req.receive(
17    [&](read_succeed_atom, const gset<int>& result) {
18      // Operation erfolgreich - Ergebnis in result
19    },
20    [&](error&) { /* Fehlerbehandlung */ }
21  );
```

Ausführen einzelner Operationen

Auflistung 4.4 zeigt, wie einzelne Operationen auf einem Replikat ausgeführt werden können, ohne den ganzen Zustand zu halten oder sich für ein Replikat anzumelden. Dazu wird das Message-Passing-Interface direkt verwendet. Zu sehen ist ein kurzlebiger `scoped_actor` (Zeile 3), welcher auf ein leeres `gset<int>` (Zeile 4) die Operation `insert(5)` ausführt (Zeile 5). Diese Operation wird in diesem Schritt nicht auf andere Knoten oder an andere lokale Aktoren

repliziert, da nicht klar ist, zu welchem Replikat die *GSet*-Instanz gehört (es ist kein Replikat-Bezeichner angegeben). Im nächsten Schritt versendet der `scoped_actor` das *GSet* (was als δ-Zustand betrachtet werden kann) an den lokalen `replicator_actor` als `request` (Zeile 6-8). Hierfür ist es notwendig, einen Timeout zu spezifizieren sowie eine *write/read*-Regel. Weiterhin muss ein Replikat-Bezeichner angegeben werden sowie eine Nachricht, die den Replik-Zustand enthält. Der `scoped_actor` wartet nun mit `receive` auf Abschluss der Operation (Zeile 9-12). Es ist möglich, dass die Operation (1) mit Empfang eines *write_succeed_atom* als erfolgreich gilt (Zeile 10); oder (2) mit Empfang eines *error* als fehlgeschlagen (Zeile 11). Ein *error* tritt auf, sofern die Operation nach dem spezifizierten Timeout (in diesem Fall zwei Sekunden) nicht beendet ist.
Es ist außerdem zu sehen, wie eine lesende Operation ausgeführt wird. Dazu wird erneut ein `request` an den *replicator_actor* gesendet (Zeile 14-15). Der `request` besteht diesmal aus einem `read_all_atom` sowie dem Replikat-Bezeichner. Anschließend wird mit Hilfe von `receive` auf Beendigung des Vorgangs gewartet (Zeile 16), dazu sind erneut zwei Nachrichtenhandler spezifiziert. Der erste Nachrichten-Handler erwartet ein `read_succeed_atom` sowie eine Instanz des abgefragten Replikats (Zeile 17). Kann der Lesevorgang nicht in der vorgegebenen Zeit beendet werden, wird ein Fehler zurückgegeben (Zeile 20).

4.8.4 Erweiterbarkeit um eigene CRDTs

Es ist möglich, eigens definierte CRDTs mit dem CAF-Modul zu verwenden. Dazu muss der CRDT nur die Basis-Klasse `base_datatype` implementieren. Vom Anwendungsentwickler müssen folgende Funktionen in der Klasse des neuen CRDTs (Beispiel: `own_class`) bereitgestellt werden:

- `template <ActorHandle> own_class(ActorType&& owner, std::string id)` - Dieser Konstruktor wird benötigt, damit beliebige Arten von Benutzer-Aktoren die Klasse initialisieren können. Die Argumente müssen lediglich an die Basisklasse (`base_datatype`) weitergereicht werden.

- `own_class merge(const own_class& other)` - Diese Funktion führt zwei Zustände (`this` und `other`) zusammen. Geändert wird lediglich der Zustand in `this`. Die Funktion muss einen neuen δ-Zustand zurückgeben, welcher die Unterschiede beider Zustände darstellt.

- `bool empty() const` - Um zu prüfen, ob ein generierter δ-Zustand leer ist, wird die `empty`-Funktion benötigt. Ein leerer δ-Zustand wird generiert, wenn die mit der `merge`-Funktion zusammengeführten Instanzen den gleichen Zustand haben.

Ein vollständiges Beispiel eines eigens implementierten CRDT findet sich in Abschnitt 5.5.

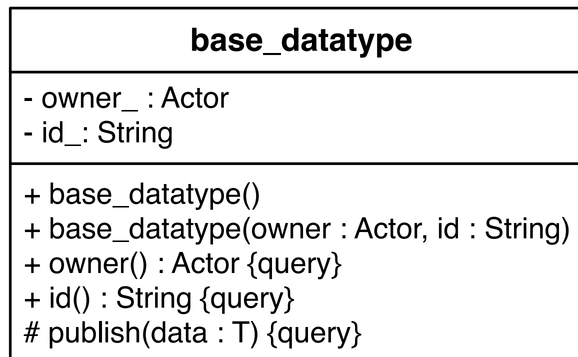

Abbildung 4.7: UML-Klassendiagramm von *base_datatype*

Abbildung 4.7 zeigt die Klasse `base_datatype`. Von dieser Klasse abgeleitete Klassen haben Zugriff auf den Replikat-Bezeichner sowie den Besitzer (`owner`) dieses Zustands. Wird in der abgeleiteten Klasse eine Änderung am Zustand vorgenommen, so muss ein δ-Zustand erstellt werden, welcher mit Hilfe der *publish*-Funktion an den Replicator weitergeleitet wird. Oft benötigen CRDTs einen einzigartigen Bezeichner für Änderungen, dazu kann das *Actor-Handle* des besitzenden Aktors verwendet werden. Um Zugriff auf das *Actor-Handle* zu erlangen, steht die Funktion `owner` bereit. Um Zugriff auf den Replikat-Bezeichner innerhalb des CRDTs zu erhalten, kann die Funktion `id` verwendet werden.

5 Implementierung

Dieses Kapitel stellt die Umsetzung des in Kapitel 4 beschriebenen Designs vor. Die Beschreibung der Implementierung teilt sich in folgende Bereiche auf: (1) Konfiguration des CAF-CRDT-Moduls und Vorstellung der konfigurierbaren Parameter. (2) Interne Verwaltung und Benutzung von Replikaten. (3) Beschreibung der zentralen Verwaltungsinstanz (*Replicator*) des CAF-CRDT-Moduls. (4) Beschreibung des „Distribution-Layers“, welches δ-CRDT-Zustände zwischen den einzelnen CAF-Knoten propagiert.

Der Sourcecode des CAF-Hauptmoduls[1] sowie der Code des CAF-CRDT-Moduls[2] sind auf GitHub als Open Source unter Boost- und BSD-Lizenz veröffentlicht. Für CAF wird ein C++11 fähiger Compiler benötigt. Von CAF unterstützte Betriebssysteme sind Linux, FreeBSD, macOS (ehemals OSX) und Windows.

Die folgenden Abschnitte beschreiben die einzelnen Teile der Implementierung. Abschnitt 5.1 beschreibt die Konfiguration des CAF-CRDT-Moduls. In Abschnitt 5.2 finden sich die interne Verwaltung von Replikaten und deren Benutzung. Die zentrale Verwaltung des CAF-CRDT-Moduls ist in Abschnitt 5.3 beschrieben und adressiert den *Replicator*. Die Verteilung von δ-CRDT-Zuständen erläutert Abschnitt 5.4. Abschitt 5.5 zeigt anhand des *PN-Counters* die Implementierung eigener CRDTs.

[1] https://github.com/actor-framework
[2] https://github.com/Hamdor/libcaf_crdt

5.1 Modul Konfiguration

Auflistung 5.1: Konfiguration des CRDT-Moduls

```
1  class crdt_config : public actor_system_config {
2  public:
3    crdt_config() : actor_system_config() {
4      load<crdt::replicator>();
5    }
6    template <class Type>
7    actor_system_config& add_crdt(const std::string& name) {
8      add_message_type<Type>(name);
9      add_actor_type<crdt::detail::replica<Type>,
10                    const uri&, const size_t&>(name);
11     return *this;
12   }
13   template <class Interval>
14   actor_system_config& set_flush_interval(Interval interval) { /*...*/ }
15   template <class Interval>
16   actor_system_config& set_notify_interval(Interval interval) { /*...*/ }
17   template <class Interval>
18   actor_system_config& set_state_interval(Interval interval) { /*...*/ }
19   template <class Interval>
20   actor_system_config& set_refresh_ids_interval(Interval interval) { /*
         ...*/ }
21 };
```

Auflistung 5.1 zeigt die Konfigurationsklasse `crdt_config` des CRDT-Moduls, die von der Klasse `actor_system_config` erbt (Zeile 1). Im Konstruktor der Klasse (Zeile 3) wird das CRDT-Modul geladen (Zeile 4). Damit kann CAF später alle Funktionen aufrufen, welche notwendig sind, um das CRDT-Modul zu initialisieren. Es stehen außerdem verschiedene Funktionen zur Verfügung, mit deren Hilfe sich das CRDT-Modul konfigurieren lässt.
Die Auflistung verzichtet teilweise auf die Rümpfe der Funktionen, da dort nur einfache Variablenzuweisungen passieren. Die erste Funktion `add_crdt` erlaubt es, CRDTs beim CRDT-Modul anzumelden (Zeile 6-7). Dabei nimmt sie einen beliebigen CRDT-Typ als Template-Parameter `Type` entgegen. Zusätzlich muss ein `string` übergeben werden, welcher einen Namen des CRDTs darstellt (Zeile 8). Dieser Name wird im Typ-System von CAF benötigt, um ihn als dynamische Laufzeitinformation in CAF verwenden zu können. Eine Anmeldung beim Typ-

System ist notwendig, um den Typen serialisieren und deserialiseren zu können. Des Weiteren wird mit dem Aufruf von `add_actor_type` ein Aktor vom Typ `replica<Type>` dynamisch per Name erzeugbar gemacht (Zeile 9-10). Dies ist notwendig, da beim späteren Spawnen von `replica`-Aktoren kein Template-Parameter für den internen CRDT mehr verfügbar ist, welcher aber für die Spezialisierung von `replica` notwendig wäre.
Es können verschiedene Intervalle konfiguriert werden. Für alle Funktionen, die Intervalle konfigurieren, gilt jedoch, dass ein Duration-Typ aus C++11 übergeben werden muss. Dies erlaubt das Verwenden von beispielsweise `std::chrono::seconds`, `std::chrono::milliseconds`, `std::chrono::minutes`.
Die verschiedenen Intervalle, welche sich konfigurieren lassen, sind:

1. Flushen des Sendebuffers für δ-CRDT-Zustände (Zeile 14), um Operationen zusammengefasst versenden zu können (Abschnitt 4.5).
2. Benachrichtigung von Abonnenten bei Änderungen an Replikaten (Zeile 16). Dabei werden Änderungen nicht sofort an Abonnenten verteilt, sondern in Intervallen. Dadurch werden Operationen, die zeitnah entstanden sind, zusammengefasst ausgeliefert.
3. Flushen vollständiger CRDT-Zustände, um eventuell verlorene Updates auszugleichen (Zeile 18) (Abschnitt 4.5).
4. Austauschen von Replikat-Bezeichnern zwischen Knoten (Zeile 20) (Abschnitt 4.4)

5.2 Verwaltung von Replikat-Daten

Es ist notwendig, pro interessiertem Knoten einen instanziierten Replikatszustand zu halten, da anhand dieses Zustands Replikat-Daten lokal und in anderen Knoten synchronisiert werden. Es ist nicht möglich, die Daten im zentralen *replicator*-Aktor (Abschnitt 5.3) zu halten, da die Typ-Informationen, die dazu notwendig wären, nicht verfügbar sind. Die Klasse *replica* übernimmt die separate Kapselung der Daten, da Typ-Informationen in diesem Aktor vorliegen. Sie ist außerdem dafür zuständig, Abonnenten ihres Replikats über Änderungen am Zustand zu informieren. Des Weiteren ist Funktionalität vorhanden, um einzelne Operationen auf Replik-Daten auszuführen, welche durch direktes Verwenden der Message-Passing API versendet werden.

Der folgende Abschnitt 5.2.1 erläutert den Zustand der *replica*-Klasse. Abschnitt 5.2.2 zeigt die Nachrichten-Handler für das An- und Abmelden von Informationen über Änderungen an Replikat-Daten. Abschnitt 5.2.3 beschreibt das Verhalten bei Empfangen von Updates.

Abschließend wird in Abschnitt 5.2.4 die direkte Verwendung der Message-Passing API für einzelne Operationen erläutert.

5.2.1 Zustand der *replica*-Klasse

Auflistung 5.2: Zustand eines Replica-Aktors

```
1  template <class T> // T ist ein CRDT
2  class replica : public event_based_actor {
3    // ...
4  private:
5    T cvrdt_;                            // Replikat-Zustand
6    T buffer_;                           // Benachrichtigungs-Buffer
7    uri id_;                             // Replikat-Bezeichner
8    size_t notify_interval_ms_;          // Benachrichtigungs-Intervall
9    std::unordered_set<actor> subs_;     // Abonnenten
10 };
```

Abbildung 5.2 zeigt die Instanzvariablen eines `replica`-Aktors. Die Klasse benötigt einen Template-Parameter `T`, welcher ein *CRDT* sein muss (Zeile 1). Zu sehen ist außerdem, dass die Klasse von `event_based_actor` abgeleitet ist und damit ein dynamisch, nicht typisierter Aktor ist (Zeile 2). Es ist es nicht möglich, den `replica`-Aktor als typisierten Aktor zu implementieren, da dieser dynamisch per Name erzeugt wird. Wäre der Aktor typisiert, müsste beim Erzeugen des Aktors neben dem Namen der genaue Typ des Aktors verfügbar sein. In diesem Anwendungsfall wird der Aktor aber gerade deshalb per Name erzeugt, weil der Typ des Aktors an der Stelle, wo dieser erzeugt wird, nicht verfügbar ist.
Der Aktor hält eine Instanz des *CRDTs* in der privaten Variable `cvrdt_` vom Typen `T` (Zeile 5). Außerdem beinhaltet die Klasse einen Benachrichtigungs-Buffer, welcher ebenfalls vom Typ `T` ist (Zeile 6). Jede Instanz von `replica` ist an einen Replikat-Bezeichner gebunden, dieser steht in der Variable `id_` vom Typen `uri` zur Verfügung (Zeile 7).
Das Intervall, in welchem der Benachrichtigungs-Buffer zu den Abonnenten versendet wird, ist in `notify_interval_ms_` gespeichert (Zeile 8). Zum Verwalten von Abonnenten steht in der Klasse eine Hash-Map über Aktoren `subs_` bereit (Zeile 9). Die Hash-Map bietet einen *best/avarage case* von $\mathcal{O}(1)$, der *worst case* liegt bei $\mathcal{O}(n)$. Die Klasse `actor`, welche als Schlüssel für die Hash-Map verwendet wird, besitzt eine Hash-Funktion mit möglichst wenig Kollisionen.

5.2.2 Verwalten von Abonnenten

Auflistung 5.3: An- und Abmelden an einem Replikat

```
1  [&](subscribe_atom) {
2    auto handle = actor_cast<actor>(current_sender());
3    if (!handle)
4      return;
5    subs_.emplace(handle);
6    // Aktuellen Zustand des Replikats an den neuen Abonnenten senden
7    if (!cvrdt_.empty())
8      send(handle, notify_atom::value, cvrdt_);
9  },
10 [&](unsubscribe_atom) {
11   auto handle = actor_cast<actor>(current_sender());
12   if (handle)
13     subs_.erase(handle);
14 }
```

Auflistung 5.3 zeigt die Nachrichten-Handler für das An- und Abmelden an Replikat-Daten. Ein Benutzer-Aktor, welcher über Änderungen an einem Replikat informiert werden möchte, sendet ein `subscribe_atom` an den *replicator*-Actor (dieser leitet die Nachricht an den passenden `replica`-Aktor weiter). Empfängt der `replica`-Aktor ein `subscribe_atom` (Zeile 1), wird zuerst ein *Aktor-Handle* auf den Sender der Nachricht erstellt (Zeile 2). Ist das erstellte *Aktor-Handle* gültig, wird es danach in die Menge der Abonnenten eingefügt (Zeile 5). Abschließend prüft der `replica`-Aktor, ob bereits Änderungen beziehungsweise Operationen auf den Replik-Daten ausgeführt worden sind (Zeile 7). Ist dies der Fall, so wird der aktuelle Zustand der Replik-Daten an den neuen Abonnenten versendet (Zeile 8).
Außerdem ist das Entfernen von Abonnenten zu sehen. Möchte ein Benutzer-Aktor keine weiteren Informationen über Änderungen mehr erhalten, so sendet er ein `unsubscribe_atom` an den passenden `replica`-Aktor. Empfängt der `replica`-Aktor ein `unsubscribe_atom` (Zeile 10), so wird zuerst ein *Aktor-Handle* generiert (Zeile 11). Wenn das *Aktor-Handle* gültig ist, wird der entsprechende Aktor aus der Menge der Abonnenten entfernt (Zeile 13).

5.2.3 Empfangen von Updates

Um ein empfangenes Update an alle lokal interessierten Benutzer-Aktoren zu propagieren, sind drei Schritte notwendig. Zuerst muss der δ-CRDT-Zustand aus der empfangenen Nachricht

extrahiert werden. Danach muss der extrahierte δ-CRDT-Zustand mit dem im `replica`-Aktor befindlichen Zustand zusammengeführt werden. Das Update muss nun noch an alle lokal interessierten Abonnenten verteilt werden.

Auflistung 5.4: Entpacken eines δ-CRDT-Zustands aus einer Nachricht

```
1 auto unpack = [&](message& msg) -> T {
2   T unpacked;
3   msg.apply([&](T& t) { unpacked = std::move(t); });
4   return unpacked;
5 };
```

Auflistung 5.4 zeigt, wie ein δ-CRDT-Zustand aus einer Nachricht entpackt wird. Zu sehen ist ein C++11-Lambda, welches ein Argument vom Typ `message&` entgegen nimmt (Zeile 1). Der Rückgabewert ist ein Objekt vom Typ `T`, wobei `T` das Template-Argument der `replica` -Klasse ist und somit ein CRDT. Um den Zustand aus der Nachricht zu entpacken, wird auf die Nachricht eine weiteres C++11-Lambda angewendet (Zeile 3). Das Lambda aus Zeile drei erwartet ein Argument vom Typ `T&` und hat keinen Rückgabewert. Das Lambda hat dabei Zugriff per Referenz auf die Variable `unpacked` aus dem äußeren Scope (Zeile 2). Bei der Ausführung der Lambda-Funktion wird der erwartete Typ in die äußere Variable `unpacked` bewegt. Das an `message::apply` übergebene Lambda wird nur ausgeführt, wenn die Nachricht die Typen enthält, wie diese in den Argumenten des Lambda spezifiziert sind. Sollte die Nachricht `msg` nicht den erwarteten Typen beinhalten, bleibt `unpacked` *default-initialisiert* (leerer CRDT-Zustand). Der Rückgabewert ist der entpackte Zustand (Zeile 4).

Auflistung 5.5: Empfang eines Updates als Nachricht

```
1 [&](publish_atom, message& msg) {
2   auto delta = cvrdt_.merge(unpack(msg));
3   if (delta.empty())
4     return; // Zustand ist bereits enthalten
5   buffer_.merge(delta);
6 }
```

Auflistung 5.5 zeigt den Nachrichten-Handler, welcher dafür zuständig ist, einzelne Updates zu verarbeiten. Einzelne Updates kommen ausschließlich von lokalen Benutzer-Aktoren, da Updates, welche über das Netzwerk gesendet werden, als zusammengefasster Container ver-

sendet werden. Zu sehen ist, wie der Nachrichten-Handler zwei Argumente (`publish_atom`, sowie `message&`) erwartet (Zeile 1). Die eingegangene Nachricht wird zuerst mit dem zuvor vorgestellten `unpack` entpackt. Der entpackte Zustand wird mit dem lokalen Zustand `cvrdt_` zusammengeführt. Der hierbei entstehende δ-CRDT-Zustand wird in der Variable `delta` gespeichert (Zeile 2). Anschließend wird geprüft, ob der entstandene δ-CRDT-Zustand leer ist (Zeile 3). Ist dies der Fall, bedeutet das, dass der Zustand bereits enthalten war. Eine Benachrichtigung lokaler Benutzer-Aktoren ist damit nicht notwendig, weshalb der Nachrichten-Handler verlassen wird (Zeile 4). Ist der δ-CRDT-Zustand in `delta` nicht leer, wird er mit dem Buffer für Benachrichtigungen (`buffer_`) zusammengeführt (Zeile 5).

Auflistung 5.6: Empfang von Updates als Nachricht

```
1 [&](publish_atom, std::vector<message>& msgs) {
2   T delta;
3   for (auto& msg : msgs)
4     delta.merge(unpack(msg));
5   delta = cvrdt_.merge(delta);
6   if (delta.empty())
7     return; // Zustände sind bereits enthalten
8   buffer_.merge(delta);
9 }
```

Updates, welche zwischen CAF Knoten durch das Netzwerk versendet werden, sind als Container zusammengefasst. Es ist effizienter, einen `vector<message>` zu versenden als jedes Element einzeln. Beim Empfänger wird dadurch außerdem pro `vector<message>` nur ein Nachrichten-Handler ausgelöst, anstelle von vielen. Auflistung 5.6 zeigt den Nachrichten-Handler, welcher ausgelöst wird, wenn der Replik-Aktor Updates über das Netzwerk empfängt. Die Updates in `msgs` werden zuerst entpackt und zusammengeführt. Dazu wird die Variable `delta` deklariert (Zeile 2). Anschließend werden alle Updates in den Zustand von `delta` zusammengeführt (Zeile 3-4). Die zusammengefassten Updates werden nun in den lokalen Zustand der `replica`-Klasse zusammengeführt, der dabei neu entstehende δ-CRDT-Zustand wird in die Variable `delta` gespeichert (Zeile 5). Zu diesem Zeitpunkt enthält `delta` das δ zwischen dem lokalen Zustand und allen gerade empfangenen Updates. Ist die Variable `delta` leer, bedeutet dies, dass alle empfangenen Zustände bereits lokal enthalten sind (Zeile 6). Es ist nicht notwendig, die Updates in den Benachrichtigungs-Buffer einzufügen, weshalb der Nachrichten-Handler verlassen wird (Zeile 7). Ist `delta` jedoch nicht leer, so wird der Benachrichtigungs-Buffer `buffer_` durch Zusammenführen ergänzt.

Auflistung 5.7: Verteilung von Updates an Abonnenten

```
1  [&](notify_atom) {
2    if (!buffer_.empty()) {
3      auto msg = make_message(notify_atom::value, buffer_);
4      for (auto& sub : subs_)
5        send(sub, msg);
6      buffer_ = {}; // Buffer zurücksetzen
7    }
8    delayed_send(this, milliseconds(notify_interval_ms_),
9                 notify_atom::value);
10 }
```

Auflistung 5.7 zeigt die periodische Verteilung der Benachrichtigungs-Buffer an lokale Abonnenten. Zu sehen ist ein Nachrichten-Handler, welcher nur ein Atom `notify_atom` entgegennimmt (Zeile 1). Zuerst wird geprüft, ob der Benachrichtigungs-Buffer `buffer_` leer ist (Zeile 2). Ist dies nicht der Fall, wird dieser an alle Abonnenten versendet (Zeile 3-5). Anschließend wir der Benachrichtigungs-Buffer geleert (Zeile 5). Um diesen Nachrichten-Handler periodisch auszuführen, sendet der Aktor mit `delayed_send` eine verzögerte Nachricht an sich selbst (Zeile 8-9). Diese Nachricht wird nach der in der Variable `notify_interval_ms_` definierten Verzögerung an den Aktor zugestellt und löst den Nachrichten-Handler erneut aus.

5.2.4 Robustes Ausführen von Operationen

Das CAF-CRDT-Modul bietet die Möglichkeit, Operationen auf mehreren Knoten gleichzeitig auszuführen. Damit wird die Chance, dass eine Operation verloren geht, reduziert. Das Verhalten ist bereits in Abschnitt 4.6 diskutiert. Der folgende Abschnitt erklärt, wie die Umsetzung von robusten Operationen gestaltet ist.

Erhält der *replicator_actor* einzelne Operationen mit einer Schreib- oder Lese-Regel, erzeugt dieser einen neuen Aktor, welcher die Benachrichtigung anderer entfernter Knoten übernimmt. Ist die Operation beendet, wird auch der neu erzeugte Aktor beendet.

Auflistung 5.8: Operation auf einem Replikat ausführen, ohne Zustand im Aktor zu halten

```
// ...
auto req = self->request(repl, seconds(2), read_all::value,
                         "gset<int>://visitors",
                         make_message(set));
req.receive(
  [&](read_succeed_atom, const gset<int>& result) {
    // Operation erfolgreich
  },
  [&](error&) { /* Fehlerbehandlung */ }
);
// ...
```

Auflistung 5.8 zeigt das Ausführen einer Lese-Operation. Die Funktion liest von allen bekannten Knoten im verteilten System. Zu sehen ist, wie ein Aktor `self` einen `request` an den lokalen Replikator `repl` versendet (Zeile 2-4). Der Aufruf `request` gibt einen so genannten *Request-Handle* zurück. Mit dem Aufruf `receive` auf den *Request-Handle* in Variable `req` werden die Nachrichten-Handler für die erwartete Antwort spezifiziert (Zeile 5). Es werden zwei Nachrichten-Handler gesetzt, wobei der erste den Fall des korrekten Ablaufs abdeckt (Zeile 6-8). Er erwartet ein `read_succeed_atom` und das Ergebnis der Operation ist `gset<int>`. Im Fehlerfall wird wird ein `error` erwartet (Zeile 9).

Auflistung 5.9: Zustand von Write-Read-Aktoren

```
template <class T>
struct wr_state {
  size_t messages_left; // Ausstehende Antworten
  T crdt;               // Temporärer Replikat-Zustand
  response_promise rp;  // Zum beantworten des eigentlichen Requests
};
```

Zum Ausführen der Operation wird ein neuer Aktor (der so genannte Write-Read-Aktor) erzeugt. Seine Aufgabe besteht darin, auf Antworten von entfernten Knoten zu warten. Liegt ein Ergebnis vor, beantwortet er den eigentlichen *Request*. Auflistung 5.9 zeigt den Zustand von Write-Read-Aktoren. Der Zustand ist mit der Klasse `wr_state` gekapselt und benötigt einen Template-Parameter `T`, welcher eine CRDT-Klasse sein muss (Zeile 1). Die Klasse enthält

außerdem einen Zähler `messages_left`, welcher zählt, wie viele Antworten von entfernten Knoten noch ausstehen (Zeile 3). Ein temporärer CRDT-Zustand (`crdt`) ist außerdem enthalten. Dieser findet allerdings nur Verwendung, wenn eine lesende Operation ausgeführt wird (Zeile 4). Zum Beantworten des eigentlichen *Requests* wird ein `response_promise` benötigt, welches die Variable `rp` hält.

Auflistung 5.10: Aktor für schreibende Operationen

```
1  auto send_write = [=](size_t k, const uri& id, const message& msg,
2                        const std::set<replicator_actor>& to) {
3    size_t sent = 0;
4    for (auto& rep : to) { // Gewünschter Anzahl an Knoten
5                           // die Operation senden
6      self->send(rep, write_local_atom::value, id, msg);
7      if (++sent == k) break;
8    }
9  };
10 // ...
11 [=](write_all_atom, const uri& id, std::set<replicator_actor> to,
12     const message& msg) {
13   send_write(to.size(), id, msg, to); // Operation zu allen Knoten senden
14 },
15 [=](write_k_atom, const uri& id, std::set<replicator_actor> to,
16     const message& msg, size_t k) {
17   auto n = std::min(to.size(), k);
18   send_write(n, id, msg, to);
19 },
20 [=](write_majority_atom, const uri& id, std::set<replicator_actor> to,
21     const message& msg) {
22   auto n = to.size() / 2 + 1;
23   send_write(n, id, msg,to);
24 },
25 [=](write_succeed_atom) {
26   if (--self->state.messages_left == 0) { // Antwort von allen Knoten
27                                           // empfangen
28     self->state.rp.deliver(write_succeed_atom::value);
29     self->quit();
30   }
31 }
```

Auflistung 5.10 zeigt Nachrichten-Handler eines Write-Read-Aktors, wobei hier nur die Nachrichten-Handler zu sehen sind, welche für schreibenden Operationen verwendet werden. Der Zustand des Aktors `wr_state` ist bereits initialisiert. Zum Versenden von Operationen wird die Funktion `send_write` verwendet (Zeile 1). Sie nimmt die Anzahl an Knoten, auf die geschrieben werden soll (`k`), sowie einen Replikat-Bezeichner (`id`) entgegen. Außerdem benötigt die Funktion die zu schreibende Änderung, welche sich in `msg` befindet. Des Weiteren wird ein `std::set<replicator_actor>` eingefordert, welches alle bekannten *replicator_actors* enthält.
Es sind außerdem verschieden Nachrichten-Handler zu sehen, welche für die verschieden starken Schreib-Operationen verwendet werden. Sie rufen die Funktion `send_write` mit der passenden Anzahl an Knoten auf (Zeile 13, 18, 23).
Wurde eine Operation auf einem entfernen Knoten erfolgreich ausgeführt, schickt der entfernte Knoten eine Nachricht mit einem `write_succeed_atom` an den Write-Read-Aktor zurück. Der Nachrichten-Handler zum Empfang der Antwort ist ab Zeile 25 zu sehen. Dieser zählt die ausstehenden Antworten herunter (Zeile 26). Sind alle erwarteten Antworten eingetroffen, wird der `response_promise` erfüllt und ein Resultat an den anstoßenden Benutzer-Aktor zugestellt (Zeile 28). Der Write-Read-Aktor hat seine Aufgabe erfüllt und beendet sich (Zeile 29).

Auflistung 5.11: Aktor für lesende Operationen

```
1 auto send_read = [=](size_t k, const uri& id,
2                      const std::set<replicator_actor>& from) {
3   size_t sent = 0;
4   for (auto& rep : from) {
5     self->send(rep, read_local_atom::value, id);
6     if (++sent == k) break;
7   }
8 };
9 // ...
10 [=](read_all_atom, const uri& id, std::set<replicator_actor> from) {
11   send_read(from.size(), id, from);
12 },
13 [=](read_majority_atom, const uri& id, std::set<replicator_actor> from) {
14   auto n = from.size() / 2 + 1;
15   send_read(n, id, from);
16 },
17 [=](read_k_atom, const uri& id, std::set<replicator_actor> from,
18     size_t k) {
19   auto n = std::min(from.size(), k);
20   send_read(n, id, from);
21 },
22 [=](read_succeed_atom, const T& msg) {
23   self->state.crdt.merge(msg);
24   if (--self->state.messages_left == 0) {
25     self->state.rp.deliver(read_succeed_atom::value, self->state.crdt);
26     self->quit();
27   }
28 }
```

Abbildung 5.11 zeigt den Aktor, welcher für lesende Operationen zuständig ist. Es ist die Funktion `send_read` zu sehen (Zeile 1), welche die gewünschte Anzahl an Knoten auffordert, ihr lokales Replikat zu lesen und als Antwort zu versenden. Es gibt erneut verschiedene Nachrichten-Handler für die verschiedenen Stärken der Operationen (Zeile 10, 13, 17), welche die `send_read` Funktion mit der passenden Anzahl an Knoten aufrufen.
Antwortet ein entfernter Knoten mit seinem Replikat-Zustand, wird dieser mit dem Nachrichten-Handler aus Zeile 22 entgegengenommen. Zuerst wird der empfangene Zustand mit dem temporären Replikatszustand des Aktor zusammengeführt (Zeile 23). Sind alle ausstehenden

Antworten empfangen worden, wird der `response_promise` beantwortet (Zeile 25), womit der Benutzer-Aktor, welcher den Ablauf ausgelöst hat, das Resultat erhält.

Auflistung 5.12: Timeout-Handler von Write-Read-Aktoren

```
1 [=](timeout_atom) {
2   self->state.rp.deliver(make_error(sec::request_timeout));
3   self->quit();
4 }
```

Abbildung 5.12 zeigt den Timeout-Handler von Write-Read-Aktoren. Wurden nicht alle ausstehenden Antworten auf entfernte Operationen empfangen, so wird der Nachrichten-Handler aus Zeile 1 ausgelöst. Dabei wird das `reponse_promise` mit einem Fehler beantwortet (Zeile 2), dadurch erhält der auslösende Benutzer-Aktor eine Fehlerbenachrichtigung. Anschließend beendet sich der Write-Read-Aktor (Zeile 3).

5.3 Der Replicator

Die Verwaltungsinstanz des CRDT-Moduls in jedem CAF-Knoten, der das CRDT-Modul geladen hat, ist der *Replicator*. Er beinhaltet den Initialisierungs- und Shutdown-Code des CRDT-Moduls. Außerdem verarbeitet er Nachrichten, welche von I/O-Hooks generiert werden. Diese I/O-Hooks sind notwendig, um zum Beispiel neue CAF-Knoten im verteilten System zu entdecken oder den Verlust von Verbindung zu bemerken. In der aktuellen Implementierung findet keine Erkennung der Topologie des Netzwerks statt.
Der *Replicator* ist in eine klassische objektorientierte Klasse (`replicator`) und einen Aktor (`replicator_actor`) aufgeteilt. Die `replicator`-Klasse stellt den Initialisierungs- und Shutdown-Code bereit. Die `replicator_actor`-Klasse übernimmt alle Aufgaben, die aus einem Aktor-Kontext gestellt werden. Sie verarbeitet Nachrichten und verwaltet den Zustand, welcher benötigt wird, um die Replikation zwischen Knoten umzusetzen. Ein Beispiel hierfür ist die Verteilungslogik.

5.3.1 Instanziierung und Stopp des CAF-CRDT-Moduls

Auflistung 5.13: Initialisierung und Start des CAF-CRDT-Moduls

```
1  void replicator::init(actor_system_config& cfg) {
2    cfg.add_hook_type<detail::replicator_hook>().
3        add_message_type<uri>("uri").
4        add_message_type<std::unordered_set<uri>>("unordered_set<uri>").
5        add_message_type<std::vector<message>>("vector<message>");
6  }
7
8  void replicator::start() {
9    manager_ = make_replicator_actor(system_);
10   system_.registry().put(replicator_atom::value,
11                          actor_cast<strong_actor_ptr>(manager_));
12 }
```

Auflistung 5.13 zeigt die Initialisierung und das Starten des CAF-CRDT-Moduls. Dabei wird zuerst die Funktion `replicator::init` aufgerufen (Zeile 1). Die Funktion registriert die für den Betrieb notwendigen I/O-Hooks (Zeile 2; die Callbacks für die I/O-Hooks sind in Auflistung 5.14 beschrieben). Des Weiteren werden Typen, welche serialisiert und deserialisiert werden müssen, beim Typ-System angemeldet. Der erste Typ, der dabei angemeldet wird, ist die `uri`-Klasse (Zeile 3), welche für Replikat-Bezeichner verwendet wird. Außerdem werden die Typen `unordered_set<uri>` und `std::vector<message>` angemeldet, welche notwendig sind, um Replikat-Bezeichner zu synchronisieren und Updates zu übertragen.
Nach der Initialisierung aller Module werden die Module gestartet. Zum Starten des CRDT-Moduls wird die Funktion `replicator::start()` ausgeführt (Zeile 8). Dabei wird die Instanz des *replicator_actor* erzeugt (Zeile 9), danach wird das Handle zu dem Aktor in der lokalen Registry angemeldet (Zeile 10-11). Die Registry ist ein Key/Value-Store in jedem CAF Knoten und erlaubt Zugriff auf registrierte *Aktor-Handles* per Schlüssel (Atom). Zur Laufzeitoptimierung verwendet CAF Atoms statt Strings, da Zugriff auf die Registry auch von entfernten Knoten erfolgen kann. Im CRDT-Modul wird dies benötigt, damit für einen neu entdeckten Knoten ein *Aktor-Handle* zu dem *replicator_actor* des neu entdeckten Knotens abgerufen werden kann.

Auflistung 5.14: Callbacks für die Hooks des I/O-Moduls

```
1  void replicator_hook::new_connection_established_cb(const node_id& node)
       {
2    on_new_connection(node);
3  }
4
5  void replicator_hook::new_route_added_cb(const node_id&,
6                                           const node_id& node) {
7    on_new_connection(node);
8  }
9
10 void replicator_hook::connection_lost_cb(const node_id& node) {
11   auto hdl = sys_.replicator().actor_handle();
12   self_->send(hdl, connection_lost_atom::value, node);
13 }
14
15 void replicator_hook::on_new_connection(const node_id& node) {
16   auto hdl = sys_.replicator().actor_handle();
17   self_->send(hdl, new_connection_atom::value, node);
18 }
```

Auflistung 5.14 zeigt die zuvor angemeldeten Callbacks für die Hooks des I/O-Moduls. Zu sehen sind drei Callbacks, wobei zwei davon beim Entdecken von neuen Knoten ausgeführt werden. Die Callbacks generieren System-Nachrichten, welche innerhalb des CRDT-Moduls verwendet werden und an den *replicator_actor* gesendet werden. Der erste Callback ist die Funktion `new_connection_established_cb`. Dieser wird immer aufgerufen, wenn das I/O-Modul eine neue TCP-Verbindung zu einem CAF-Knoten aufgebaut hat. Der Callback ruft lediglich die Funktion `replicator_hook::on_new_connection` auf (Zeile 2), welche eine Nachricht mit einem `new_connection_atom` und dem Bezeichner des neuen Knotens zum *replicator_actor* sendet (Zeile 16-17).
Der zweite Callback `new_route_added_cb` (Zeile 5-6) wird aufgerufen, wenn ein neuer Knoten vom I/O-Modul entdeckt wurde, aber zu diesem keine direkte Verbindung per TCP besteht. Das erste Argument, welches hier nicht verwendet wird, ist der Knoten-Bezeichner, über den Nachrichten an den neu entdeckten Knoten geroutet werden. Das zweite Argument (`node`) ist der Bezeichner des neu entdeckten Knotens.
Der dritte Callback `replicator_hook::connection_lost_cb` (Zeile 15) wird aufgerufen, wenn das I/O-Modul feststellt, dass eine Verbindung zu einem Knoten verloren wurde. Dabei wird

eine Nachricht an den *replicator_actor* gesendet, welche ein `connection_lost_atom` und den Bezeichner des verlorenen Knotens enthält (Zeile 11-12).

Auflistung 5.15: Herunterfahren des CAF-CRDT-Moduls

```
1 void replicator::stop() {
2   scoped_actor self{system(), true};
3   self->monitor(manager_);
4   self->send_exit(manager_, exit_reason::user_shutdown);
5   self->wait_for(manager_);
6   destroy(manager_);
7 }
```

Auflistung 5.15 zeigt das Herunterfahren des CRDT-Moduls. Die Funktion `replicator::stop` wird von CAF aufgerufen, wenn alle sichtbaren Benutzer-Aktoren beendet wurden. Zu sehen ist, wie der *replicator_actor* beendet wird, welcher sich in Variable `manager_` befindet. Da die Klasse `replicator` selbst ein Handle zum *replicator_actor* hält, muss die Referenz dieses Handles mit Hilfe von `destroy` freigegeben werden. Jedoch muss, bevor dies geschieht, sichergestellt sein, dass der Aktor sich beendet hat. Um dies zu erreichen, wird ein `scoped_actor` (Zeile 2) gespawnt, welcher so lange blockiert, bis der *replicator_actor* beendet ist (Zeile 5).

Verhalten bei unerwartetem Beenden der CAF-Anwendung Wird die CAF-Anwendung unerwartet beendet, werden die *shutdown* Funktionen der CAF-Module nicht ausgeführt. Lokale Operationen, die noch nicht propagiert wurden, sind somit verloren. CAF besitzt aktuell keine Funktionalität um den Zustand von Aktoren nach einem Crash wiederherzustellen (Persistenz). Andere CAF-Instanzen erkennen jedoch, dass der Knoten nicht mehr erreichbar ist (Verlieren der TCP-Verbindung) und können auf den Verlust der Verbindung reagieren.

5.3.2 Typisiertes Interface des *replicator_actor*

Der *replicator_actor* verfügt über ein typisiertes Nachrichten-Interface. Damit werden Nachrichten, die an den *replicator_actor* gesendet werden, zur Kompilierzeit geprüft. Wird dem *replicator_actor* eine Nachricht gesendet, welche nicht unterstützt wird, so gibt der Compiler einen Fehler aus. Das Interface des *replicator_actors* lässt sich in zwei Teile aufteilen: (1) ein öffentliches Interface für den Anwendungsentwickler und (2) ein privates Interface für interne Nachrichten.

Auflistung 5.16: Öffentlich zugängliches Interface des *replicator_actor*

```
using replicator_actor =
  typed_actor<
    // Der Sender dieser Nachricht meldet sich an Replikat an
    reacts_to<subscribe_atom, uri>,
    // Der Sender meldet sich von einem Replikat ab
    reacts_to<unsubscribe_atom, uri>,
    // Liest ein Replikat von allen Knoten
    reacts_to<read_all_atom, uri>,
    // Liest ein Replikat von K Knoten
    reacts_to<read_k_atom, size_t, uri>
    // Liest ein Replikat von der Mehrheit der Knoten
    reacts_to<read_majority_atom, uri>,
    // Liest ein Replikat (nur lokal)
    reacts_to<read_local_atom, uri>,
    // Schreibt ein delta auf alle Knoten
    reacts_to<write_all_atom, uri, message>,
    // Schreibt ein delta auf K Knoten
    reacts_to<write_k_atom, size_t, uri, message>,
    // Schreibt ein delta auf die Mehrheit der Knoten
    reacts_to<write_majority_atom, uri, message>,
    // Schreibt ein delta auf den lokalen Knoten
    reacts_to<write_local_atom, uri, message>,
    // Löscht ein Replikat lokal
    reacts_to<del_id_atom, uri>
>;
```

Auflistung 5.16 zeigt einen Ausschnitt des *replicator_actor*-Interfaces. Zu sehen sind die öffentlichen Funktionen, die vom Anwendungsentwickler beziehungsweise Benutzer-Aktoren verwendet werden können. Die folgenden Unterpunkte listen die einzelnen unterstützten Funktionen auf:

- `reacts_to<subscribe_atom, uri>` (Zeile 4) erlaubt es, dass sich Benutzer-Aktoren an Replikaten anmelden können. Das Replikat, an welchem der Benutzer-Aktor sich anmeldet, ist als `uri` übergeben. Bei Änderungen am Zustand des Replikats wird der angemeldete Aktor (Abonnent) benachrichtigt. Der Abonnent erhält initial den aktuellen Zustand des lokalen Knotens.

- `reacts_to<unsubscribe_atom, uri >` (Zeile 6) meldet den Sender der Nachricht von zukünftigen Benachrichtigungen eines Replikats ab, das Replikat ist erneut als `uri` spezifiziert.
- `reacts_to<read_all_atom, uri>` (Zeile 8) liest ein Replikat von allen Knoten. Dabei wird der Zustand von allen Knoten kollabiert beziehungsweise durch die *merge*-Funktion der CRDT zusammengefasst. Das zusammengefasste Ergebnis wird asynchron an den sendenden Benutzer-Aktor gesendet.
- `reacts_to<read_k_atom, size_t, uri>` (Zeile 10) dient dazu, ein Replikat von k Knoten zu lesen. Dazu übergibt der Benutzer-Knoten zusätzlich eine Anzahl an Knoten K als `size_t`. Wie gewohnt wird für das Beziehen auf ein Replikat ein `uri` verwendet.
- `reacts_to<read_majority_atom, uri>` (Zeile 12) liest von der Mehrheit an Knoten. Es wird wieder ein Replikat-Bezeichner als `uri` übergeben. Die Mehrheit an Knoten berechnet sich aus $N/2 + 1$, wobei N die totale Anzahl an Knoten ist.
- `reacts_to<read_local_atom, uri>` (Zeile 14) liest nur die lokale Instanz des Replikats. Erneut ist ein Replikat-Bezeichner anzugeben.
- `reacts_to<write_all_atom, uri, message>` (Zeile 16) schreibt einen δ-Zustand auf allen bekannten Knoten. Ein Replikat-Bezeichner muss übergeben werden sowie der δ-Zustand, welcher geschrieben werden soll. Der δ-Zustand muss in eine Nachricht gepackt sein, da der Replikator an dieser Stelle den Typen des CRDT nicht verarbeiten kann.
- `reacts_to<write_k_atom, size_t, uri, message>` (Zeile 18) erlaubt es, einen δ-Zustand an K Knoten im System zu schreiben. Ein K als `size_t` muss übergeben werden sowie der Replikat-Bezeichner und der δ-Zustand als Nachricht verpackt.
- `reacts_to<write_majority_atom, uri, message>` (Zeile 20) schreibt einen δ-Zustand an die Mehrheit der Knoten. Es muss erneut ein Replikat-Bezeichner angegeben werden sowie der zu schreibende δ-Zustand als Nachricht verpackt.
- `reacts_to<write_local_atom, uri, message>` (Zeile 22) schreibt einen δ-Zustand auf das lokale Replikat. Der δ-Zustand wird erst beim nächsten *flush* des δ-CRDT-Buffers propagiert.
- `reacts_to<del_id_atom, uri>` (Zeile 24) löscht ein Replikat lokal. Dabei wird nur der Replikat-Bezeichner benötigt.

Auflistung 5.17: Privates Interface des *replicator_actor*

```
1  // ...
2  // Einzelnes delta für ein Replikat
3  reacts_to<uri, message>,
4  // Mehrere deltas für ein Replikat
5  reacts_to<uri, std::vector<message>>,
6  // Löst aus, dass Replikate ihren gesamten Zustand in den Sendebuffer
       schreiben
7  reacts_to<tick_state_atom>,
8  // Vollständiger Zustand, wird in Sendebuffer gespeichert
9  reacts_to<copy_ack_atom, uri, message>,
10 // Löst das Synchronisieren von Replikat-Bezeichnern aus
11 reacts_to<tick_ids_atom>,
12 // Flusht den Sendebuffer
13 reacts_to<tick_buffer_atom>,
14 // Es wurde ein neuer CAF-Knoten entdeckt
15 reacts_to<new_connection_atom, node_id>,
16 // Ein bekannter CAF-Knoten ist nicht länger erreichbar
17 reacts_to<connection_lost_atom, node_id>,
18 // Ein anderer Knoten fragt nach den Replik-Bezeichnern
19 reacts_to<get_ids_atom, size_t>,
20 // Antwort auf die Frage nach Replik-Bezeichnern
21 reacts_to<size_t, std::unordered_set<uri>>,
```

Das Private-Interface wird zwischen *replicator_actor*-Instanzen und *replica*-Aktor-Instanzen verwendet. Auflistung 5.17 zeigt die privaten Funktionen, die intern im CRDT-Modul verwendet werden. Die folgenden Funktionen werden mit dem Interface abgedeckt:

- `reacts_to<uri, message>` (Zeile 3) wird ausgelöst, sobald eine lokale Änderung an einem Replikat vorgenommen wurde. Es wird ein Replikat-Bezeichner als `uri` versendet sowie der δ-Zustand, welcher die Änderung beschreibt.
- `reacts_to<uri, std::vector<message>>` (Zeile 5) wird ausgelöst, wenn Replikat-Daten von einem anderen Knoten empfangen werden. Es wird erneut ein Replikat-Bezeichner als `uri` übergeben sowie ein `vector<message>`, welcher mehrere δ-Zustände beinhalten kann.
- `reacts_to<tick_state_atom>` (Zeile 7) löst aus, dass `replica`-Aktoren ihren Zustand in den Sendebuffer des *replicator_actors* schreiben. Dies geschieht mit einer separaten Antwort-Nachricht.

- `reacts_to<copy_ack_atom, uri, message>` (Zeile 9) ist die Antwort auf die Nachricht, die in dem Handler von Zeile sieben ausgelöst wurde. Die Antwort beinhaltet einen Replikat-Bezeichner als *uri* sowie den vollständigen Zustand des lokalen Replikats verpackt in eine Nachricht.
- `reacts_to<tick_ids_atom>` (Zeile 11) löst aus, dass Replikat-Bezeichner von anderen Knoten angefordert werden.
- `reacts_to<tick_buffer_atom>` (Zeile 13) flusht den Sendebuffer, welcher δ-Zustände enthält, die noch nicht propagiert sind.
- `reacts_to<new_connection_atom, node_id>` (Zeile 15) wird von den Callbacks für die I/O-Hooks versendet. Der *replicator_actor* prüft nun, ob der neu entdeckte Knoten einen *replicator_actor* instanziiert hat. Ist dies der Fall, wird der Knoten in die Liste der Knoten aufgenommen, welche Daten replizieren können.
- `reacts_to<connection_lost_atom, node_id>` (Zeile 17) wird ebenfalls von den Callbacks für die I/O-Hooks versendet. Der *replicator_actor* entfernt den Knoten aus der Liste von Knoten, welche Daten replizieren.
- `reacts_to<get_ids_atom, size_t>` (Zeile 19) wird ausgeführt, wenn ein anderer Knoten den empfangenden Knoten nach seinen instanziierten Replikat-Bezeichnern fragt. Dabei übergibt er die letzte bekannte Versionsnummer des empfangenen Knoten mit.
- `reacts_to<size_t, std::unordered_set<uri>>` (Zeile 21) reagiert auf Ergebnis der Anfrage nach Replikat-Bezeichnern. Dabei wird die aktuelle Versions-Nummer des Knotens als `size_t` sowie die von dem Knoten instanziierten Replikat-Bezeichner als `unordered_set` übertragen.

5.3.3 Erstellen von *replica*-Aktor Instanzen

Neue Replikate werden dann lokal instanziiert, wenn ein Benutzer-Aktor Interesse an einem Replikat-Bezeichner hat, welcher lokal noch nicht verwendet wurde. Es muss zuerst ein neuer Aktor vom Typ *replica* gespawnt werden, welcher zum Typ des Replikat-Bezeichners passt.

Auflistung 5.18: Instanziieren neuer Replikate

```
1  // ...
2  std::unordered_map<uri, actor> states_; // Replikat-ID -> replica<T>
3  // ...
4  actor find_actor(const uri& u) {
5    auto iter = states_.find(u);
6    if (iter == states_.end()) {
7      auto args = make_message(u, notify_interval_ms_);
8      auto opt = system().spawn<actor>(u.scheme(), std::move(args));
9      iter = states_.emplace(u, *opt).first;
10     dist_.add_id(u);
11   }
12   return iter->second;
13 }
14 // ...
15 [&](subscribe_atom atm, const uri& u) {
16   delegate(find_actor(u), subscribe_atom::value);
17 }
18 // ...
```

Auflistung 5.18 zeigt, was notwendig dafür ist, um einen Aktor für ein neues Replikat zu erzeugen. Der `replicator_actor` besitzt eine `unordered_map<uri, actor> states_` (Zeile 2), dessen Aufgabe es ist, von Replikat-Bezeichnern nach Aktoren abzubilden.
Außerdem ist eine Funktion `find_actor` zu sehen, welche in der Map nach einem Aktor zu dem passenden Replikat-Bezeichner sucht (Zeile 4). Ist kein Aktor für diesen Replikat-Bezeichner vorhanden, so wird ein neuer Aktor für den Typ des Replikats erzeugt (Zeile 5-11). Das besondere hierbei ist, dass der Aktor erzeugt wird, ohne dass ein Typ angegeben ist. Der Aktor wird anhand der Typ-Information erzeugt, welche im *scheme* des `uri` (Replikat-Bezeichners) kodiert ist (Zeile 8).
Der neu erzeugte Aktor wird zur Map unter dem passenden Replikat-Bezeichner hinzugefügt (Zeile 9). Des Weiteren wird dem Verteilungs-Layer (welches separat in Abschnitt 5.4 beschrieben ist) mitgeteilt, dass dieser Knoten nun ein Replikat für den passenden Replikat-Bezeichner instanziiert.
Der gerade beschriebene Ablauf wird durch den Nachrichten-Handler (Zeile 15-17) angestoßen. Ist der neue Aktor erfolgreich erzeugt, delegiert er die Aufgabe der Anmeldung an den neuen Aktor.
Das Delegieren von Nachrichten (Zeile 16) erlaubt es, Aufgaben weiterzuleiten und die resul-

tierende Antwort trotzdem an den auslösenden Aktor zu senden. Beim Delegieren wird der Absender nicht geändert. Es wird eine neue Nachricht in die Mailbox des Ziels gelegt, dabei ist der Absender jedoch der der alten Nachricht.
Daten, die pro Knoten gespeichert werden, sind in dem struct `node_data` zu sehen (Zeile 2-6). Dabei handelt es sich um einen Versionszähler vom Typ `size_t` (Zeile 3) für die Liste von instanziierten Replikaten in dem Knoten (Zeile 5). Außerdem wird ein *Aktor-Handle* auf den *replicator*-Aktor des jeweiligen Knotens gehalten (Zeile 4).

5.4 Distribution Layer

Das *Distribution-Layer* ist Teil des *replicator*-Actors. Der *replicator*-Actor hält das *Distribution-Layer* in seinem Zustand und entkoppelt so Verteilungs-Logik und den eigentlichen Aktor.

Auflistung 5.19: Zustand der *distribution_layer*-Klasse

```
1  class distribution_layer {
2    struct node_data {
3      size_t version;
4      replicator_actor replicator;
5      std::unordered_set<uri> filter;
6    };
7    using map_type = std::unordered_map<node_id, node_data>;
8    using buffer_type = std::unordered_map<uri, std::vector<message>>;
9    using uri_to_nodeid_type = std::unordered_map<uri, std::set<node_id>>;
10   // ...
11 private:
12   node_data local_; // Informationen des lokalen Knotens
13   map_type store_;  // Informationen von entfernten Knoten (node_id =>
         node_data)
14   uri_to_nodeid_type uri_to_nodes_; // Bildet Replikat-Bezeichner auf
         interessierte Knoten ab
15   buffer_type buffer_; // Buffer für delta-CRDTs (Replikat-Bezeichner =>
         Nachrichten)
16 };
```

Auflistung 5.19 zeigt den Zustand des *distribution_layer*. Es sind drei Maps vom Typ `std::unordered_map` zu sehen, welche einen effizienten Zugriff von $\mathcal{O}(1)$ (*best* & *avarage* case) bieten.
Die erste Map-Typ (Zeile 7) bildet von Knoten-Bezeichnern `node_id` auf `node_data` ab. Jeder

bekannte CAF-Knoten, der das CRDT-Modul geladen hat, besitzt einen Eintrag in dieser Map. Die zweite Map-Typ (Zeile 8) bildet von Replikat-Bezeichnern auf einen `vector<message>` ab. Der `vector<message>` beinhaltet δ-CRDT-Zustände.
Der dritte Map-Typ (Zeile 9) bildet von Replikat-Bezeichnern auf interessierte Knoten `set<node_id>` ab.

5.4.1 Handshake mit neu entdeckten Knoten

Auflistung 5.20: Aktion nach Entdecken eines neuen Knotens

```
1  void add_new_node(const node_id& nid) {
2    auto& mm = impl_->home_system().middleman();
3    auto query = mm.remote_lookup(replicator_atom::value, nid);
4    if (query) {
5      auto repl = actor_cast<replicator_actor>(query);
6      node_data data{0, repl, {}};
7      store_.emplace(nid, std::move(data));
8      send_as(impl_, repl, get_ids_atom::value, size_t{0});
9    }
10 }
```

Abbildung 5.20 zeigt die Funktion `distribution_layer::add_new_node`. Diese Funktion wird ausgeführt, sobald ein neuer CAF-Knoten entdeckt wurde. Der *replicator*-Aktor führt diese Funktion aus, sobald er von dem entsprechenden *I/O-Callback* eine Nachricht erhält. Im ersten Schritt wird geprüft, ob der neu entdeckte Knoten das CAF-CRDT-Modul geladen hat. Um dies zu prüfen wird mit der Funktion `remote_lookup` (Zeile 3) der *replicator*-Aktor des neu entdeckten Knotens abgerufen. Ist in dem neu entdeckten Knoten kein *replicator*-Aktor registriert, ist das CAF-CRDT-Modul nicht geladen. Es wird keine weitere Aktion durchgeführt (Zeile 4). Konnte mit Hilfe von `remote_lookup` ein *replicator*-Aktor angefordert werden, so wird der notwendige Zustand für den neuen Knoten als `node_data` initialisiert (Zeile 5-7). Der Beginn des initialen Handshakes, welcher die in den jeweiligen Knoten verfügbaren Replikat-Bezeichner austauscht, ist in Zeile 8 zu sehen. Dabei wird eine Nachricht mit einem `get_ids_atom` und der letzten bekannten Version (initial 0) des neuen Knotens an den neuen Knoten gesendet.

5.4.2 Periodisches Austauschen von Replikat-Bezeichnern

Knoten synchronisieren ihre Replikat-Bezeichner in einer konfigurierbaren Periode. Der Algorithmus zum Synchronisieren ist in drei Funktionen aufgeteilt, welche im Folgenden erklärt sind.

Auflistung 5.21: Replikat-Bezeichner von bekannten Knoten anfordern

```
1 void pull_ids() {
2   for (auto& entry : store_) {
3     auto& data = entry.second;
4     send_as(impl_, data.replicator, get_ids_atom::value, data.version);
5   }
6 }
```

Auflistung 5.21 zeigt die Funktion `pull_ids`. Diese Funktion fragt alle bekannten Knoten nach ihren Replikat-Bezeichnern. Dabei wird jedem bekannten Knoten die letzte von diesem Knoten bekannte Versionsnummer seiner Replikat-Bezeichner gesendet. Stellt der empfangende Knoten fest, dass die Versionsnummer nicht aktuell ist, werden die aktuellen Replikat-Bezeichner übertragen. Zu sehen ist, wie über alle Einträge in `store_` iteriert wird (Zeile 2). `store_` ist die Map, welche zu jedem bekannten Knoten die `node_data` hält. Für jeden bekannten Knoten wird nun eine Nachricht an den jeweiligen *replicator*-Aktor gesendet, welche aus einem `get_ids_atom` und der Versionsnummer besteht (Zeile 4).

Auflistung 5.22: Knoten empfängt Anfrage nach Replikat-Bezeichnern

```
1 void get_ids(const node_id& intrested_node, size_t seen) {
2   if (seen == local_.version)
3     return;
4   auto iter = store_.find(instested_node);
5   if (iter == store_.end())
6     return;
7   auto& data = iter->second;
8   send_as(impl_, data.replicator, local_.version, local_.filter);
9 }
```

Auflistung 5.22 zeigt die Funktion `get_ids`, welche ausgeführt wird, wenn ein *replicator*-Aktor eine Nachricht mit einem `get_ids_atom` und einer Versionsnummer empfängt. Als

Referenz für den anfragenden Knoten wird in die Funktion das Argument `intrested_node` vom Typ `node_id` übergeben, außerdem wird die empfangene Versionsnummer in Argument `seen` übergeben (Zeile 1).
Zuerst wird geprüft, ob die empfangene Versionsnummer `seen` der aktuellen Versionsnummer des lokalen Knotens entspricht (Zeile 2). Ist dies der Fall, ist es nicht notwendig die Replikat-Bezeichner zu übertragen, die Funktion wird sofort mit `return` verlassen (Zeile 3).
Ist die empfangene Versionsnummer nicht mehr aktuell, so wird zuerst der *replicator*-Aktor des interessierten Knotens gesucht. Ein *Aktor-Handle* auf den Aktor befindet sich in den `node_data` für den interessierten Knoten (Zeile 4). Abschließend werden die aktuellen Daten des lokalen Knotens übertragen (Zeile 8). Die `node_data` für den lokalen Knoten befinden sich in der Variable `local_`.

Auflistung 5.23: Knoten empfängt die Replikat-Bezeichner eines anderen Knotens

```
1  void update(const node_id& nid, size_t version,
2              std::unordered_set<uri>&& ids) {
3    auto& data = store_[nid];
4    if (version > data.version) {
5      // Alte IDs vom Mapping entfernen
6      for (auto& u : data.filter) uri_to_nodes_[u].erase(nid);
7      // Neue Werte setzen (Replikat-Bezeichner, Version)
8      data.filter = std::move(ids);
9      data.version = version;
10     // Neue Werte zum Mapping hinzufügen
11     for (auto& u : data.filter) uri_to_nodes_[u].emplace(nid);
12   }
13 }
```

Auflistung 5.23 zeigt den letzten Schritt der Replikat-Bezeichner Synchronisation. Zu sehen ist die Funktion `update`, sie wird ausgeführt, wenn ein Knoten eine neue Versionsnummer sowie einen Container mit Replikat-Bezeichnern eines anderen Knotens empfängt. Als Argumente werden Knoten-Bezeichner des Knotens übergeben (`nid`), welcher die Daten übertragen hat. Außerdem wird die empfangene Versionsnummer und der Container, welche die Replikat-Bezeichner enthält, übergeben (Zeile 1-2).
Die Funktion holt zuerst für den passenden Knoten die `node_data` (Zeile 3). Danach prüft die Funktion ob die empfangene Versionsnummer größer der bekannten Versionsnummer ist. Ist dies nicht der Fall, werden die empfangenen Daten im nächsten Schritt übernommen

(Zeile 4). Um die empfangenen Daten zu übernehmen ist es zuerst notwendig, die Abbildung von Replikat-Bezeichnern auf interessierte Knoten anzupassen. Es werden alle Einträge zu dem Knoten aus der Abbildung entfernt (Zeile 6). Anschließend werden die neuen Daten übernommen, dazu werden die Daten des Knotens einfach mit den neuen Daten überschrieben (Zeile 8-9). Im letzten Schritt wird die Abbildung von Replikat-Bezeichnern wieder mit den interessierten Knoten ergänzt (Zeile 11).

5.4.3 Änderung an den lokalen Replikat-Bezeichnern

Die lokalen Replikat-Bezeichner werden geändert, wenn ein *replic*-Aktor in einem Knoten neu *gespwant* wird oder entfernt wird.

Auflistung 5.24: Änderungen am lokalen (eigenen) *node_data* Datensatz

```
1 inline void modify_ids(const uri& id, bool erase) {
2   if (erase) local_.filter.erase(id);
3   else       local_.filter.emplace(id);
4   local_.version++;
5 }
```

Auflistung 5.24 zeigt die Funktion `modify_ids`, welche aufgerufen wird, wenn ein *replic*-Aktor erstellt oder terminiert wird. Dabei wird der mit `id` übergebene Replikat-Bezeichner (Zeile 1) entweder aus der Menge der lokalen Replikat-Bezeichner entfernt oder hinzugefügt (Zeile 2-3). Außerdem wird die lokale Versionsnummer mit eins inkrementiert (Zeile 4). Es findet keine Übertragung von Daten statt, andere Knoten werden erst informiert, wenn diese nach aktuellen Daten fragen.

5.4.4 Buffern und Senden von δ-CRDT-Zuständen

Auflistung 5.25: Einfügen von δ-CRDT-Zuständen in den Sendebuffer

```
1 void publish(const uri& id, const message& msg) {
2   buffer_[id].emplace_back(msg);
3 }
```

Auflistung 5.25 zeigt die *publish*-Funktion. Sie wird aufgerufen, sobald lokal ein δ-CRDT-Zustand generiert wurde und im verteilten System propagiert werden soll. Es ist zu sehen, dass

ein Replikat-Bezeichner `id` und eine Nachricht `msg` (welche den δ-CRDT-Zustand enthält) als Argumente erwartet werden (Zeile 1). Die Nachricht wird in den Sendebuffer für das Replikat geschrieben (Zeile 2). Der Sendebuffer ist eine Map, wobei von Replikat-Bezeichnern auf ein Container von Nachrichten abgebildet wird.

Auflistung 5.26: Flushen der Sendebuffer

```
1  void flush_buffer() {
2    for (auto& entry : buffer_) { // Iteriere über alle Replikat
3                                  // Sendebuffer
4      auto& id  = entry.first;    // Replikat-Bezeichner
5      auto& set = entry.second;   // Gebufferte Einträge
6      if (set.empty())
7        continue; // Kein Eintrag vorhanden, Buffer überspringen
8      auto& intrested_nodes = uri_to_nodes_[id]; // An Replikat
9                                                 // interessierte Knoten
10     for (auto& node : intrested_nodes) { // Buffer an alle interessierten
11                                          // Knoten senden
12       send_as(impl_, store_[node].replicator, id, set);
13     }
14     set.clear(); // Buffer zurücksetzen
15   }
16 }
```

Auflistung 5.26 zeigt die Funktion zum Flushen des Sendebuffers. Es wird über alle Replikat Sendebuffer iteriert (Zeile 2). Ist ein Sendebuffer leer, so wird der Eintrag übersprungen (Zeile 6-7). Für jeden Eintrag wird festgestellt, welche anderen Knoten an dem Replikat interessiert sind (Zeile 7). Der Zugriff auf die interessierten Knoten ist dabei möglichst effizient und bietet eine Komplexität von $\mathcal{O}(1)$ (*best & avarage* case). Im nächsten Schritt wird eine Nachricht an alle interessierten Knoten (beziehungsweise an die *replicator*-Aktors in den Knoten) gesendet (Zeile 10-12). Versendet wird der Replikat-Bezeichner (`id`) sowie alle gesammelten δ-CRDT-Zustände (`set`). Abschließend wird der Sendebuffer für das gerade verarbeitete Replikat geleert (Zeile 14).

5.5 Erweiterung um eigene CRDTs

Auflistung 5.27 zeigt am Beispiel des *PN-Counters*, wie Entwickler eigene CRDTs mit CAF implementieren können und was notwendig ist, um eigene CRDTs zu implementieren. Um

Auflistung 5.27: Implementierung eines Benutzer-Definierten CRDT

```
template <class T> class pn_counter : public base_datatype {
  pn_counter(actor& key, T up, T down) {
    up_.map[key] = up;
    down_.map[key] = down;
  }
public:
  pn_counter() = default;

  template <class ActorType>
  pn_counter(ActorType&& owner, std::string id)
    : base_datatype(std::forward<ActorType>(owner), std::move(id)) {}

  void inc() {
    auto key = this->owner();
    auto new_val = ++up_.map_[key];
    publish(pn_counter{key, new_val, 0});
  }

  void dec() {
    auto key = this->owner();
    auto new_val = ++down_.map_[key];
    publish(pn_counter{key, 0, new_val});
  }

  T count() const { return up_.count() - down_.count(); }

  pn_counter<T> merge(const pn_counter<T>& other) {
    pn_counter<T> delta;
    delta.up_ = up_.merge(other);
    delta.down_ = down_.merge(other);
    return delta;
  }

  inline bool empty() const { return up_.empty() && down_.empty(); }

  template <class Processor>
  friend void serialize(Processor& proc, pn_counter<T>& x) {
    proc & x.up_;
    proc & x.down_;
  }

private:
  gcounter<T> up_;
  gcounter<T> down_;
};
```

zu zeigen, dass CRDTs kombinierbar sind, implementiert die Auflistung den *PN-Counter* mit Hilfe von zwei *GCountern*, wobei einer die Additionen und ein anderer die Subtraktionen zählt. Die *GCounter* sind jeweils als Maps implementiert. Jeder Aktor, der den Zähler ändert, hat dabei einen eigenen Eintrag in der Map, welcher auch nur vom jeweiligen Aktor geändert wird. Liegen für einen Schlüssel (Aktor) verschiedene Werte vor, gewinnt der größere Wert.

Zu sehen ist die Klasse *pn_counter*, welche ein Template-Argument T entgegen nimmt und von der Klasse *base_datatype* erbt (Zeile 1). Außerdem ist ein privater Konstruktor zu sehen, welcher verwendet wird, um einen δ-Zustand zu instanziieren (Zeile 2-5).
Es wird ein *default*-Konstruktor eingefordert (Zeile 7), welcher von CAF benötigt wird, um Instanzen dieses Typs zu serialisieren und zu deserialisieren. Um eine Instanz der Klasse in einem Aktor-Zustand halten zu können, wird ein weiterer Konstruktor benötigt (Zeile 9-11). Dieser nimmt ein beliebiges *Aktor-Handle* entgegen, welches den Aktor darstellt, in dessen Zustand das spätere Replikat gehalten wird. Außerdem wird ein `std::string` übergeben, welcher den Bezeichner des Replikats darstellt. Beide Argumente werden an den Konstruktor der Basisklasse *base_datatype* übergeben.

Die Funktionen *inc* (Zeile 13-17) und *dec* (Zeile 19-23) sind spezifisch für den *PN-Counter* und inkrementieren oder dekrementieren um den Wert eins. Beide Funktionen sind ähnlich aufgebaut. Zuerst wird der Schlüssel abgerufen (`this->owner()`). Mit diesem Schlüssel wird die Map des jeweiligen Zählers hochgezählt. Danach wird ein δ-Zustand generiert, welcher an die Funktion *publish* (geerbt von *base_datatype*) übergeben wird. Dadurch wird der δ-Zustand im verteilten System asynchron propagiert.
Die Funktion *count* (Zeile 25) stellt den totalen Wert des Zählers dar. Es wird die Anzahl der Inkrementierungen von der Anzahl an Dekrementierungen subtrahiert.

Die *merge*-Funktion (Zeile 27-32) muss eine neue Instanz der Klasse zurückgeben, welche einen δ-Zustand darstellt. Ein δ-Zustand beinhaltet die symmetrische Differenz zwischen der *this*-Instanz und der übergebenen Instanz *other*. Die *merge*-Funktion ruft in diesem Fall lediglich die *merge*-Funktionen der beiden *GCounter*-Instanzen auf. Die dabei resultierenden δ-Zustände werden in einer neuen *PN-Counter*-Instanz (*delta*) gesetzt und die neue Instanz, welche den δ-Zustand darstellt, wird zurückgegeben.

Die Klasse muss zudem eine Funktion `serialize` bereitstellen (Zeile 34-38). Diese Funktion wird von CAF benötigt, um Zugriff auf die zu serialisierenden beziehungsweise deserialisie-

renden Mitglieder der Klasse zu erhalten. Die Funktion ist als *friend* deklariert, womit die Funktion nicht zur Klasse gehört, sondern eine freie Funktion ist. Die Funktion hat jedoch Zugriff auf als *privat* oder *protected* deklarierte Mitglieder der Klasse.

5.6 Diskussion

Die Design Ziele wurden nur teilweise erreicht. Um eine funktionierende Konvergenz in einer abschätzbaren Zeit zu gewährleisten, wurde ein einfacher Verteil-Algorithmus verwendet. Problematisch dabei ist, dass jeder Knoten mit jedem Knoten kommuniziert und es so zu einem hoher Kommunikationsaufwand kommt. Die absehbare Konvergenzzeit lässt sich jedoch mit Hilfe der einzelnen Buffer-Intervalle berechnen, womit die Korrektheit der Implementierung besser verifiziert werden kann.

6 Evaluation

Dieser Abschnitt beschreibt die Evaluierung der Implementierung des CAF-CRDT-Moduls. Abschnitt 6.2 beschreibt die Evaluierung einzelner implementierter CRDTs mit Hilfe von Unit-Tests. Anschließend ist in Abschnitt 6.3 erläutert, wie die Konvergenz der Replikate verifiziert wurde. In Abschnitt 6.4 wird das Datenaufkommen im Netzwerk anhand verschiedener Systemgrößen (Anzahl von Knoten im verteilten System) betrachtet, dabei werden δ-CRDTs mit *CvRDTs* verglichen. In Abschnitt 6.5 wird ein Multi-Writer-Fall getestet, bei welchem alle Knoten im verteilten System schreibend auf ein Replikat zugreifen. Das lokale Verteilen zum δ-CRDT-Zuständen ist in Abschnitt 6.6 behandelt. Abschnitt 6.7.2 behandelt die Verteilung von δ-CRDT-Zuständen im verteilten System, dabei wird geprüft, ob die erwartete Konvergenzgeschwindigkeit eingehalten werden kann.
Die Benchmarks sind in einem eigenen GitHub-Repository[1] zu finden.

6.1 Testumgebung

Die Tests und Messungen wurden unter Debian 9 (Stretch, 4.9.0-2-amd64) auf einer Intel Core i5-3570 CPU mit 3,4GHz durchgeführt. Das Testsystem verfügt über 16 GByte RAM. Als C++-Compiler wurde GCC 6.3.0 (20170406) eingesetzt. Mininet [41] wurde in der Version 2.3.0d1 verwendet.

6.2 Unit-Tests

Die Unit-Tests wurden mit dem von CAF bereitgestellten Unit-Test-Framework implementiert. Damit gliedern sich die Unit-Tests in die CAF-Test-Suite ein. Für die einzelnen vom CAF-CRDT-Modul implementierten CRDTs wird in einem lokalen Aufbau getestet, ob CRDTs zum erwarteten Zustand konvergieren. Dabei werden auch Fälle getestet, die mit einem nicht als CRDT implementierten Datentyp zu einem Konflikt führen würden, welcher repariert werden müsste.

[1] https://github.com/Hamdor/caf-crdt-benchmarks

Im Folgenden sind exemplarische Beispiele der Unit-Tests für die von CAF implementierten CRDTs gezeigt.

6.2.1 GSet

Fall	A	B	$B \setminus A$
1	$\{1, 2, 3, 4\}$	$\{5, 6, 7, 8\}$	$\{5, 6, 7, 8\}$
2	$\{\}$	$\{5, 6, 7, 8\}$	$\{5, 6, 7, 8\}$
3	$\{1, 2, 3, 4\}$	$\{\}$	$\{\}$
4	$\{1, 2, 3, 4\}$	$\{1, 2, 5\}$	$\{5\}$
5	$\{\}$	$\{\}$	$\{\}$

Tabelle 6.1: Testfälle mit Eingabe und erwarteter Ausgabe für die GSet-Klasse

Tabelle 6.1 zeigt die Testfälle für den GSet Unit-Test. Zu sehen sind vier Spalten, wobei die erste Spalte den Fall bezeichnet, die zweite Spalte beschreibt den ersten Eingabeparameter (A), die dritte Spalte den zweiten Eingabeparameter (B). Die letzte Spalte listet das erwartete Ergebnis der Operation $B \setminus A$.
Der erwartete Wert nach Zusammenführen zweier Instanzen ($A \leftarrow A \cup B$) wird durch $\delta \leftarrow A \setminus B$ beschrieben. Die symmetrische Differenz ($\delta \leftarrow A \triangle B$) wird nicht benötigt, ist aber ebenfalls eine gültige Ausgabe nach dem Zusammenführen.

Auflistung 6.1: Unit-Test der GSet-Klasse

```
1  void test_merge(const std::set<int>& lhs_, const std::set<int>& rhs_,
2                  const std::set<int>& assumed_) {
3    gset<int> lhs, rhs;
4    lhs.subset_insert(lhs_);
5    rhs.subset_insert(rhs_);
6    auto delta = lhs.merge(rhs);
7    for (auto& i : assumed_)
8      CAF_CHECK(delta.element_of(i));
9    CAF_CHECK(delta.size() == assumed_.size());
10 }
11
12 CAF_TEST(merge) {
13   test_merge({1,2,3,4}, {5,6,7,8}, {5,6,7,8}); // Fall 1
14   test_merge({}, {5,6,7,8}, {5,6,7,8});        // Fall 2
15   test_merge({1,2,3,4}, {}, {});               // Fall 3
16   test_merge({1,2,3,4}, {1,2,5}, {5});         // Fall 4
17   test_merge({}, {}, {});                      // Fall 5
18 }
```

Abbildung 6.1 zeigt den Unit-Test für die zuvor spezifizierten Testfälle. Zu sehen ist die Funktion `test_merge` (Zeile 1-10), welche als Argumente zwei Eingabewerte `lhs_` und `rhs_` erwartet. Diese verhalten sich zur zuvor gezeigten Tabelle 6.1 wie A und B. Das dritte Argument beschreibt das erwartete Ergebnis der Operation $B \setminus A$. In der Funktion werden zwei `gset <int>` Instanzen erzeugt und mit den zu testenden Werten gefüllt (Zeile 3-5). Die Operation `auto delta = lhs.merge(rhs)` (Zeile 6) ist dabei äquivalent zu $(A \leftarrow A \cup B) \wedge (\delta \leftarrow B \setminus A)$. Es wird geprüft, ob alle erwarteten Ergebnisse aus `assumed_` in `delta` enthalten sind (Zeile 7-8). Anschließend wird geprüft, ob die Zahl der Elemente von `delta` mit der der erwarteten Ergebnisse übereinstimmt (Zeile 9). Alle Testfälle werden in die Funktion `test_merge` übergeben (Zeile 13-18).

6.2.2 GCounter

Auflistung 6.2: Unit-Test der GCounter-Klasse

```
1  CAF_TEST(merge) {
2    auto dummy_actor = [](event_based_actor*) {};
3    gcounter<int> lhs, rhs;
4    lhs.set_owner(system.spawn(dummy_actor));
5    rhs.set_owner(system.spawn(dummy_actor));
6    lhs.increment();
7    CAF_CHECK(lhs.count() == 1);
8    rhs.increment();
9    CAF_CHECK(rhs.count() == 1);
10   auto delta = lhs.merge(rhs);
11   CAF_CHECK(lhs.count() == 2);
12   CAF_CHECK(rhs.count() == 1);
13   CAF_CHECK(delta.count() == 1);
14   delta = rhs.merge(lhs);
15   CAF_CHECK(lhs.count() == 2);
16   CAF_CHECK(rhs.count() == 2);
17   CAF_CHECK(delta.count() == 1);
18 }
```

Abbildung 6.2 zeigt den Unit-Test für die GCounter-Klasse. Die hierbei kritische Situation ist, wenn zwei (logisch) nebenläufige ändernde Operationen (Inkrementierungen) stattfinden. Die Abbildung zeigt, wie zuerst die beiden *GCounter* `lhs_` und `rhs_` initialisiert werden (Zeile 3). Da die *GCounter* intern mit einer Map arbeiten, bei welcher ein Aktor-Handle als Schlüssel benötigt wird, muss ein Besitzer der Instanzen gesetzt werden, dieser wird intern als Schlüssel verwendet. Der Besitzer wird mit Hilfe der Funktion `set_owner` gesetzt, welche nur für Unit-Tests verwendet werden darf (Zeile 4-5). Damit beide Instanzen ihren eigenen Schlüssel für den Zugriff auf die interne Map besitzen, wird jeweils ein `dummy_actor` (Zeile 2) erstellt. Der `dummy_actor` hat dabei keine eigene Funktionalität.
Nach der Initialisierung werden beide *GCounter* um eins inkrementiert, danach wird geprüft, ob beide jeweils den Wert eins zählen (Zeile 6-9). Anschließend werden Instanz `lhs_` (A) mit `rhs_` (B) zusammengeführt (Zeile 10, $A \leftarrow A \sqcup B \wedge \delta \leftarrow B \setminus A$). Die Instanz `lhs_` muss nun beide Inkrementierungen enthalten und den Wert zwei zählen (Zeile 11). Die Instanz `rhs_` muss weiterhin unverändert sein (Zeile 12). Anschließend wird geprüft, ob das `delta` den korrekten Zustand beinhaltet. Es wird angenommen, dass `delta` `rhs_` komplett beinhaltet

(Zeile 13).
Anschließend wird `rhs_` und `lhs_` zusammengeführt (Zeile 14). Beide Instanzen sollten nun den Wert zwei zählen (15-16).

6.2.3 MV-Register

Auflistung 6.3: Unit-Test der MV-Register-Klasse

```
1  CAF_TEST(merge) {
2    auto dummy_actor = [](event_based_actor*) {};
3    mv_register<int> lhs, rhs;
4    lhs.set_owner(system.spawn(dummy_actor));
5    rhs.set_owner(system.spawn(dummy_actor));
6    lhs.set(1);
7    CAF_CHECK(lhs.get_set().size() == 1);
8    CAF_CHECK(lhs.get() == 1);
9    rhs.set(2);
10   CAF_CHECK(rhs.get_set().size() == 1);
11   CAF_CHECK(rhs.get() == 2);
12   lhs.merge(rhs);
13   CAF_CHECK(lhs.get_set().size() == 2);
14   if (lhs.get_set().size() == 2) {
15     auto first  = *lhs.get_set().begin();
16     auto second = *(++lhs.get_set().begin());
17     CAF_CHECK(std::get<0>(first) == 1);
18     CAF_CHECK(std::get<0>(second) == 2);
19     CAF_CHECK(std::get<1>(first).compare(std::get<1>(second))
20                == vector_clock_result::concurrent);
21   }
22   rhs.merge(lhs);
23   CAF_CHECK(rhs.get_set().size() == 2);
24   rhs.set(5);
25   CAF_CHECK(rhs.get_set().size() == 1 && rhs.get() == 5);
26 }
```

Auflistung 6.3 zeigt den Unit-Tests für das MV-Register. Die kritische Situation im MV-Register ist es, wenn zwei Werte (logisch) nebenläufig gesetzt wurden und später zusammengeführt werden. Zu sehen sind zwei `mv_register<int>` Instanzen `lhs_` und `rhs_` (Zeile 3). Den beiden Instanzen wird wie in dem vorherigen Unit-Test ein Besitzer zugeteilt. Dieser ist notwendig, damit in der internen Vektoruhr der *MV-Register* der passende Slot inkrementiert werden kann.

Zunächst wird der Wert eins in das Register `lhs_` geschrieben (Zeile 6). Das Register sollte nun einen einzelnen Wert gesetzt haben (Zeile 7). Durch die Funktion `get_set()` wird eine Menge (`std::set<std::tuple<T, vector_clock>>`) zurückgegeben, welche zu jedem nebenläufig zugewiesenen Wert den Zeitstempel enthält. Anschließend wird geprüft, ob der gesetzte Wert dem tatsächlich erwarteten Wert entspricht (Zeile 8).
Der Wert 2 wird in Register `rhs_` geschrieben (Zeile 9), woraufhin zunächst die Größe des Registers und der Wert geprüft wird (Zeile 10-11).
Register `rhs_` wird nun in den Zustand von `lhs_` zusammengeführt, da beide Zuweisungen (logisch) nebenläufig geschehen sind, muss `lhs_` nun zwei Werte enthalten (Zeile 13). Es muss außerdem geprüft werden, ob die Zeitstempel der beiden Werte im Register nebenläufig sind. Dazu werden die beiden Einträge aus dem Register extrahiert (Zeile 15-16). Es wird geprüft, ob die erwarteten Werte eins und zwei gesetzt sind (Zeile 17-18). Anschließend wird geprüft, ob beide Vektoruhr-Zeitstempel nebenläufig sind (Zeile 19-20).
Der Unit-Test prüft außerdem, ob sich das zusammengeführte `lhs_` mit `rhs_` zusammenführen lässt (Zeile 22). `rhs_` sollte nun ebenfalls zwei Werte enthalten (Zeile 23).
Zuletzt prüft der Unit-Test, ob sich die zusammengeführten Werte überschreiben lassen (Zeile 24-25).

6.3 Verifikation der Konvergenz

Um zu testen, ob alle mit dem CAF-CRDT-Modul verteilten Replikate konvergieren, wird ein *GCounter* verwendet, welcher pro Knoten mehrfach hochgezählt wird. Da bekannt ist, wie viele Knoten verwendet werden und wie oft diese hochzählen, kann geprüft werden, ob alle Replikate irgendwann denselben Wert halten und damit konvergiert sind.

Auflistung 6.4: Aktor für den Konvergenztest

```
1  class incrementer : public notifiable<gcounter<int>>::base {
2  public:
3    incrementer(actor_config& cfg)
4      : notifiable<gcounter<int>>::base(cfg),
5        state_(this, "gcounter<int>://counter") {
6      // nop
7    }
8  protected:
9    notifiable<gcounter<int>>::behavior_type make_behavior() override {
10     state_.increment_by(1);
11     return {
12       [&](notify_atom, const gcounter<int>& t) {
13         state_.merge(t);
14         if (state_.count() == expected)
15           quit();
16       }
17     };
18   }
19 private:
20   gcounter<int> state_;
21 };
```

Auflistung 6.4 zeigt den Aktor, welcher für die Verifikation der Konvergenz verwendet wird. Zu sehen ist ein Klassenbasierter Aktor `incrementer` (Zeile 1), welcher einen `gcounter<int> state_` in seinem Zustand hält (Zeile 20). Wird der Aktor erstellt, inkrementiert er den *GCounter* um eins (Zeile 10). Er überwacht Änderungen auf dem Replikat und stellt einen Nachrichten-Handler für den Replikat-Typen bereit (Zeile 12). Wird eine Änderung am *GCounter* vorgenommen, wird der *incrementer* benachrichtigt und führt den empfangenen Zustand mit seinem lokalen Zustand zusammen (Zeile 13). Sobald der erwartete Wert im *GCounter* vorhanden ist, wird der Aktor beendet (Zeile 14-15).

Der Test terminiert nur, nachdem alle Aktoren von Typ *incrementer* den erwarteten Wert gesehen haben und damit terminiert sind.

6.4 Datenaufkommen im Netzwerk

Dieser Abschnitt diskutiert die Messung des Datenaufkommens im Netzwerk. Es werden δ-CRDTs und *CvRDTs* gegenüber gestellt, um zu zeigen, dass δ-CRDTs wesentlich weniger Verkehr im Netzwerk erzeugen als *CvRDTs*. Außerdem wird demonstriert, dass sich δ-CRDTs mit dem implementierten Algorithmus effizienter verteilen lassen als *CvRDTs*. Die zugehörigen Tests werden mit verschiedenen Systemgrößen ausgeführt (Anzahl an Knoten).

6.4.1 Vorgehen und Erwartung

Der Benchmark verwendet ein repliziertes *GSet* über Integer (4 Byte). Pro Messung variiert die Anzahl an Knoten, wobei die Anzahl an Operationen auf CvRDT und δ-CRDT gleich bleibt.
Es ist zu erwarten, dass CvRDTs ein höheres Datenaufkommen im Netzwerk verursachen, welches sowohl mit der Anzahl an ausgeführten Operationen als auch mit der Anzahl an Knoten im System skaliert. δ-CRDTs sollten dabei die gleiche Skalierung aufweisen, es sollte jedoch ein wesentlich geringeres Datenaufkommen zu beobachten sein, da weniger Daten versendet werden.
Alle Operationen werden bei diesem Benchmark auf einem Knoten ausgeführt, dies macht den Benchmark zu einem Single-Writer-Fall.

6.4.2 Aufbau

Mit Hilfe von Mininet [41] wurde ein verteiltes System simuliert. Mininet simuliert neben Rechnern auch im Netzwerk befindliche Einheiten wie Switches. Es erlaubt außerdem, Paketverlust und Verzögerung auf dem Netzwerk zu simulieren, was aber für diesen Test nicht verwendet wurde.

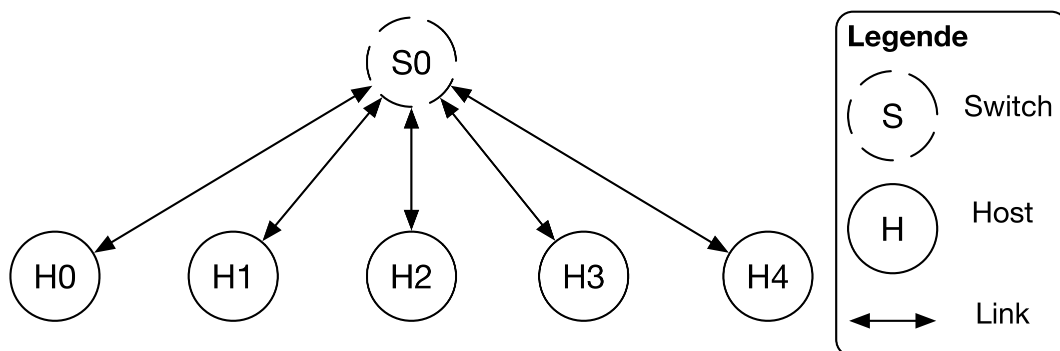

Abbildung 6.1: Topologie des mit Mininet erstellten Netzwerks

Abbildung 6.1 zeigt die Topologie des mit Mininet erstellen Netzwerks. Für alle verteilten Benchmarks wird die dargestellte Topologie verwendet. Zu sehen ist ein Switch (gestrichelter Kreis) sowie Hosts (geschlossene Kreise), welche mit bidirektionalen Links (Pfeile) verbunden sind.
Mininet wird für jeden Durchlauf separat gestartet, dabei variiert für die Durchläufe die Anzahl an Knoten (Hosts). Die Hosts sind wie im Bild zu sehen, alle mit dem selben Switch verbunden. Jeder Host hat eine eindeutige Nummer, welche in dem jeweiligen Kreis zu sehen ist.

6.4.3 Ergebnisse & Diskussion

Das erwartete Datenaufkommen des replizierten *GSet* über *Integer* als *CvRDT* lässt sich wie folgt berechnen:

$$\frac{x^2 + x}{2} * (K - 1) * sizeof(int) \tag{6.1}$$

Dabei steht x für die Anzahl an Operationen auf dem *GSet*, K ist die Anzahl an Knoten im verteilten System und $sizeof(int)$ steht für die Größe eines *Integers* in Bytes. Ist das *GSet* mit einem anderen Typen spezialisiert, so muss die Größe dieses Typs in Bytes verwendet werden.

Anzahl Operationen (x)	Anzahl Knoten (K)	$sizeof(int)$	Wert (in MByte)
1000	10	4	18,018
1000	20	4	38,038
1000	40	4	78,078
10000	10	4	1800,18
10000	20	4	3800,38
10000	40	4	7800,78

Tabelle 6.2: Erwartetes Datenaufkommen bei Verwendung von *CvRDTs* in Bezug zu verschiedener Anzahl an Knoten und verschiedener Anzahl an Operationen

Tabelle 6.2 zeigt das erwartete Datenaufkommen für den *CvRDT* Benchmark. Es ist davon auszugehen, dass die folgenden Messergebnisse sich an den Größenordnungen der Tabelle orientieren. Die Messergebnisse sollten dabei jedoch über den tatsächlichen Messwerten der Tabelle liegen, da es sich hierbei nur um die reinen Daten des *GSet* handelt. Datenaufkommen, das durch Aufbau der Verbindung, Austausch von Nachrichten im Allgemeinen oder die Übertragung von Replikat-Bezeichnern entsteht, wurde bei der Berechnung der Tabellenwerte nicht berücksichtigt, ist aber in den totalen Messwerten enthalten.

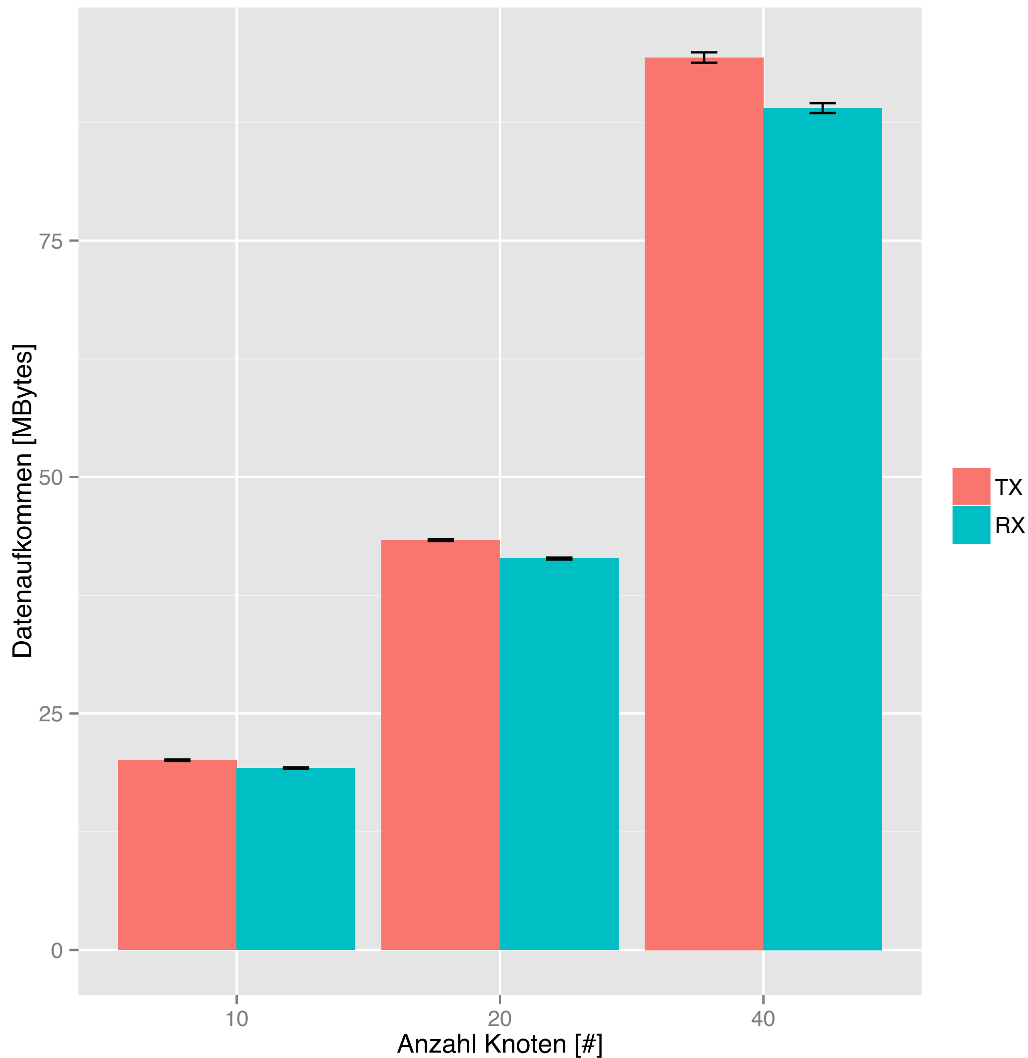

Abbildung 6.2: Gemessenes Datenaufkommen im Netzwerk bei Verwendung von *CvRDTs*, dabei wurden 1000 Operationen auf dem *GSet* durchgeführt

Abbildung 6.2 zeigt das akkumulierte Datenaufkommen im Netzwerk zwischen allen Links als Balkendiagramm. Es wird zwischen *TX* (gesendet, rot) und *RX* (empfangen, türkis) unterschieden. Die X-Achse zeigt die verwendete Anzahl an Knoten, wobei pro Durchlauf die Anzahl an Knoten verdoppelt wurde. Die Y-Achse zeigt das Datenaufkommen in Megabytes (MBytes). Die Durchläufe wurden pro Skalierungsstufe zehnmal wiederholt, die Abweichung der Durchläufe lässt sich anhand der Fehlerbalken auf der Y-Achse ablesen.
Zu sehen sind die Werte beim Versenden von *CvRDTs*, welche bei Änderungen immer den

vollständigen Zustand versenden. Die empfangenen und versendeten Werte bewegen sich auf gleichem Niveau. Bei zehn Knoten bewegen sich TX und RX in der Größenordnung von 20 MByte. Weiterhin ist erkennbar, dass sich bei Verdoppelung der Knoten TX und RX ebenfalls auf ein Niveau von ungefähr 42 MByte verdoppeln. Die nächste Skalierungsstufe mit 40 Knoten zeigt ein TX- und RX-Niveau auf ungefähr 90 MByte und hat sich damit erneut ungefähr verdoppelt.

Damit konnten die zuvor berechneten Werte aus Tabelle 6.2 für 1000 Operationen eingehalten werden.

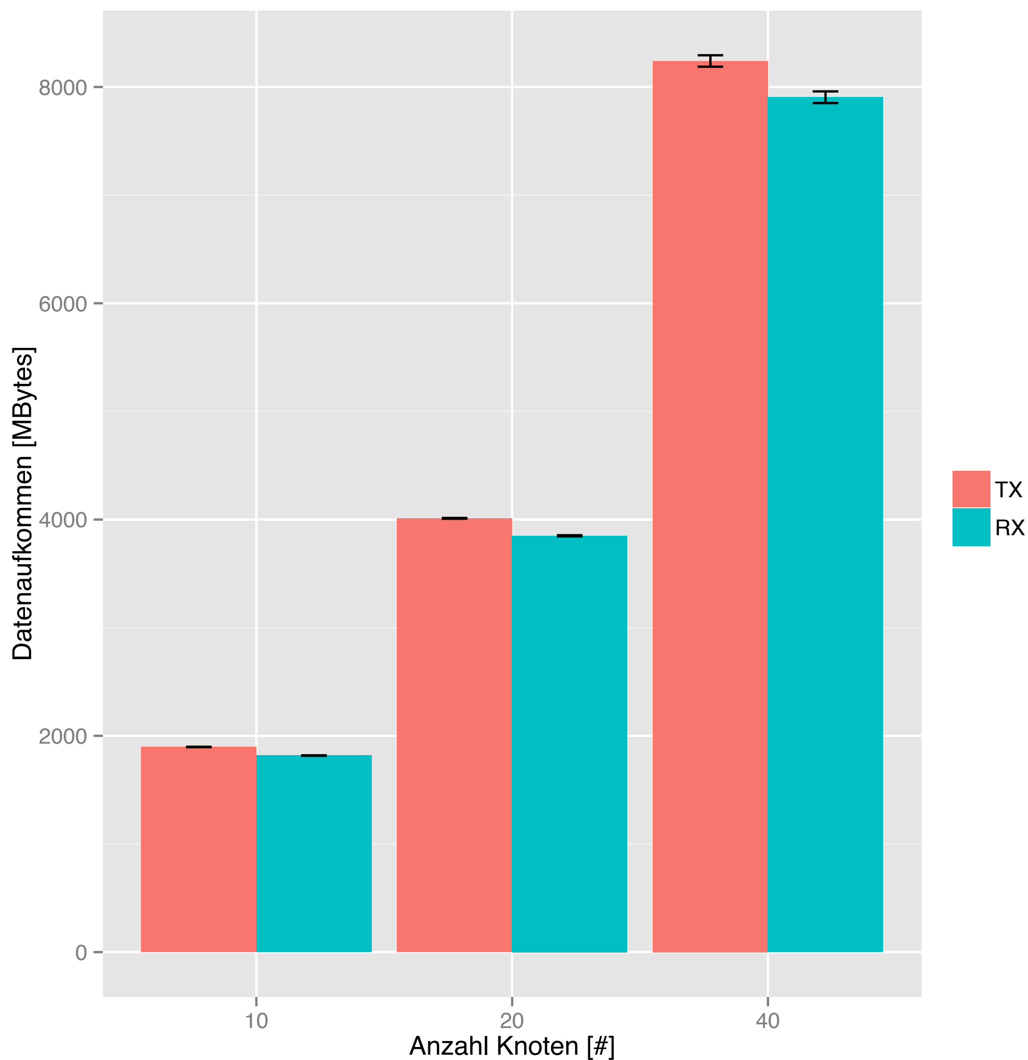

Abbildung 6.3: Gemessenes Datenaufkommen im Netzwerk bei Verwendung von *CvRDTs*, dabei wurden 10000 Operationen auf dem *GSet* durchgeführt

Abbildung 6.3 zeigt erneut das akkumulierte Datenaufkommen im Netzwerk zwischen allen Links als Balkendiagramm. Es wird zwischen *TX* (gesendet, rot) und *RX* (empfangen, türkis) unterschieden. Die X-Achse zeigt die verwendete Anzahl an Knoten, wobei pro Durchlauf die Anzahl an Knoten verdoppelt wurde. Die Y-Achse zeigt das Datenaufkommen in Megabytes (MBytes). Die Durchläufe wurden pro Skalierungsstufe 10 mal wiederholt, die Abweichung der Durchläufe lässt sich anhand der Fehlerbalken auf der Y-Achse ablesen.
Es ist zu sehen, wie sich TX und RX bei zehn Knoten bereits bei 1900 MByte befinden. Bei

Verdoppelung der Knoten steigen TX und RX auf etwa 4000 MByte. Die letzte Stufe der Skalierung mit 40 Knoten zeigt TX und RX auf dem Niveau von etwa 8200 MByte.
Die erwarteten Werte aus Tabelle 6.2 für 10000 Operationen konnte somit erneut eingehalten werden.

Operationen	Knoten	Erwarteter Wert (MByte)	Mittelwert (TX, MByte)	Abweichung
1000	10	18,018	20,06248	~11,35%
1000	20	38,038	43,32373	~13,9%
1000	40	78,078	94,35059	~20,84%
10000	10	1800,18	1896,959	~5,38%
10000	20	3800,38	4011,080	~5,54%
10000	40	7800,78	8240,527	~5,63%

Tabelle 6.3: Gegenüberstellung von erwarteten und gemessenen Werten des *GSet* als *CvRDT*

Tabelle 6.3 zeigt die erwarteten und die gemessenen Werte (als Mittelwert der zehn Durchläufe) des replizierten *GSets* als *CvRDT*. Die Tabelle zeigt außerdem die Abweichung in Prozent zwischen gemessenen Wert und dem errechneten Wert. Es zeigt sich, dass die Abweichung kleiner wird, sobald die Anzahl an Operationen erhöht wird. Die Abweichung steigt hingegen leicht bei der Erhöhung der Anzahl an Knoten. Dies lässt sich damit erklären, dass für eine höhere Anzahl an Knoten allgemein mehr Verbindungen bestehen und mehr Meta-Informationen ausgetauscht werden müssen.
Die kleinere Abweichung bei einer hohen Anzahl von Operationen lässt sich damit erklären, dass die Operationen einen hohen Anteil am Datenaufkommen im Netzwerk haben.

Für das δ-CRDT *GSet* über *Integer* lässt sich ebenfalls der erwartete Datenverkehr abschätzen:

$$x * (K - 1) * sizeof(int) \tag{6.2}$$

Wobei x wieder die Anzahl der Operationen, K die Anzahl der Knoten, und $sizeof(int)$ die Größe des spezialisierten Typs ist.

Anzahl Operationen (x)	Anzahl Knoten (K)	$sizeof(int)$	Wert (in KByte)
1000	10	4	36
1000	20	4	76
1000	40	4	156
10000	10	4	360
10000	20	4	760
10000	40	4	1560

Tabelle 6.4: Erwartetes Datenaufkommen bei Verwendung von δ-CRDTs in Bezug zu verschiedener Anzahl an Knoten und verschiedener Anzahl an Operationen

Tabelle 6.4 zeigt die erwarteten Werte für das replizierte *GSet* über *Integer* als δ-CRDT. Die erwarteten Werte sind wesentlich niedriger im Vergleich zum *CvRDT GSet* und in Kilobyte (KByte) angegeben.

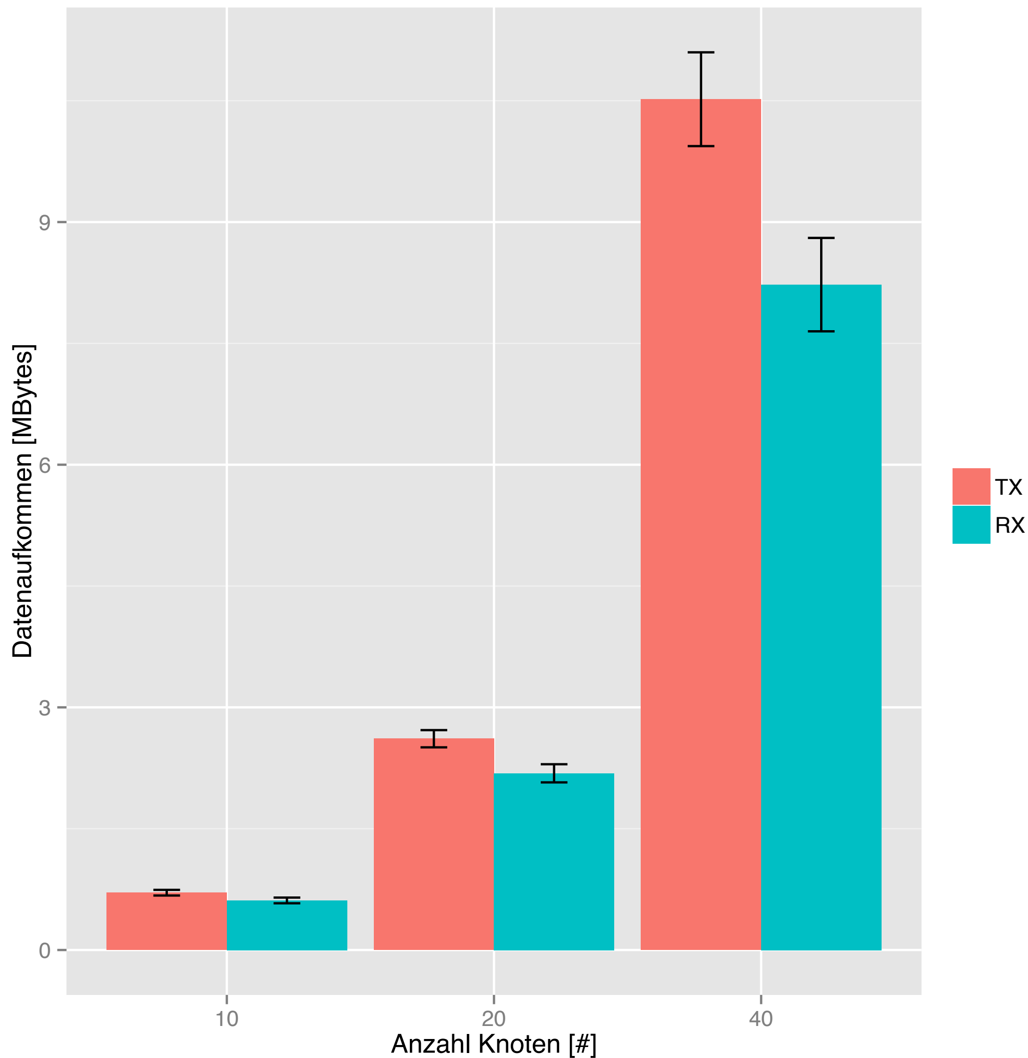

Abbildung 6.4: Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 1000 Operationen auf dem *GSet* durchgeführt

Abbildung 6.5 zeigt das akkumulierte Datenaufkommen im Netzwerk bei Verwendung eines *GSet* über *Integer* als δ-CRDT. Wie bei den vorherigen Balkendiagrammen ist wieder zwischen TX (Rot) und RX (Türkis) aufgeschlüsselt. Die X-Achse zeigt die Anzahl an verwendeten Knoten. Die Y-Achse zeigt das Datenaufkommen in Megabyte (MByte). Die Messpunkte besitzen wieder einen Fehlerbalken, welcher die Schwankungen der zehn Durchläufe darstellt. In dieser Messung wurden 1000 schreibende Operationen auf dem *GSet* ausgeführt.
Es ist zu sehen, wie sich TX bei zehn Knoten etwa bei 0,7 MByte befindet. Bei 20 Knoten ist TX

bereits bei 2,6 MByte. Bei 40 Knoten ist TX etwa auf 10,5 MByte. Der Wert vervierfacht sich bei Verdoppelung der Anzahl an Knoten. Die gemessenen Werte liegen wesentlich höher als die erwarteten. Dies lässt sich darauf zurückführen, dass andere Operationen wie Austausch von sonstigen Nachrichten, Meta-Informationen sowie der Aufbau und Erhalt der Verbindungen einen im Vergleich zu den erwarteten Datenaufkommen des *GSets* großen Teil ausmachen.

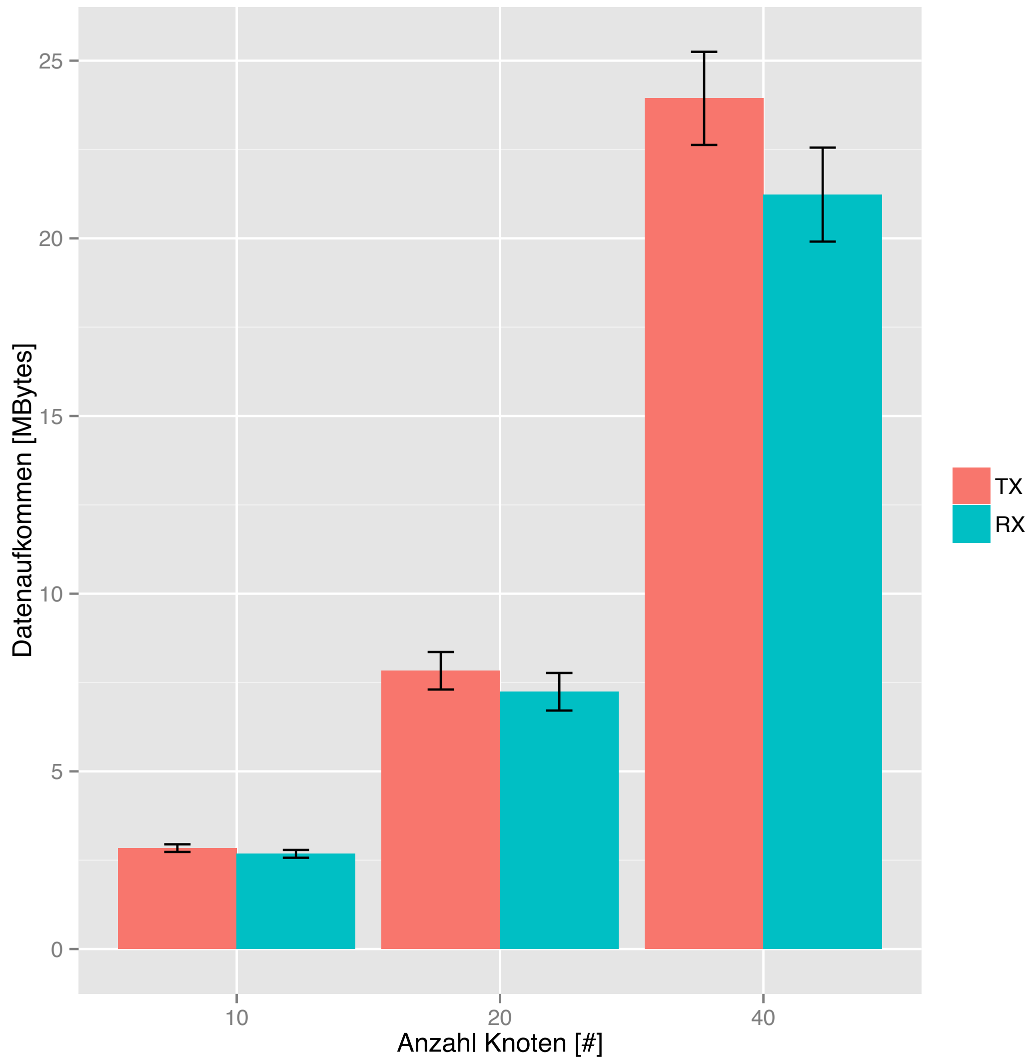

Abbildung 6.5: Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 10000 Operationen auf dem *GSet* durchgeführt

Abbildung 6.5 zeigt das Datenaufkommen im Netzwerk akkumuliert zwischen allen Links als Balkendiagramm. Es ist erneut zu sehen, wie zwischen *TX* (gesendet, rot) und *RX* (empfangen, türkis) unterschieden ist. Die X-Achse zeigt erneut die Anzahl an Knoten, wobei pro Durchlauf die Anzahl an Knoten verdoppelt wurde. Die Y-Achse zeigt das Datenaufkommen in Megabytes (MBytes). In dieser Messung wurden 10000 schreibende Operationen auf dem *GSet* ausgeführt. Es ist zu sehen, wie TX sich bei zehn Knoten bei 2,8 MByte befindet. Bei Verdoppelung der Anzahl an Knoten auf 20 befindet sich TX bei etwa 7,8 MByte. Bei erneuter Verdoppelung auf 40 Knoten ist TX auf etwa 24 MByte angestiegen. Pro Verdoppelung ist damit eine Verdreifachung des Datenaufkommens zu sehen.

Operationen	Knoten	Erwarteter Wert (KByte)	Mittelwert (TX, KByte)	Abweichung
1000	10	36	708,4907	∼1968,68%
1000	20	76	2611,9375	∼3436,76%
1000	40	156	10518,6059	∼6742,7%
10000	10	360	2839,784	∼788,82%
10000	20	760	7831,178	∼1030,42%
10000	40	1560	23939,126	∼1534,56%

Tabelle 6.5: Gegenüberstellung von erwarteten und gemessenen Werten des *GSet* als δ-CRDT

Tabelle 6.5 zeigt die Mittelwerte von TX in Kilobyte (KB) für das replizierte *GSet* über *Integer* als δ-CRDT. Die erwarteten Werte sind gegenübergestellt und die Abweichung zu dem Wert ist in Prozent angegeben.
Aus der Tabelle lässt sich schließen, dass das Datenaufkommen, welches für das Replizieren des *GSets* notwendig war, gering ist im Vergleich zu den sonstigen Übertragungen.
Es ist außerdem zu sehen, dass die Abweichung zum erwarteten Wert stetig sinkt, wenn die Anzahl an Operationen erhöht werden. Die Steigung der Abweichung ist bei hoher Anzahl an Operationen ebenfalls geringer als bei niedriger Anzahl an Operationen.
Dies lässt sich damit erklären, das der Anteil des Datenaufkommens, welcher für das Replizieren des *GSets* notwendig ist, gering ist im Vergleich zu dem restlichen Datenaufkommen im Netzwerk.

Es hat sich gezeigt, dass δ-CRDTs weniger Datenaufkommen im Netzwerk generieren als *CvRDTs*. Außerdem haben δ-CRDTs einen geringeren Anteil am gesamten Datenaufkommen zwischen den Knoten.

6.5 Datenaufkommen im Multi-Writer-Fall

Dieser Abschnitt diskutiert die Messung des Datenaufkommens im Netzwerk. Dabei ist der Aufbau der selbe wie im vorherigen Benchmark, welcher in Abschnitt 6.4 beschrieben ist. Dabei soll im Gegensatz zum vorherigen Benchmark ein Fall getestet werden, in dem eine feste Anzahl an Operationen auf verschieden Knoten ausgeführt werden.

6.5.1 Vorgehen und Erwartung

Der Benchmark verwendet einen replizierten *GCounter* über *Integer* (4 Byte). Pro Messung variiert die Anzahl an Knoten, wobei die Anzahl an Operationen gleich bleibt. Die Anzahl an Operationen, die ein Knoten ausführen muss, ergibt sich, indem man die Anzahl an Operationen durch die Anzahl an Knoten teilt.
Der Benchmark wurde mit zwei Problemgrößen ausgeführt, wobei zuerst 1000 Operationen getestet wurden. Für die zweite Problemgröße wurde eine Verzehnfachung der ursprünglichen Problemgröße gewählt, es wurden also 10000 Operationen getestet. Der in dieser Messung replizierte *GCounter* ist als δ-CRDT implementiert.

6.5.2 Ergebnisse & Diskussion

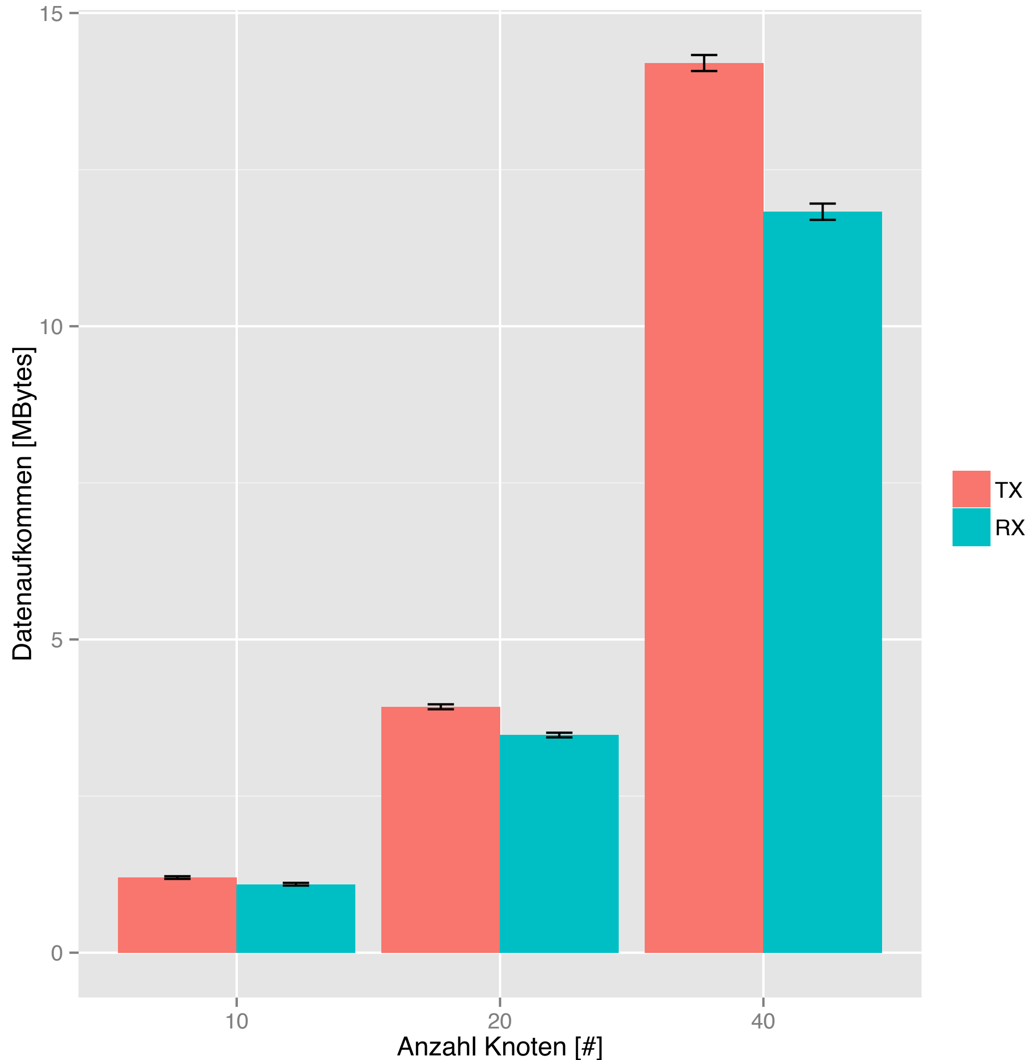

Abbildung 6.6: Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 1000 Operationen auf einem *GCounter* ausgeführt

Abbildung 6.6 zeigt das Datenaufkommen im Netzwerk akkumuliert zwischen allen Links als Balkendiagramm. Die X-Achse zeigt dabei TX (gesendet, rot) und RX (empfangen, türkis), dabei sind Fehlerbalken zu erkennen, welche die Schwankungen der Messwerte aus den 10 Durchläufen darstellen. Die Y-Achse zeigt die verschiedenen Systemgrößen (Anzahl an Knoten). Es ist zu sehen, dass sich das Datenaufkommen in der ersten Skalierungsstufe mit 10 Knoten

auf etwa 1,2 MByte befindet. Wird die Anzahl an Knoten auf 20 verdoppelt, erhöht sich das Datenaufkommen auf 3,8 MByte. Bei erneuter Verdoppelung auf 40 Knoten befindet sich das Datenaufkommen bei etwa 14 MByte.

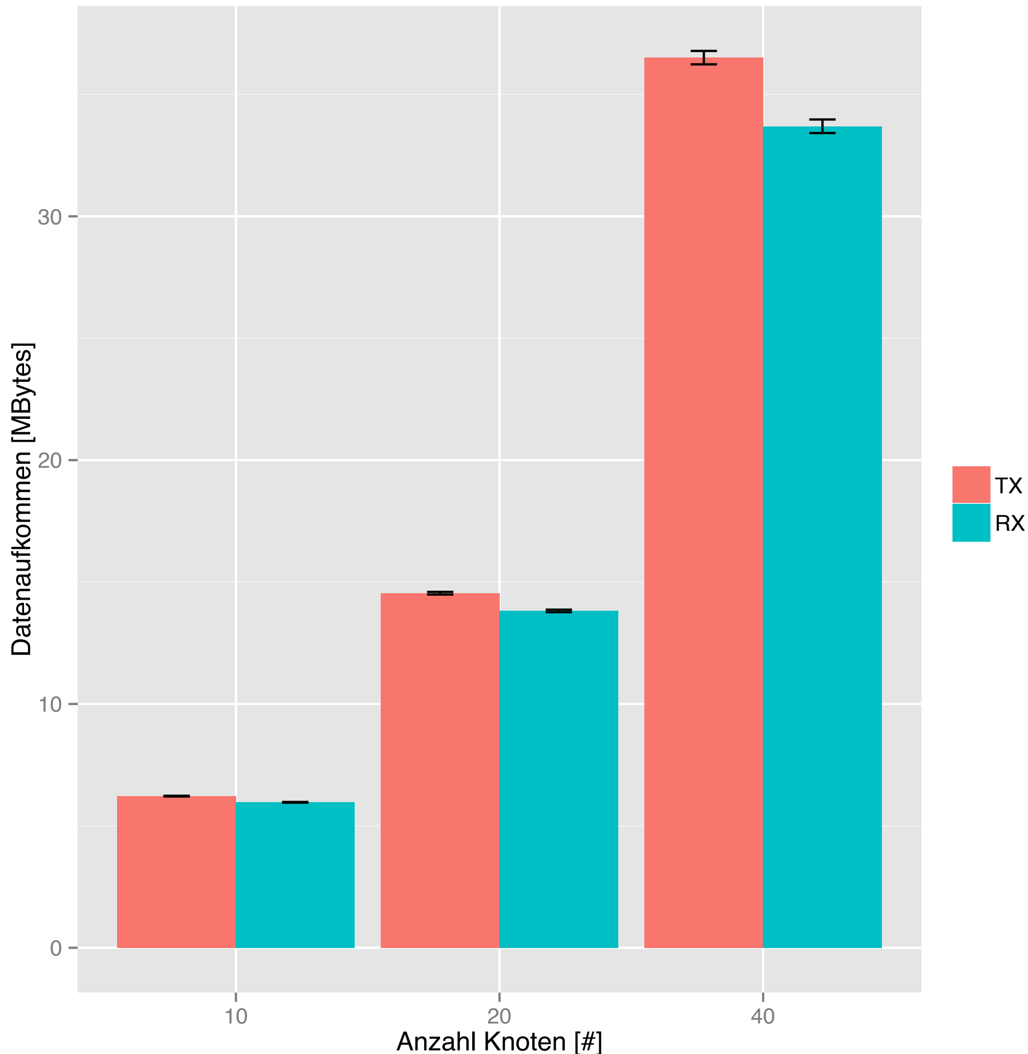

Abbildung 6.7: Gemessenes Datenaufkommen im Netzwerk bei Verwendung von δ-CRDTs, dabei wurden 10000 Operationen auf einem *GCounter* ausgeführt

Abbildung 6.7 zeigt erneut das Datenaufkommen im Netzwerk akkumuliert zwischen allen Links als Balkendiagramm. Es wurden insgesamt 10000 Operationen ausgeführt. Zu sehen ist, dass sich das Datenaufkommen bei zehn Knoten bei etwa 6 MByte befindet. Bei Verdoppelung

auf 20 Knoten liegt es bei etwa 14 MByte. Bei erneuter Verdoppelung auf 40 Knoten erhöht sich das Datenaufkommen auf etwa 36 MByte. Damit ist zu sehen, dass sich bei Verdoppelung der Anzahl von Knoten das Datenaufkommen etwas mehr als verdoppelt. Die Problemgröße hat sich in dieser Messung verzehnfacht, das Datenaufkommen hat sich jedoch nicht verzehnfacht. Dies lässt sich erneut damit erklären, dass der Anteil des zu replizierenden Objekts im Datenaufkommen im Vergleich zum restlichen Datenaufkommen gering ist. Mit einer weiteren Erhöhung der Problemgröße ließe sich wahrscheinlich erkennen, dass sich die Menge an Datenaufkommen ungefähr im gleiche Verhältnis erhöht wie die Anzahl an Operationen, weil der Anteil an restlichem Datenaufkommen dann geringer ist.

Die Skalierung des Datenaufkommens bei Erhöhung der Anzahl an Operationen sowie Anzahl an Knoten verhält sich ähnlich wie im vorherigen Benchmark aus Abschnitt 6.4. Damit hat sich gezeigt, dass das CRDT-Modul im Single-Writer-Fall sowie im Multi-Writer-Fall ähnlich skaliert.

6.6 Performance lokaler Verteilung

Dieser Abschnitt misst den lokalen Laufzeit-Overhead, welcher beim lokalen Verteilten von δ-CRDT-Zuständen an interessierte Abonennten entsteht.

6.6.1 Vorgehen und Erwartung

Dieser Benchmark wird auf einem Knoten in einem einzelnen Prozess ausgeführt. Der Benchmark verwendet ein GSet, in welches Benutzer-Aktoren lesend oder schreibend zugreifen. Jeder Benutzer-Aktor ist dabei lokal am Replikat angemeldet und *cached* den Zustand in seinem eigenen Zustand. Pro Durchlauf variiert die Anzahl an Benutzer-Aktoren (Abonnenten), jede Problemgröße wurde zudem zehnmal separat ausgeführt. Der Benchmark misst die Laufzeit, bis alle Benutzer-Aktoren alle Operationen gesehen haben. Es werden zwei Fälle getestet: (1) Es gibt nur einen Aktor, welcher Operationen ausführt. (2) Jeder Benutzer-Aktor führt eine gewisse Anzahl an Operationen aus. Die Operationen werden nicht zusammengefasst, sondern sofort an Abonnenten versendet. Im ersten Fall wird ein linearer Anstieg der Laufzeit erwartet. Im zweiten Fall wird ein kurvenartiger Anstieg der Laufzeit erwartet, da pro Benutzer-Aktor mehr Operationen ausgeführt werden.

6.6.2 Ergebnisse & Diskussion

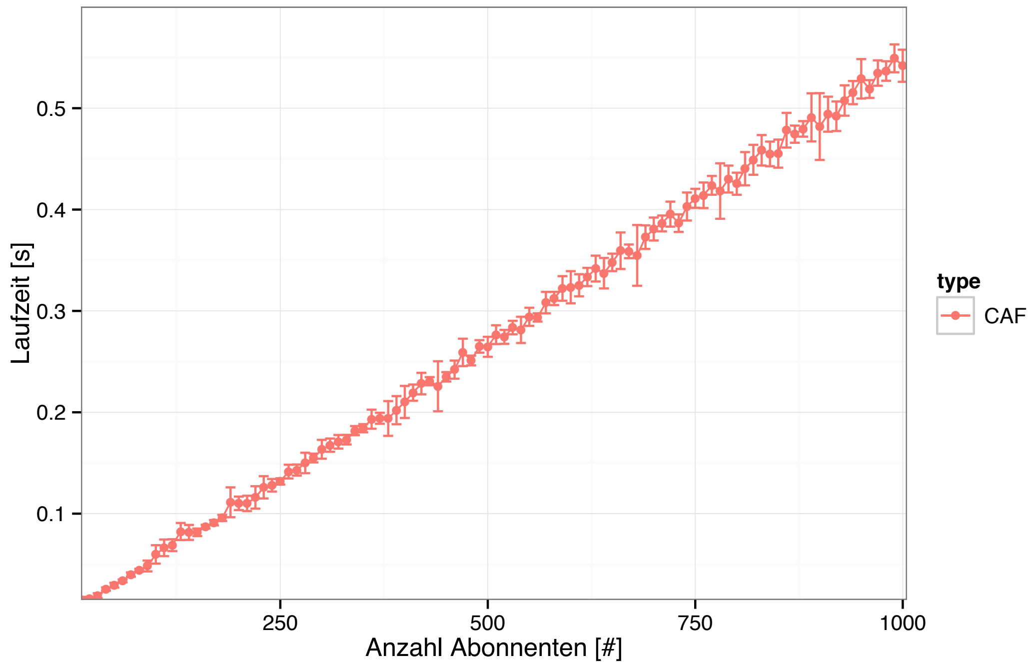

Abbildung 6.8: Total 1000 Operationen, variable Anzahl an Abonnenten

Abbildung 6.8 zeigt die Ergebnisse des ersten Falls. Die Laufzeit der Anwendung ist auf der Y-Achse zu sehen, während die X-Achse die Anzahl der Abonnenten beschreibt. Pro Messpunkt gibt es auf der Y-Achse eine Fehlerbar, welche die verschiedenen Laufzeiten der zehn separaten Durchläufe darstellt. Die Zeiteinheit, welche auf der Y-Achse zu sehen ist, sind Sekunden.
Der Benchmark beginnt bei zehn Benutzer-Aktoren, welche an einem Replikat abonniert sind, in Zehnerschritten wird bis auf 1000 Abonnenten erhöht. 100 Benutzer-Aktoren haben etwa eine Laufzeit von 50ms. Die Laufzeitmarke von 100ms (0.1 Sek) wird bei ungefähr 190 Benutzer-Aktoren erreicht. Ab 250 Benutzer-Aktoren liegt die Laufzeit bei etwa 135ms. Die maximale Laufzeit ist bei 1000 Benutzer-Aktoren mit etwa 550ms erreicht. Es ist ein linearer Anstieg der Laufzeit zu erkennen, welcher sich mit den Erwartungen deckt.

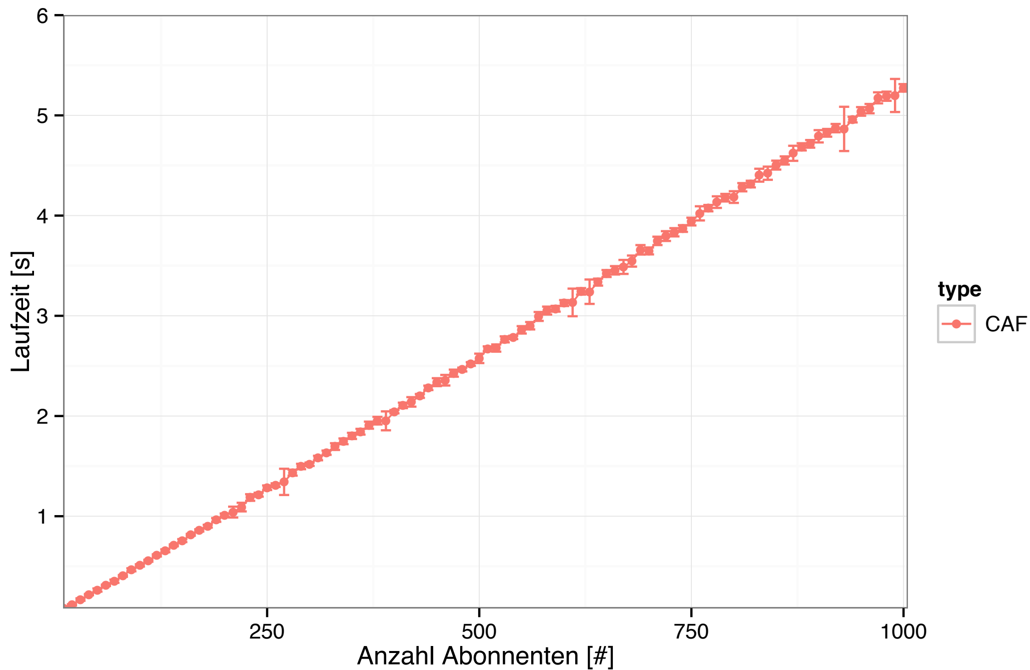

Abbildung 6.9: Total 10000 Operationen, variable Anzahl an Abonnenten

Im folgenden Fall wurde die Problemgröße verzehnfacht, es werden anstelle von 1000 Operationen 10000 Operationen durchgeführt und an alle Abonnenten verteilt. Abbildung 6.9 zeigt wieder die Laufzeit auf der Y-Achse in Sekunden, auf der X-Achse sind die verschiedenen Anzahlen an Benutzer-Aktoren zu sehen. Die Fehlerbalken auf den Messpunkten beschreiben wieder die abweichenden Laufzeiten der zehn Durchläufe.
Der Benchmark beginnt wieder mit zehn Benutzer-Aktoren. Eine Laufzeit von etwa 500ms (0.5 Sek) wird mit etwa 100 Benutzer-Aktoren erreicht. Die Laufzeit von einer Sekunde wird mit etwa 190-200 Benutzer-Aktoren erreicht. Mit 250 Benutzer-Aktoren liegt die Laufzeit bei etwa 1.250 Sekunden. Die Maximale Laufzeit von 5.25 Sekunden wird bei 1000 Benutzer-Aktoren erreicht.
Wie erwartet hat sich bei der Verzehnfachung der Problemgröße die Laufzeit um ungefähr denselben Faktor erhöht. Der Verlauf der Laufzeit ist damit noch immer linear.

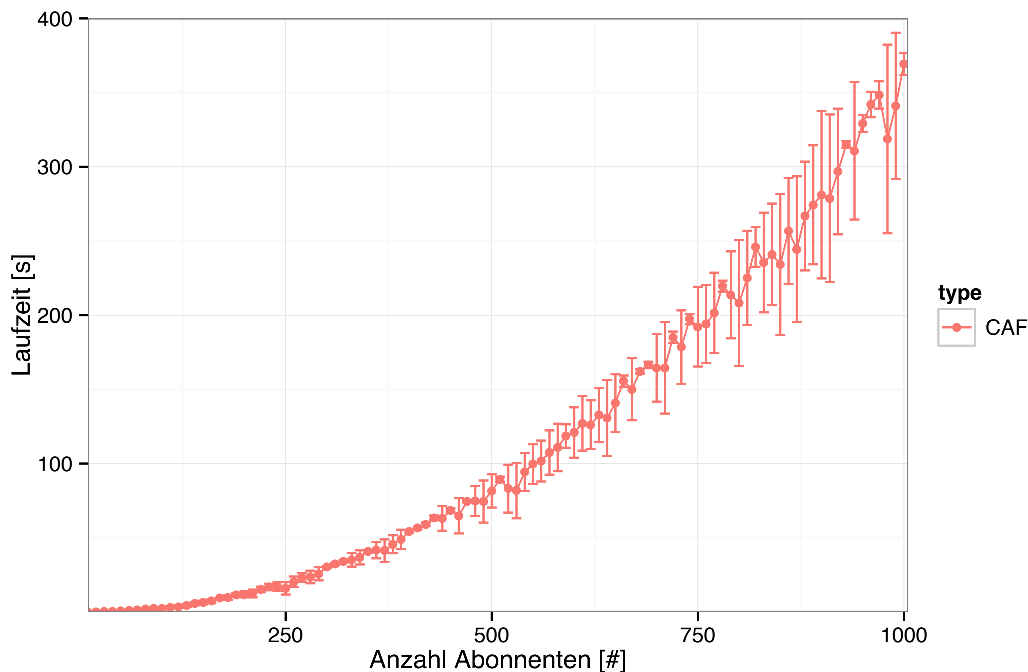

Abbildung 6.10: Jeder Abonnent mit 1000 eigenen Operationen, variable Anzahl an Abonnenten

Im folgenden Benchmark hat jeder Benutzer-Aktor 1000 eigene Operationen auf dem GSet ausgeführt. Es wird erwartet, dass die Laufzeit exponentiell steigt. Abbildung 6.10 zeigt den Verlauf der Laufzeit, dabei ist auf der X-Achse die Anzahl der Abonnenten zu sehen. Die Y-Achse beschreibt die Laufzeit in Sekunden. Die einzelnen Messpunkte besitzen wieder einen Fehlerbalken, welcher die verschiedenen Laufzeiten der zehn separaten Durchläufe darstellt. Es ist zu sehen, wie die Laufzeit von etwa 20 Sekunden bei etwa 250 Abonnenten erreicht wird. Dabei entstanden auf dem GSet 250000 Operationen. Bei 500 Abonnenten liegt die Laufzeit bereits bei ungefähr 80 Sekunden, wobei 500000 Operationen entstanden sind. Ab 750 Abonnenten liegt die Laufzeit überhalb von 200 Sekunden, dabei sind 750000 Operationen entstanden. Bei 1000 Abonnenten liegt die Laufzeit bei etwa 375 Sekunden. Wie erwartet, ist ein immer stärker werdender Anstieg der Laufzeit zu erkennen. Es werden mehr Operationen pro Abonnent generiert, außerdem muss jede Operation zusätzlich zu mehr Abonnenten verteilt werden. Die Fehlerbalken der einzelnen Messpunkte weisen ein hohes Spektrum auf. Bei den Problemgrößen, in denen die Fehlerbalken hohe Unterschiede der Laufzeit ausweisen, war der Rechner voll ausgelastet, außerdem stieg mit steigender Anzahl an Abonnenten der benötigte Speicher. Die hohen Ausschläge der Fehlerbalken lässt sich auf die hohe Auslastung des Rechners und das stetige Allozieren neuen Speichers vom Betriebssystem erklären.

6.7 Auswirkung der Systemgröße auf die Konvergenzgeschwindigkeit

In diesem Abschnitt wird getestet, wie sich die Systemgröße auf die Konvergenzgeschwindigkeit auswirkt. Dabei wird getestet, wie lange es dauert, bis ein generierter δ-CRDT-Zustand zu allen Teilnehmern im System verteilt wurde.

6.7.1 Vorgehen und Erwartungen

In dieser Messung wird erneut ein GSet verwendet. Dabei enthält das GSet einen Zeitstempel der lokalen High-Resolution-Clock (HRE). Wird ein δ-CRDT von einem anderen Aktor (Abonnenten) empfangen, so wird zuerst die aktuelle Zeit der HRE abgefragt. Die Differenz zwischen beiden Zeitstempeln wird gebildet, die Differenz beschreibt dabei, wie lange es gedauert hat, den δ-CRDT-Zustand zu dem Teilnehmer zu propagieren.
Dieser Benchmark ist auf mit Mininet simulierten Knoten ausgeführt. Alle simulierten Knoten verwenden die Uhr des Host-Systems. Wäre dieser Benchmark verteilt, müsste eine synchronisierte Uhr verwendet werden.
In dem Benchmark generiert ein Aktor Zeitstempel, welche dann an andere Knoten und Abonnenten verteilt werden. Es ergeben sich insgesamt vier Testfälle:

1. Es wird gemessen, welche Differenzen bei nur lokaler Verteilung von δ-CRDT-Zuständen entstehen. Dazu wird der Benchmark lediglich in einem Knoten und einer Anwendung ohne Mininet ausgeführt. Die Anzahl an Abonnenten variiert in diesem Fall.

2. Es werden die Differenzen gemessen, die mit verschiedener Anzahl an Knoten entstehen. Der Sendebuffer ist so konfiguriert, dass er eine Sekunde buffert. Pro Knoten existiert nur ein Abonnent. Das Intervall des Benachrichtgungsbuffers, welcher festlegt, in welchen Intervallen Abonnenten benachrichtigt werden, liegt bei 500ms.

3. Wie Fall (2), wobei der Sendebuffer auf eine Bufferzeit von zwei Sekunden konfiguriert ist.

4. Wie Fall (2), wobei der Sendebuffer auf eine Bufferzeit von vier Sekunden konfiguriert ist

In Fall (1) kann von einer leichten Erhöhung der maximalen Differenz bei Erhöhung der Anzahl von Abonnenten ausgegangen werden. Für Fall (2,3,4) wird erwartet, dass die maximale Differenz bei der konfigurierten Sendebuffer Zeit + Benachrichtigungsbuffer Zeit liegt. Für die drei Fälle (2-4) entsteht eine erwartete maximale Differenz von:

Fall 2: $1s + 500ms = 1.5s$

Fall 3: $2s + 500ms = 2.5s$

Fall 4: $4s + 500ms = 4.5s$

Der Median sollte ungefähr bei der Hälfte der maximalen Differenzen liegen.

6.7.2 Ergebnisse & Diskussion

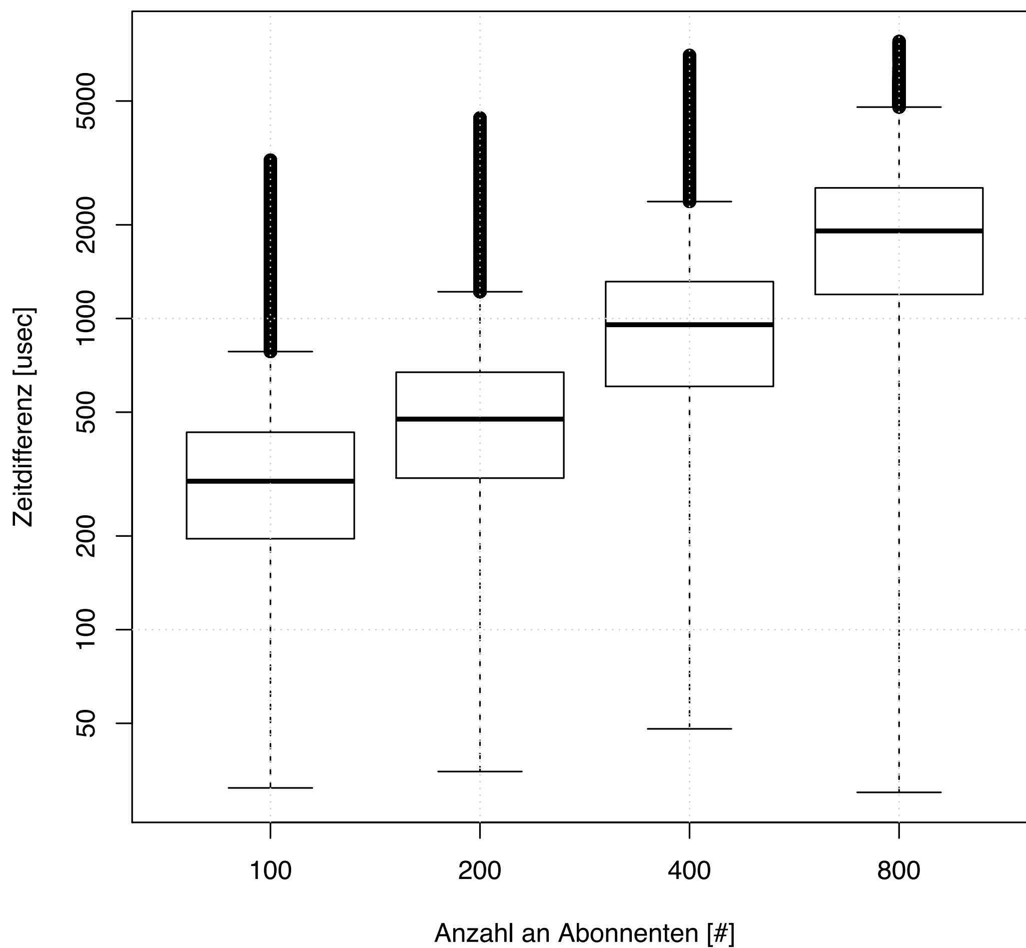

Abbildung 6.11: Verteilung der Zeitdifferenz bei unterschiedlicher Anzahl an Abonnenten auf nur einen Knoten

Abbildung 6.11 zeigt die Ergebnisse des ersten Falls. Dabei wurde nur lokal in einer Anwendung gemessen, wie sich die Anzahl an Abonnenten auf die verschiedenen Auslieferungszeiten auswirkt. Zu sehen sind vier Boxplots, welche auf der X-Achse eine verschiedene Anzahl an Abonnenten zeigen. Dei Y-Achse beschreibt die Verteilung der einzelnen Messwerte. Die Y-Achse zeigt eine Auflösung von Mikrosekunden (usec) und ist logarithmisch skaliert. Für die einzelnen Durchläufe wurde die Anzahl an Abonnenten variiert, wobei bei 100 Abonnenten begonnen und pro Durchlauf jeweils verdoppelt wurde.
Es ist zu sehen, wie der Median sich bei Verdoppelung der Anzahl an Abonnenten verdoppelt. Bei 100 Abonnenten liegt der Median bei 250usec, bei 200 Abonnenten liegt der Median bei 500usec. Bei 400 Abonnenten liegt der Median bereits bei der erwarteten Verdopplung von 1000usec. Die letzte Stufe der Verdopplung auf 800 Abonnenten weist einen Median von 2000usec auf.

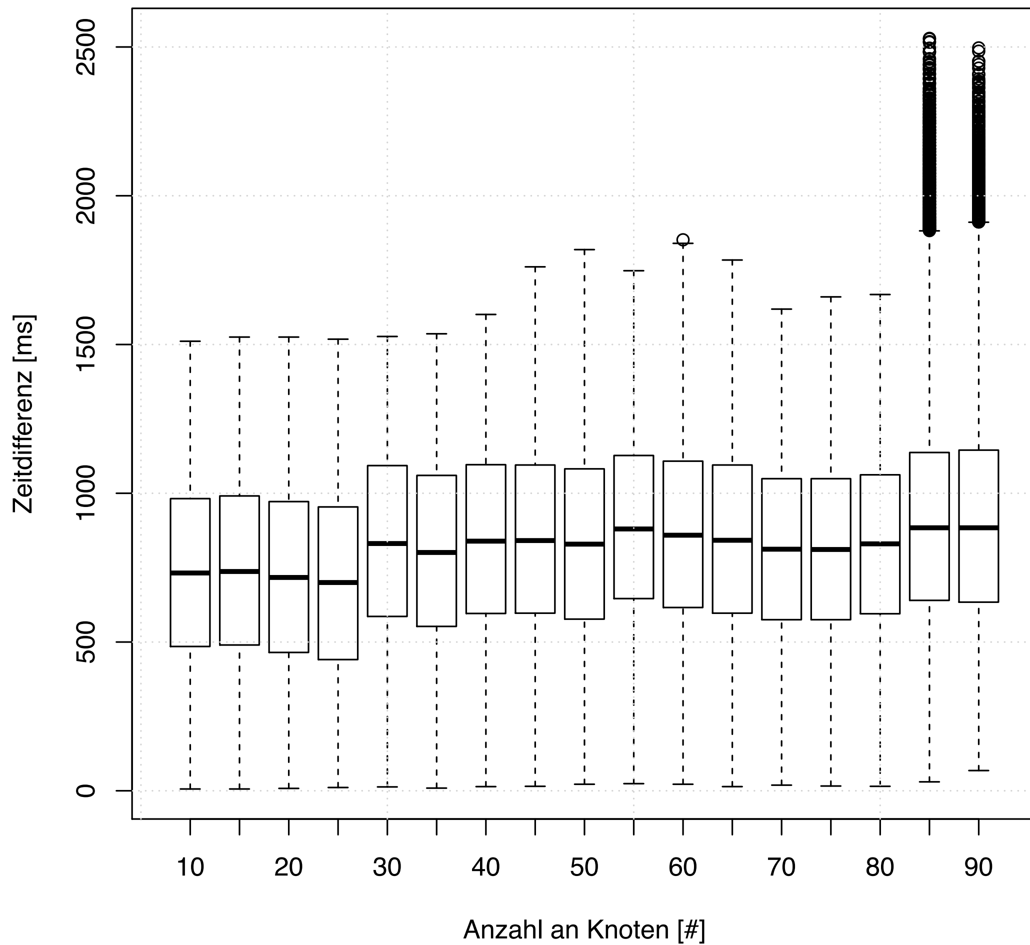

Abbildung 6.12: Verteilung Zeitdifferenz bei unterschiedlicher Anzahl an Knoten bei einem Flush-Intervall von einer Sekunde.

Die Ergebnisse des ersten mit Hilfe von Mininet verteilten Falls (Fall 2) sind in Abbildung 6.12 gezeigt. Zu sehen sind mehrere Boxplots, wobei jeder Durchlauf seinen eigenen Boxplot besitzt. Auf der X-Achse sind die verschiedenen Anzahlen an verwendeten Knoten zu sehen, wobei bei zehn Knoten begonnen wurde zu simulieren. Die Anzahl an Knoten wurde in Fünferschritten erhöht, bis 90 Knoten erreicht wurden. Die Y-Achse zeigt eine Auflösung von Millisekunden (ms) und ist nicht logarithmisch skaliert.

Es ist zu sehen, wie die maximal erwartete Differenz von $1.5s$ in den meisten Fällen nicht überschritten wird.

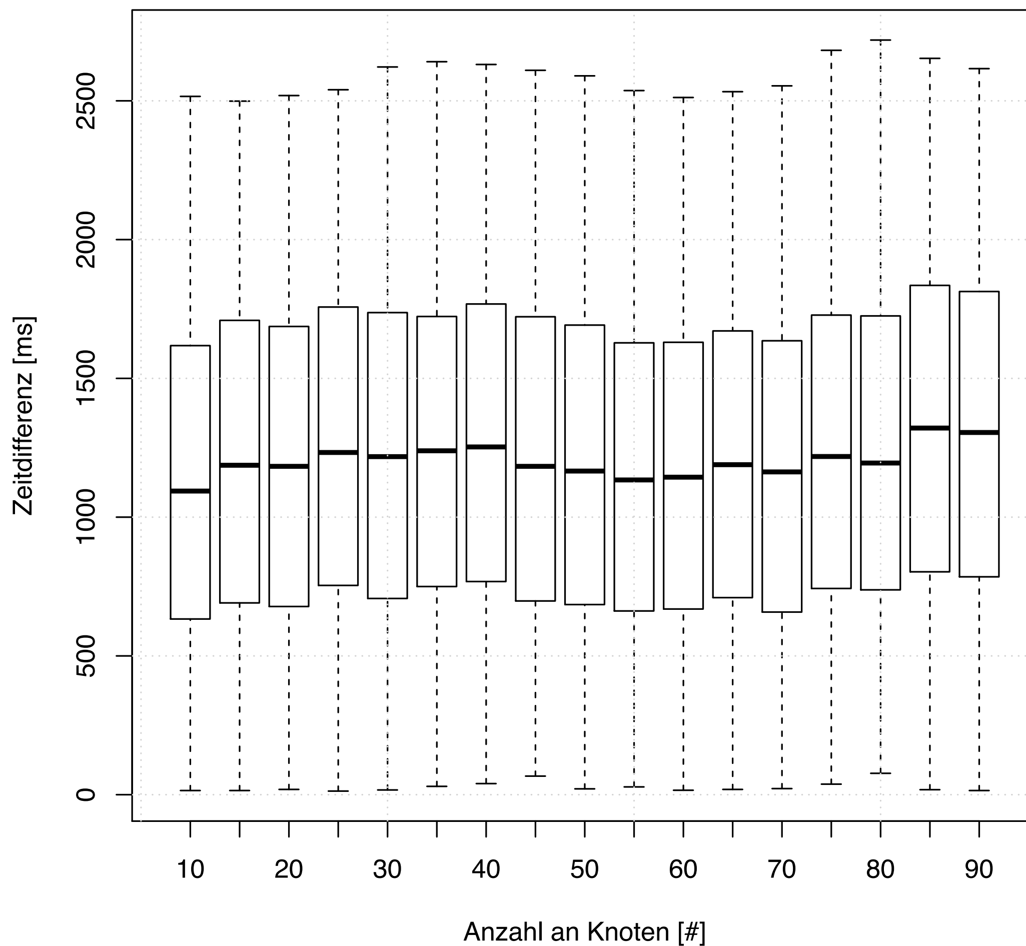

Abbildung 6.13: Verteilung Zeitdifferenz bei unterschiedlicher Anzahl an Knoten bei einem Flush-Intervall von zwei Sekunden

Abbildung 6.13 zeigt die Ergebnisse für den Fall 3. Zu sehen sind wieder mehrere Boxplots, wobei die X-Achse wieder von 10 bis 90 Knoten in Fünferschritten steigt. Die Y-Achse zeigt eine Auflösung in Millisekunden und ist nicht logarithmisch skaliert.
Es ist zu sehen, wie die maximal erwartete Differenz von $2.5s$ eingehalten wird, lediglich kleine Abweichungen im Bereich von etwa $250ms$ sind zu sehen.

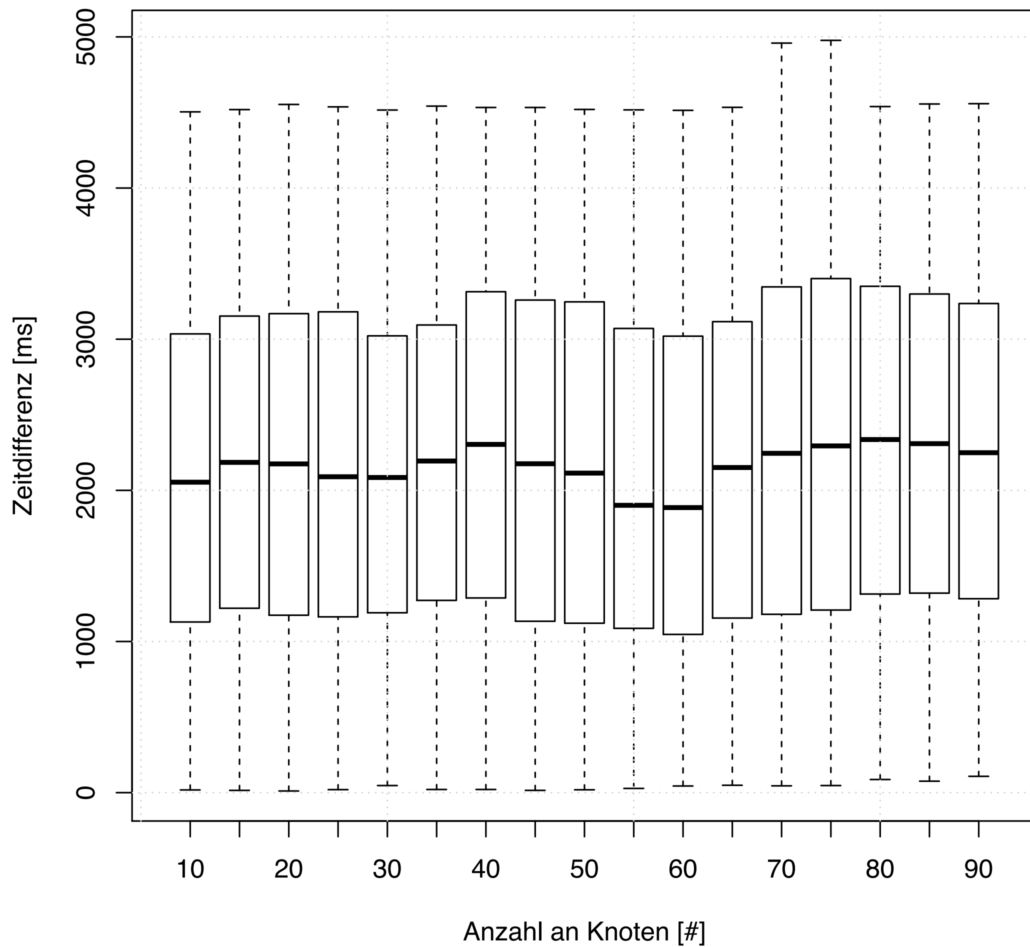

Abbildung 6.14: Verteilung Zeitdifferenz bei unterschiedlicher Anzahl an Knoten bei einem Flush-Intervall von vier Sekunden

Abbildung 6.14 zeigt die Ergebnisse des letzten verteilten Falls (Fall 4). Dabei sind wieder mehrere Boxplots auf der X-Achse zu sehen, wobei die Skalierungsschritte dieselben wie in den vorherigen Fällen ist. Die Y-Achse zeigt die Differenz in Millisekunden (ms) und ist nicht logarithmisch skaliert.
Es ist zu sehen, wie die maximale erwartete Differenz von $4.5s$ bei jeder bis auf zwei Durchläufen eingehalten wird. Die Durchläufe, welche die Differenz um 500ms übersteigen, sind bei 70 und 75 Knoten.

7 Schlussbetrachtung

7.1 Fazit

Das Ziel dieser Bachelor-Arbeit war die Integration von „Conflict-free Replicated Datatypes“ (CRDTs) in ein Aktor-Programmierkonzept. Außerdem sollten die Grundlagen für die Replikation von Daten und CRDTs beschrieben werden. Dabei wurde die Funktionsweise einfacher bekannter CRDTs erläutert. Ein CRDT-Modul für das C++ Actor Framework (CAF) wurde erstellt, welches die erläuterten CRDTs bereitstellt und sich außerdem um eigene CRDTs erweitern lässt. Dabei mussten unter anderem Anforderungen, die ein auf Aktoren basiertes, lose gekoppeltes verteiltes System stellt, im Design berücksichtigt werden. Hier wurde weiterhin abgewogen, wie Replikate Aktoren zugänglich gemacht und wie Replikate von der CAF-Laufzeit adressiert werden sollen. Außerdem wurden verschiedene Garantien verglichen, welche die Zuverlässigkeit, Konsistenz und Verfügbarkeit des Systems betreffen, wie zum Beispiel die Lebenszeit einzelner Replikate, oder mit welchen Zusicherungen Operationen auf mehreren Knoten ausgeführt werden. Es wurde diskutiert, welche Klasse von CRDTs (*CmRDTs*, *CvRDTs* oder δ-CRDTs) in CAF unterstützt werden soll. Schließlich wurde ein Algorithmus zum Verteilen von Replikaten beziehungsweise δ-CRDT-Zuständen vorgestellt.
In der Evaluation hat sich gezeigt, dass die Implementierung die zuvor geschätzte Konvergenzgeschwindigkeit einhalten kann. Die Verteilung von Replikaten sollte möglichst effizient arbeiten, weshalb das Modul sogenannte δ-CRDTs umsetzt. In der Evaluation hat sich gezeigt, dass δ-CRDTs im Vergleich zu *CvRDTs* weniger Daten im Netzwerk versenden. Es hat sich jedoch auch gezeigt, dass der verwendete Verteilungsalgorithmus durch das direkte Versenden an alle Knoten zu ineffizient arbeitet, womit eine der Anforderung an das Modul nicht erfüllt wurde. Außerdem skaliert das Versenden von notwendigen Meta-Informationen („welcher Knoten repliziert welche Replikate“) nicht effizient genug.

7.2 Ausblick

Ein möglicher Ansatzpunkt zur Optimierung ist die Umsetzung eines effizienteren Algorithmus zum Versenden von Meta-Informationen sowie Replikat-Daten. Dazu muss lediglich die Klasse

`distribution_layer` neu implementiert werden. Für die Verteilung von Meta-Informationen könnte ein Gossip-Verfahren [42, 43] zum Einsatz kommen. Es wäre auch denkbar, dass automatisch Netzwerk-Topologien erkannt und Daten anhand der Topologie effizient verteilt werden.

Ein weiterer möglicher Ansatz für eine Verbesserung ist die Implementierung einer Distributed-Hash-Table (DHT) [44, 45]. Diese kann dazu verwendet werden, um festzustellen, auf welchen Knoten Replikate gespeichert sind. Die Implementierung eines DHT würde den periodischen aktuell ineffizienten Austausch von Meta-Informationen unter Knoten ersetzen.

Literaturverzeichnis

[1] Lightbend Inc, "Akka java documentation, release 2.4.12." [Online]. Available: http://doc.akka.io/docs/akka/current/AkkaJava.pdf

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[4] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," Inria – Centre Paris-Rocquencourt ; INRIA, Research Report RR-7506, Jan. 2011. [Online]. Available: https://hal.inria.fr/inria-00555588

[5] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," *CoRR*, vol. abs/1410.2803, 2014. [Online]. Available: http://arxiv.org/abs/1410.2803

[6] G. A. Agha, "Actors: A model of concurrent computation in distributed systems." Ph.D. dissertation, MIT Computer Science and Artificial Intelligence Laboratory, 1985.

[7] "Erlang Programming Language," https://www.erlang.org/, accessed: 2017-04-27.

[8] J. Armstrong, "Making reliable distributed systems in the presence of software errors," Ph.D. dissertation, The Royal Institute of Technology Stockholm, Sweden, 2003.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[10] R. Klophaus, "Riak core: building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14.

[11] N. Hayashibara, X. Defago, R. Yared, and T. Katayama, "The φ accrual failure detector," in *Reliable Distributed Systems, 2004. Proceedings of the 23rd IEEE International Symposium on.* IEEE, 2004, pp. 66–78.

[12] "Protocol Buffers - Google Developers," https://developers.google.com/protocol-buffers/, accessed: 2017-04-27.

[13] P. Deutsch, "GZIP file format specification version 4.3," Network Working Group, Tech. Rep., 1996.

[14] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch, "Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments," in *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!* New York, NY, USA: ACM, Oct. 2013, pp. 87–96.

[15] I.B.M. Corporation, "IBM System/370 Extended Architecture, Principles of Operation," IBM, Tech. Rep. SA22-7085, 1983.

[16] S. Gilbert and N. Lynch, "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/564585.564601

[17] Tanenbaum, Andrew S and van Steen, Maarten, *Verteilte Systeme: Grundlagen und Paradigmen.* Pearson Studium, 2003, vol. 1.

[18] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[19] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide.* O'Reilly Media, Inc., 2010.

[20] A. MySQL, "Mysql: the world's most popular open source database," 2005.

[21] M. Stonebraker and L. A. Rowe, *The design of Postgres.* ACM, 1986, vol. 15, no. 2.

[22] M. N. Vora, "Hadoop-hbase for large-scale data," in *Computer science and network technology (ICCSNT), 2011 international conference on*, vol. 1. IEEE, 2011, pp. 601–605.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365815.1365816

[24] K. Chodorow, *MongoDB: The Definitive Guide.* O'Reilly Media, Inc., 2013.

[25] J. L. Carlson, *Redis in Action.* Greenwich, CT, USA: Manning Publications Co., 2013.

[26] F. J. Torres-Rojas and M. Ahamad, "Plausible clocks: constant size logical clocks for distributed systems," *Distributed Computing*, vol. 12, no. 4, pp. 179–195, 1999.

[27] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 173–182, 1996.

[28] M. Rabinovich and E. D. Lazowska, "An efficient and highly available read-one write-all protocol for replicated data management," in *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on.* IEEE, 1993, pp. 56–65.

[29] S. Goel and R. Buyya, "Data replication strategies in wide area distributed systems," *Enterprise service computing: from concept to deployment*, vol. 17, 2006.

[30] L. Torvalds and J. Hamano, "Git: Fast version control system," *URL http://git-scm. com*, 2010.

[31] H. Yu and A. Vahdat, "Design and evaluation of a continuous consistency model for replicated services," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4.* USENIX Association, 2000, p. 21.

[32] N. Ramsey, E. Csirmaz *et al.*, "An algebraic approach to file synchronization," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5. ACM, 2001, pp. 175–185.

[33] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1435417.1435432

[34] M. Letia, N. Preguiça, and M. Shapiro, "Crdts: Consistency without concurrency control," *arXiv preprint arXiv:0907.0929*, 2009.

[35] P. Hartmann, *Mathematik für Informatiker: Ein praxisbezogenes Lehrbuch.* Springer-Verlag, 2012.

[36] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed computing*, vol. 7, no. 3, pp. 149–174, 1994.

[37] P. R. Johnson and R. Thomas, "Maintenance of duplicate databases," 1975.

[38] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 6, no. 1, pp. 51–81, 1988.

[39] P. L. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek, "Resolving file conflicts in the ficus file system." in *USENIX Summer*, 1994, pp. 183–195.

[40] T. Berners-Lee, R. Fielding, and L. Masinter, "Rfc 3986: Uniform resource identifier (uri): Generic syntax," The Internet Society, Tech. Rep., 2005.

[41] "Mininet: An Instant Virtual Network on your Laptop (or other PC)," http://mininet.org/, accessed: 2017-05-02.

[42] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/41840.41841

[43] K. Birman, "The promise, and limitations, of gossip protocols," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, pp. 8–13, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1317379.1317382

[44] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse, *Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 160–169. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45172-3_15

[45] M. F. Kaashoek and D. R. Karger, *Koorde: A Simple Degree-Optimal Distributed Hash Table.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 98–107. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45172-3_9

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 15. Mai 2017 Marian Triebe