Snapshots in Scalaris

Ein Algorithmus zum Speichern konsistenter Zustände in verteilten Hashtabellen

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker(in)

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: Stefan Keidel
geboren am: 28. Januar 1985
in: Beeskow

Gutachter: Prof. Dr. Alexander Reinefeld
Prof. Dr. Miroslaw Malek

eingereicht am: ..... verteidigt am: .....

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Weiterhin erkläre ich, eine Diplomarbeit in diesem Studiengebiet erstmalig einzureichen.

Berlin, den 3. August 2012 .....

Danksagung

Mein Dank gilt den beiden Gutachtern dieser Arbeit, Prof. Dr. Alexander Reinefeld, Konrad-Zuse-Zentrum für Informationstechnik Berlin, Leiter des Lehrstuhls Parallele und Verteilte Systeme an der Humboldt-Universität zu Berlin und Prof. Dr. Miroslaw Malek, Leiter des Lehrstuhls Rechnerorganisation und Kommunikation an der Humboldt-Universität zu Berlin. Weiterhin danke ich den 3 Hauptentwicklern von *Scalaris* Nico Kruber, Thorsten Schütt und vor allem dem Betreuer dieser Arbeit, Dr. Florian Schintke für zahlreiche anregende Diskussionen und Hilfestellungen. Zu guter Letzt soll ein spezieller Dank an alle Freunde und Familienmitglieder gerichtet werden, die mich auf meinem Weg unterstützt haben.

Zusammenfassung

Eines der größten Hindernisse beim praktischen Einsatz von *Scalaris*, einer skalierbaren Implementierung einer verteilten Hashtabelle mit Unterstützung für Transaktionen, ist das Fehlen eines Verfahrens zur Aufnahme eines konsistenten Zustandes des gesamten Systems. Wir stellen in dieser Arbeit ein einfaches Protokoll vor, dass diese Aufgabe erfüllt und sich, auf Grund der von uns gewählten Herangehensweise, leicht implementieren lässt.

Als Ausgangspunkt dafür wählen wir aus einer Reihe von „klassischen“ Snapshot-Algorithmen ein 1993 von Mattern entworfenes Verfahren, welches auf dem Algorithmus von Lai und Yang basiert, aus. Diese Entscheidung basiert auf einer gründlichen Analyse der Protokolle unter Berücksichtigung der Architektur der existierenden Software.

Im nächsten Arbeitsschritt benutzen wir unser vollständiges Wissen über die Interna des Transaktionssystems von *Scalaris* und vereinfachen damit das Verfahren hinsichtlich Benutzbarkeit und Implementierungskomplexität, ohne die Anforderungen an den aufgenommenen Zustand aufzuweichen. Statt einer losen Anhäufung lokaler Zustände der einzelnen Teilnehmerknoten können wir am Ende eine große Schlüssel-Wert-Tabelle als Ergebnis erzeugen, die konsistent ist, sich leicht weiterverarbeiten lässt und die einem Zustand entspricht, in dem sich das System einmal befunden haben könnte.

Nachdem wir das Verfahren dann in Software umgesetzt haben, werten wir die Ergebnisse hinsichtlich des Einflusses auf die Performanz des Gesamtsystems aus und diskutieren mögliche Weiterentwicklungen.

Inhaltsverzeichnis

1 Einleitung **1**

2 Grundlagen **3**
2.1 Verteilte Hashtabellen und strukturierte Overlay-Netze 3
2.1.1 Chord und Chord# 4
2.1.2 Scalaris 7
2.2 Ereignisse und konsistente Schnitte 15
2.2.1 Ereignismodelle 15
2.2.2 Konsistente Schnitte 16

3 Snapshot-Algorithmen **19**
3.1 Der Algorithmus von Chandy und Lamport 19
3.1.1 Modell 19
3.1.2 Der Algorithmus 20
3.1.3 Eigenschaften des lokalen Zustandes 21
3.1.4 Variationen des Chandy-Lamport Algorithmus 22
3.2 Der Algorithmus von Lai und Yang 28
3.3 Die Algorithmen von Mattern 30
3.3.1 Adaption von Lai und Yang 30
3.3.2 Adaption von Chandy und Lamport 32
3.4 Hemmung und die Algorithmen von Taylor 33
3.4.1 Protokolle 35
3.4.2 Zusammenfassung 38

4 Analyse der Algorithmen **40**
4.1 Bewertungskriterien 40
4.2 Bewertung der Algorithmen 42
4.2.1 Systemmodell 43
4.2.2 Konsistenzmodell 44
4.2.3 Topologie 45
4.2.4 Kommunikationskomplexität 47
4.2.5 Hemmung von Aktionen 48
4.2.6 Minimal-invasive Implementierung 48
4.3 Betrachtung der Algorithmen im Kontext von Scalaris 49
4.3.1 Systemmodell 50

4.3.2 Konsistenzmodell 50
4.3.3 Topologie 51
4.3.4 Kommunikationskomplexität und Hemmung von Aktionen . . 51
4.3.5 Minimal-invasive Implementierung 52
4.4 Destillation einer Implementierungsempfehlung 52
4.4.1 Wahl des Verfahrens 54

5 Anpassung des Algorithmus 56
5.1 Grundlegendes Design 56
5.2 Der Algorithmus 60
5.3 Zusammenfassung des Algorithmus 63
5.4 Ausgewählte Implementierungsdetails 65

6 Auswertung 68
6.1 Benutzung und Beispielabläufe 68
6.2 Performanz des Gesamtsystems 70
6.3 Abbrüche von Transaktionen 71
6.4 Privatsphäre und Sicherheit 73
6.5 Mögliche Weiterentwicklungen 73

7 Rückblick 75

1 Einleitung

Immer mehr IT-Ressourcen und Daten werden heutzutage in die „Cloud“ verlagert. Dieser Prozess des „Cloudsourcing“ geht einerseits mit der Einführung vieler neuer Anwendungen, andererseits mit der Umstellung älterer Applikationen auf neue Technologien einher. Ein Sektor, der sich bedingt durch diesen Prozess seit einiger Zeit auf dem aufstrebenden Ast befindet, ist die Gruppe der *NoSQL*-Datenbanken. Hier wird der klassischen Lehre der relationalen Datenbanken fast vollständig abgesagt und stattdessen auf einfachere Ansätze zurückgegriffen.

Eine wichtige Art von *NoSQL*-Datenbanken sind die sogenannten *Distributed Hash Tables* (DHT), also auf verteilten Systemen umgesetzte Tabellen, deren Inhalt nur aus Schlüssel-Wert-Paaren besteht. In diesem Bereich ist das Software-Projekt *Scalaris* anzusiedeln. Hier wurden bekannte Verfahren wie *Chord* und *Paxos Commit* mit eleganten Erweiterungen versehen und zusammengefügt, um eine skalierbare DHT mit Unterstützung für Transaktionen zu implementieren.

In einem solchen System sind die eigentlichen Daten jedoch in gewisser Weise schwer zugänglich. Sie werden nach einem bestimmten Schema unter den Knoten des verteilten Systems aufgeteilt und repliziert, so dass ihre Verfügbarkeit jederzeit sichergestellt ist. Bisher ist es jedoch nicht möglich, eine klassische Sicherungskopie, bzw. einen konsistenten Zustand des Gesamtsystems und dessen Datenbestandes herzustellen.

In der Praxis ist dies ein großes Hindernis für potentielle Nutzer, die ihre Daten gegebenenfalls zu einem früheren Zustand zurücksetzen wollen, zum Beispiel um Eingabefehler zu berichtigen, oder die möglicherweise ihr gesamtes verteiltes System zum Einspielen von Software-Updates kontrolliert herunterfahren und später neu starten wollen. Die Garantien, die Scalaris bezüglich Verfügbarkeit und Konsistenz der Daten liefern kann, sind in der Praxis nicht ausreichend. Es wird ein Verfahren zur Aufnahme eines Zustands des Systems, eines Snapshots, benötigt. Dieses Defizit beabsichtigen wir, mit dieser Arbeit zu beseitigen.

Dazu wählen wir folgende Herangehensweise: In Kapitel 2 werden die notwendigen theoretischen Grundlagen zu der Architektur von Scalaris und zu konsistenten Schnitten in verteilten Systemen zusammengefasst. In Kapitel 3 widmen wir uns dann einem umfangreichen Überblick über existierende, auf allgemeine verteilte Sys-

teme anwendbare Snapshot-Algorithmen um diese Verfahren in Kapitel 4 bezüglich der Anwendbarkeit für unser Szenario zu analysieren und einen Favoriten bzw. eine Empfehlung zu destillieren. In Kapitel 5 passen wir dieses Protokoll so an, dass es sich gut in das existierende Software-Projekt einfügt und die Komplexität der Implementierung möglichst gering bleibt. Nachdem das Ganze dann in Software gegossen wurde, geht es in Kapitel 6 um die Auswertung unserer Arbeit und möglichen Verbesserungen und Erweiterungen.

2 Grundlagen

2.1 Verteilte Hashtabellen und strukturierte Overlay-Netze

Eine verteilte Hashtabelle ist im Grunde eine Menge von Schlüssel-Wert-Paaren die in einem verteilten System realisiert wurde. Jeder der Knoten des Systems stellt als Teil der Tabelle eine Funktion *lookup(key)* zur Verfügung, die den Wert in der Tabelle zu dem (eindeutigen) Schlüssel *key* zurückgibt. Dabei ist hervorzuheben, dass jeder Knoten zu jedem Schlüssel befragt werden kann, nicht nur zu denen, die zufälligerweise lokal gespeichert sind. Um diese Art Operationen bereitzustellen, hält jeder Knoten eine Menge von Zeigern auf benachbarte Knoten vor (*routing pointer*), über die nicht lokal beantwortbare Anfragen an andere weitergeleitet werden können. Diese Zeiger, die wenig mit der physischen Netzwerkstruktur zu tun haben, bilden gemeinsam mit den Knoten im System ein so genanntes strukturiertes Overlay-Netz (*structured overlay network SON*), also eine Struktur (zum Beispiel ein Ring oder ein Stern) welche über dem eigentlichen Netzwerk (im Sinne des OSI-Stack oberhalb der Transportschicht) liegt. Jeder Knoten wird dabei versuchen, die Liste seiner Nachbarn so zu verwalten, dass die Struktur des Netzes erhalten bleibt (zum Beispiel durch das Schließen einer Lücke in einem Ring).

Üblicherweise können Systeme, die nach den obigen Paradigmen aufgebaut wurden, ein hohes Maß an Skalierbarkeit garantieren, d.h. es kann eine große Anzahl von Knoten betrieben werden (und damit natürlich auch eine entsprechende Anzahl von Schlüssel-Wert-Paaren gespeichert werden). Vorraussetzung für diese Eigenschaft ist ein effizientes *Routing* von nicht lokal beantwortbaren Anfragen. Bei aktuellen Systemen wie Chord [32] oder Chord# [27] ist die Anzahl der Sprünge durch das Overlay-Netz (*hops*) logaritmisch von der Zahl der Knoten im Netz abhängig, so daß zum Beispiel in einem System mit einer Million Teilnehmern im Durchschnitt lediglich etwa 20 Sprünge zu Nachbarknoten benötigt werden, bis eine Anfrage den richtigen Adressaten erreicht hat. Die Anfragezeit wächst mit zunehmender Größe des Systems also nicht linear, wohl aber die Anzahl der Schlüssel-Wert-Paare, die gespeichert werden können (unter der Annahme, dass alle Knoten gleich viele Daten aufnehmen können). Bemerkt sei zusätzlich, dass auch Systeme existieren, bei denen sich die Basis b der Logarithmusfunktion ($\log_b n$), die die durchschnittliche Anzahl

der benötigten Sprünge angibt, zur Entwurfszeit festlegen lässt [1].

Die ursprüngliche Motivation bei der Entwicklung von DHT-Schemata war die Bereitstellung von effizientem *File Sharing* (Freigabe von Dateien) [32]. Traditionelle Systeme auf diesem Gebiet sahen entweder einen zentralen Index mit Metadaten der Dateien vor (ein prominentes Beispiel ist die einstige Funktionsweise der erfolgreichen Datei-Tauschbörse *Napster*) oder verfolgten einen komplett dezentralen Ansatz, bei dem jeder Knoten nur Indexinformationen über die lokale Datenmenge besitzt, so dass bei einer Anfrage möglichst viele Knoten kontaktiert werden müssen (Beispiel *Gnutella*). Heute werden DHTs allerdings für verschiedene Arten von Diensten benutzt, auf den effizienter Zugriff benötigt wird. Bei Anwendungen wie dem *File Sharing* wird implizit immer davon angenommen, dass es sich um ein Netz von vielen unzuverlässigen und heterogenen Knoten handelt (ein klassisches Peer-To-Peer-Netz), in dem jeder Nutzer zu jeder Zeit seinen Computer ausschalten kann (Knoten fällt aus). Es kommt also zu einer extremen Dynamik durch ausfallende und neu hinzukommende Knoten und einer sich entsprechend häufig ändernden Struktur des Overlay-Netzes (*Churn* [1]). Weiterhin sind die Knoten im Allgemeinen nicht vertrauenswürdig, d.h. auch potentielle Angreifer können sich dem Overlay-Netz anschließen.

2.1.1 Chord und Chord#

Nachdem bisher ein Überblick über die allgemeinen Eigenschaften und über die Bedeutung von DHTs und SONs gegeben wurde, werden wir nun am Beispiel vom 2001 am Massachusetts Institute of Technology entwickelten Schema *Chord* [32] darstellen, wie sich eine solche Struktur konstruieren lässt und später aufzeigen, welche Schwächen dieses Aufbaus sich mit relativ einfachen Maßnahmen beheben lassen, was uns zur Modifikation Chord# [27] führt.

Chord

Die zu Grunde liegende Struktur bei Chord ist ein Ring. Es wird ein logischer Namensraum (*Identifier Space*) benutzt, der aus den Bezeichnern $\{0, 1, 2, \ldots, n-1\}$ besteht. Dieser Namensraum ist dann ein logischer Ring modulo n. Jeder Knoten im System sucht sich mit Hilfe einer Hashfunktion h_2 einen „zufälligen“ Bezeichner aus dem *identifier space* aus. Die Funktion sollte so gewählt werden, dass der Namensraum gleichmäßig an Knoten verteilt wird. Auf Basis dieser Zuweisung kann

[1] *Churn* ist die Kombination aus den englischen Verben *change* und *turn* und bezeichnet im Kontext von Peer-To-Peer-Netzwerken Umgebungen, die sich mit hoher Frequenz ändern (Knoten verlassen das Netz, neue Knoten kommen hinzu, Netzwerkfehler treten auf, etc.)

nun der folgender Begriff definiert werden: Der *Nachfolger* eines Knotens oder eines Bezeichners ist der nächste Knoten den man bei einer Bewegung im Uhrzeigersinn durch den Namensraum trifft. Analog lässt sich der *Vorgänger* mit einer gegen den Uhrzeigersinn gerichteten Bewegung entlang des Ringes beschreiben.

Zur Bestimmung des Speicherortes der Daten benutzen wir ebenfalls wieder eine global bekannte Hashfunktion (h_1). Mit deren Hilfe wird jedem Schlüssel-Wert-Paar, welches in der DHT abgelegt werden soll, ein Bezeichner zugewiesen ($b = h_1(key)$). Das Paar wird dann auf dem Knoten $k = \text{succ}(b)$ (dem Nachfolger dieses Bezeichners) gespeichert.

Um das Routing zu ermöglichen muss jeder Knoten eine gewisse Menge an Zeigern vorhalten, mit denen die SON-Struktur (in diesem Fall der Ring) aufrecht erhalten wird. Bei Chord zeigt jeder Knoten i auf seinen Nachfolger ($s = \text{succ}(i + 1)$) und seinen Vorgänger ($p = \text{pred}(i-1)$). Wenn ein Knoten dann eine Anfrage nach einem Schlüssel k bekommt, berechnet er den entsprechenden Wert der global bekannten Hashfunktion $v = h_1(k)$ und folgt dann den Nachfolger-Zeigern bis der Schlüssel gefunden wurde. Von nun an nennen wir den Prozess des Findens eines Nachfolgers eines Bezeichners *Lookup*. Nach dieser Methode kann es im schlechtesten Fall zu n Lookup-Operationen kommen, wenn das System aus n Knoten besteht. Dieser Zustand wird nun verbessert, indem eine Tabelle eingeführt wird, in der jeder Knoten weitere Zeiger auf andere Knoten als den direkten Vorgänger und den direkten Nachfolger vorhält. In dieser Tabelle zeigt der erste Eintrag auf den Knoten $\text{succ}(n+1)$, der Zweite auf $\text{succ}(n+2)$, der Dritte auf $\text{succ}(n+4)$ und der m-te auf $\text{succ}(n + 2^{m-1})$. Dadurch wird die Distanz zum Ziel jeweils halbiert. Die Größe der Tabelle ist m, wobei $n = 2^m$ (n ist die Größe des Namensraums). Jeder Knoten kennt also für $i = 1, \ldots, m$ die Knoten $\text{succ}(n + 2^{i-1})$. Dadurch werden durchschnittlich nur noch $\log_2(n)$ Sprünge von einem Knoten zu einem beliebigen anderen Knoten auf dem Ring benötigt.

Weiterhin muss sichergestellt werden, dass die Ringstrukur auch im Fehlerfall erhalten bleibt. Da das Routing komplett von den Nachfolger-Zeigern abhängig ist, ist es essentiell, dass diese zu jedem Zeitpunkt korrekt gesetzt sind. Wir erinnern uns, dass jeder Knoten auch einen Vorgänger-Zeiger vorhält. Dieser dient zur Erhaltung der Ringstruktur. Wir versuchen also den Nachfolger-Zeiger *succ* und den Vorgänger-Zeiger *pred* auf den jeweils nächsten jeweils „aktiven“ Knoten zu setzen. Diesen Vorgang nennen wir *periodic stabilization*. Dabei fragt jeder Knoten nach einem bestimmten Intervall den Vorgänger seines Nachfolgers ab und setzt, falls der erhaltene Wert nicht leer und im Intervall $(n, succ]$ ist, seinen Nachfolger-Zeiger auf den eben abgefragten Knoten. Zusätzlich sendet er in jedem Fall eine Nachricht an seinen Nachfolger, welcher in voriger Anweisung gegebenenfalls neu gesetzt wurde. Empfängt ein Knoten eine Nachricht, setzt dieser seinen Vorgänger-Zeiger auf den Knoten, von dem er die Nachricht erhielt (p), falls sein Vorgänger-Zeiger nicht gesetzt ist oder falls $p \in (pred, n]$. Ist diese Methode implementiert, können neue

Knoten dem Ring beitreten. Darüber hinaus wird eine Strategie zur Aktualisierung der Routing-Tabellen benötigt (*finger stabilization*). Diese sieht vor, die Einträge der Tabelle periodisch zu aktualisieren (mit Hilfe von Lookup-Operationen für alle m).

Der nunmehr letzte Punkt ist die Handhabung von Fehlern. Ist einmal ein *succ*-Zeiger falsch gesetzt, kollabiert der gesamte Ring. Aus diesem Grund wird nun für jeden Knoten eine Nachfolger-Liste der Größe r eingeführt, welche die r direkten Nachfolger enthält, wobei $r = log(N)$ oder $r = const$. Weiterhin wird die *periodic stabilization* wie folgt erweitert: Wenn der Nachfolger eines Knotens ausfällt, ersetzt er seinen entsprechenden Zeiger mit dem nächsten „lebenden“ Nachfolger (wahrscheinlich der zweite Eintrag in der angesprochenen Nachfolger-Liste). Wenn der direkte Vorgänger ausfallen sollte, löscht er seinen Vorgänger-Zeiger. Dazu wird allerdings ein Fehlerdetektor benötigt. Alle weiteren notwendigen Vorgänge werden von der *periodic stabilization* ausgeführt. Sind all diese Strategien implementiert, können Knoten dem Ring hinzugefügt werden und Knoten den Ring verlassen, ohne die Struktur des Ringes nachhaltig zu beeinträchtigen.

Chord#

Das primäre Ziel bei der Einführung von Chord# war die Eliminierung der beiden Hashfunktionen, h_1 zur Abbildung von Schlüsseln in den Namensraum und h_2 zur Abbildung von Knoten in den Namensraum. Wir entfernen zuerst die Hashfunktion h_1 und benutzen statt Funktionswerten der Schlüssel die Schlüssel-Zeichenketten selbst, um Daten in der DHT abzulegen (die Hashfunktion ist also die Identitätsfunktion). Die Daten liegen dann in lexikografischer Ordnung vor, das Routing von Chord funktioniert weiterhin. Wir können allerdings nicht mehr sicherstellen, dass die Knoten durch die Wahl einer guten Funktion gleichmäßig verteilt sind. Es kann also - je nach Datenmenge - Knoten geben, die überladen sind. Dies wird nun bei der Eliminierung von h_2 abgefangen. Statt die Knoten mit einer Hashfunktion im Namensraum zu verteilen, geben wir nun als vages Kriterium an, dass die Knoten so gesetzt werden sollen, dass die Schlüssel gleichmäßig über alle Knoten verteilt sind. Die lexikografische Ordnung der Schlüssel bleibt dabei weiterhin erhalten, es gehen jedoch die Routing-Garantien von Chord verloren, da der Aufbau der Routing-Tabellen weiterhin im Namensraum (*key space*) geschieht. Dort sind die Knoten allerdings nicht mehr gleichmäßig verteilt. Die Konsequenz ist, dass das Routing im *node space*, also direkt auf der Knotenmenge geschehen muss. Entsprechend brauchen wir eine neue Regel für die Berechnung des i-ten Eintrages der Routing-Tabelle.

Dieser ergibt sicht nun wie folgt:

$$p_i = \begin{cases} succ \text{ , wenn } i = 0 \\ p_{i-1}.p_{i-1} \text{ , wenn } i \neq 0 \end{cases}$$

Die Notation $p_{i-1}.p_{i-1}$ referenziert den $i-1$-ten Tabelleneintrag des Knotens, auf den der eigene $i-1$-te Tabelleneintrag zeigt. Dadurch ergeben sich eine Reihe von Vorteilen gegenüber dem klassischen Chord:

- Das System funktioniert ohne Anpassungen mit jeder Art von Schlüsselmenge, vorausgesetzt auf dieser Menge existiert eine natürliche Ordnung.
- Die Wartung der Einträge in der Routing-Tabelle ist weniger aufwändig [25]: Es wird lediglich ein Sprung benötigt.
- Es kann dynamisch auf Lastverschiebungen reagiert werden.
- Es können theoretisch Anfragen nach Schlüsselbereichen (range queries) unterstützt werden, da bekannt ist, welche Schlüssel sich wo befinden.
- Während Chord nur im Durchschnitt logarithmisches Routing liefert, kann Chord# $\log_2 n$ als obere Schranke garantieren.

Die Kenntnis über die Schlüsselposition kann allerdings auch dazu führen, dass potentiellen Angreifern die Arbeit erleichtert wird.

2.1.2 Scalaris

Da sich die Betrachtungen in Kapitel 3 und 4 hauptsächlich auf die Anwendbarkeit verschiedener Algorithmen auf das existierende Software-Projekt *Scalaris* [22, 24, 28] beziehen werden, wird nun eine Einführung in die grundlegenden Konzepte der Anwendung folgen. Bei Scalaris handelt es sich um eine in der Programmiersprache Erlang [2] implementierte DHT mit Unterstützung für Transaktionen und Replikation. Die Architektur sieht eine Unterteilung des Systems in drei grundlegende Schichten vor, die jeweils eine oder mehrere wichtige Eigenschaften für das Gesamtsystem bereitstellen.

- Auf der untersten Ebene, der Peer-To-Peer-Schicht, befindet sich ein SON, welches die Basisfunktionalität zum skalierbaren Speichern und Abfragen von Schlüsseln und den zugehörigen Werten bereitstellt.
- In der mittleren Schicht wird die Replikation der Daten implementiert und

somit deren Verfügbarkeit sichergestellt.

- Die oberste Schicht steuert die Unterstützung für Transaktionen[2] bei. Damit wird Atomizität[3], strenge Konsistenz[4] und Isolation[5] für verschiedene Mengen von Anfragen gewährleistet. Es werden eine optimistische Nebenläufigkeitskontrolle und ein nicht-blockierendes Commit-Protokoll benutzt.

Peer-To-Peer-Schicht

Auf der Peer-To-Peer-Ebene wird das in Abschnitt 2.1.1 eingeführte Verfahren Chord# verwendet, um Schlüssel-Wert-Paare zu speichern und die Daten wieder auszulesen. Dadurch erreicht Scalaris die erwünschten Eigenschaften: Skalierbarkeit mittels $\log(N)$-Routing, Gleichverteilung der Schlüssel auf die Knoten und die Sicherheit, dass das Overlay-Netz seine strukturelle Integrität selbst verwaltet. Ein in Abschnitt 2.1.1 schon kurz angesprochenes Problem von Chord und Chord# ist die Notwendigkeit eines Fehlerdetektors. Dieser, in der Theorie oftmals als existent angenommene, essentielle Teil des Algorithmus musste hier nun praktisch implementiert werden. Im realen Einsatz kann es dazu kommen, dass der Fehlerdetektor falsch liegt. So kann ein Knoten (bzw. dessen Verbindung) temporär so langsam sein, dass der Fehlerdetektor ihn zu Unrecht als ausgefallen einstuft. Zudem kann es, auch ohne außergewöhnliche Belastungen des Overlays durch *Churn* zu verschiedenen Inkonsistenzen kommen. Der erste Typ ist die *responsibility inconsistency*: In bestimmten Fällen kann es passieren, dass zwei Knoten annehmen für ein Schlüssel verantwortlich zu sein. Der zweite Typ ist die *lookup inconsistency*: Es existiert die Möglichkeit, dass Lookups nach einem Schlüssel auf zwei verschiedenen Knoten zwei verschiedene Ergebnisse liefern.

Replikationsschicht

Auf Replikationsebene wird bei Scalaris ein symmetrisches Schema benutzt, d.h. es existiert eine global bekannte Funktion f, welche die Menge der Schlüssel bestimmt, unter denen die Daten in der DHT abgelegt werden. Es muss also Maßnahmen zur Sicherstellung der Datenkonsistenz geben, da einzelne Knoten falsche Informationen

[2] Eine Transaktion ist eine logisch zusammengehörige Folge von Lese- und Schreiboperationen auf einem Datenbestand

[3] Das Prinzip Atomizität sieht vor, dass die Transaktion entweder vollständig oder gar nicht ausgeführt wird.

[4] Ein verteilter Datenspeicher ist streng konsistent, wenn alle Schreiboperationen für alle Prozesse unmittelbar sichtbar sind.

[5] Das Prinzip der Isolation sieht vor, dass sich nebenläufig ablaufende Transaktionen nicht gegenseitig beeinflussen.

liefern könnten. Im Falle von Scalaris wird dies mit Hilfe von Mehrheitsentscheidungen beim Lesen und Schreiben gewährleistet. Auch hier kann die *lookup inconsistency* Probleme verursachen. In einem solchen Fall kann es mehr als m Replikate (m sei die gewünschte Anzahl) und damit zwei oder mehr nicht-überlappende Mehrheiten geben. Dieses Problem würde durch die Benutzung stärkerer Mehrheitsentscheidungen als der einfachen Mehrheit (z.B. zwei Drittel) etwas entschärft, jedoch nicht gelöst. Es fehlt derzeit ein Verfahren zur Wartung und Reparatur von Replikaten. Nichtsdestotrotz wird durch die Benutzung von Replikation und Mehrheitsentscheidungen ein deutliches Plus an Konsistenz und Verfügbarkeit der Daten gewonnen.

Transaktionsschicht

Als Transaktionsprotokoll wird bei Scalaris eine Abwandlung von *Paxos Commit* [10] (welches auf *Paxos Consensus* [16, 6] basiert) benutzt. Für eine detaillierte Beschreibung der beiden Algorithmen verweisen wir auf [23]. Wir beschränken uns hier auf die Einbettung von *Paxos Commit* in die Architektur des Systems, basierend auf den Beschreibungen in [22, 23].

Scalaris unterstützt also Transaktionen, d.h. dass ein mit dem System verbundener Client eine Sequenz von Lese- und Schreiboperationen in einem Transaktionskontext (zwischen *begin of transaction - BOT* und *end of transaction - EOT*), insbesondere isoliert und atomar ausführen kann. Auf dem DHT-Knoten, mit dem der Client verbunden ist, wird ein lokaler Transaktionsmanager (TM) gestartet, welcher die Ausführung der Sequenz steuert.

Transaktionen in Scalaris werden nach einem optimistischen Schema ausgeführt. Jede Transaktion wird auf dem Client in einer Arbeitsphase komplett lokal ausgeführt. Der Client erstellt dabei ein so genanntes Transaktionslog, in dem die auszuführenden Operationen verzeichnet sind. Jeder Eintrag ist dabei ein 5-Tupel mit den folgenden Einträgen: (1) die Operation, (2) der Schlüssel, (3) ein Statusbit, welches den Erfolg der Operation anzeigt, (4) der zugehörige Wert und (5) eine Versionsnummer (s.u.). Eine Leseoperation führt zu einem *quorum read* (eine Leseoperation auf allen Replikaten mit Mehrheitsentscheidung) auf dem Ring, wenn der Schlüssel noch nicht im Log enthalten ist. Eine Schreiboperation führt ebenfalls zu einem *quorum read*, wenn der Schlüssel noch nicht enthalten ist. Daraufhin wird ein neuer Eintrag mit erhöhter Versionsnummer und dem neuen Wert erstellt (bzw. aktualisiert). Wenn eine *quorum read* Operation fehlschlägt, wird dies mit Hilfe des Statusbits im Log vermerkt. Wenn eines der Statusbits im Log einen Fehler anzeigt, wird die Transaktion abgebrochen. Der Client hat in der Arbeitsphase die Möglichkeit, beliebigen Programmcode auszuführen, um zum Beispiel Werte im Datenbestand auf bestimmte Bedingungen zu prüfen und die Transaktion entsprechend zu modifizieren.

Wenn diese Phase erfolgreich verläuft, versucht der TM die Sequenz permanent werden zu lassen (*commit*) und auf den jeweils verantwortlichen DHT-Knoten zu speichern. Die Nebenläufigkeitskontrolle ist Teil dieser zweiten Phase. Eine Transaktion t bricht nur dann ab, wenn:

(a) andere Transaktionen die Mehrheit an Sperren auf überlappenden Datensätzen halten (simultane Validation), oder wenn

(b) andere erfolgreich abgeschlossene Transaktionen Datensätze, auf die in der zu überprüfenden Transaktion t zugegriffen wird, verändert haben (Versionskonflikt).

Jeder Datensatz erhält, wie bei der Beschreibung der Arbeitsphase bereits erwähnt, eine Versionsnummer. Lese- bzw. Schreiboperationen holen eine Mehrheit der Replikate, um die höchste Versionsnummer (und damit die neueste Version) zu erhalten. Leseoperationen wählen den Wert mit der höchsten Versionsnummer, während Schreiboperationen die höchste Versionsnummer inkrementieren.

In der Validationsphase wird eine adaptierte Version des *Paxos Commit* Protokolls [10, 23] verwendet. Entscheidend hierbei ist, dass das Protokoll nicht-blockierend ist und (im Gegensatz zu einem 3-Phasen-Commit-Protokoll, welcher in verteilten Datenbanken eingesetzt wird) im Falle einer Partitionierung des Netzwerkes durch einen Fehler zumindest in der größeren Partition weiterhin funktioniert (so lange entsprechende Mehrheiten erreicht werden können). Dem einzelnen Transaktionsmanager TM werden wird hierbei eine Gruppe von replizierten Transaktionsmanagern (RTM) zur Seite gestellt. Zusammen bilden der TM und die RTMs (im Sinne des *Paxos Consensus* Algorithmus [23]) eine Menge von Akzeptoren, in der der TM als Anführer auftritt.

Im fehlerfreien Fall führt Scalaris in der Validationsphase einer Transaktion die folgenden vier Schritte aus (vgl. Abbildung 2.1). Sei r der im System verwendete Replikationsfaktor. Dann hält jeder Knoten im SON jederzeit eine Liste von $r - 1$ anderen Knoten vor, die als RTMs verwendet werden können. Der Ort der Knoten könnte entsprechend eines symmetrischen Replikationsschemas bestimmt werden.

(a) Der Client kontaktiert einen beliebigen Knoten im Scalaris-Ring mit einem Transaktionslog (*translog*) von Lese- und Schreiboperationen für die Validationsphase. Dieser Knoten wird zum TM. Der Knoten wählt einen Transaktions-Identifikatior (*Tid*) und einen Paxos Consensus Identifikator (P_i) für jedes Replikat jedes Datensatzes. Er sendet eine *init_RTM*-Nachricht mit dem *translog*, der *Tid*, allen P_i und den Adressen aller RTMs zu jedem RTM. Zusätzlich sendet der TM an alle Transaktionsteilnehmer (TPs - diejenigen Knoten, die für die Datensätze verantwortlich sind) eine *init_TP*-Nachricht mit dem *translog*, der *Tid*, den RTMs und den jeweiligen P_i.

(b) Jeder TP initiiert einen *Paxos Consensus* mit dem empfangenen P_i. Jeder TP schlägt in einer *accept*-Nachricht an die Akzeptoren entsprechend seiner lokalen Validationsstrategie (diese wird im Folgenden noch explizit beschrieben) entweder *prepared* oder *abort* vor. Im Falle einer Entscheidung für *prepared*, sperrt der TP sein Replikat des Datensatzes. Wenn ein TM oder RTM eine Nachricht von einem TP empfängt, lernt er gleichzeitig die Adresse des jeweiligen Knotens zur späteren Verwendung.

(c) Der TM erhält in jeder *Paxos Consensus* Instanz die Rolle des „Lerners“. Um den TM in die Lage zu versetzen, jede Instanz zu entscheiden, sendet jeder RTM eine Liste von *accepted*-Nachrichten an den TM. Sobald der TM eine Mehrheit von *accepted*-Nachrichten für eine gegebene Instanz P_i erhalten hat, entscheidet er i.

(d) Der TM entscheidet, die Transaktion zu übergeben (*commit*) wenn für jeden Datensatz eine Mehrheit der Instanzen *prepared* entschieden hat, andernfalls entscheidet er *abort*. Wenn die TPs die Entscheidung des TM erhalten haben, führen sie die Änderungen lokal aus, geben die Sperren frei und beenden die Commit-Phase.

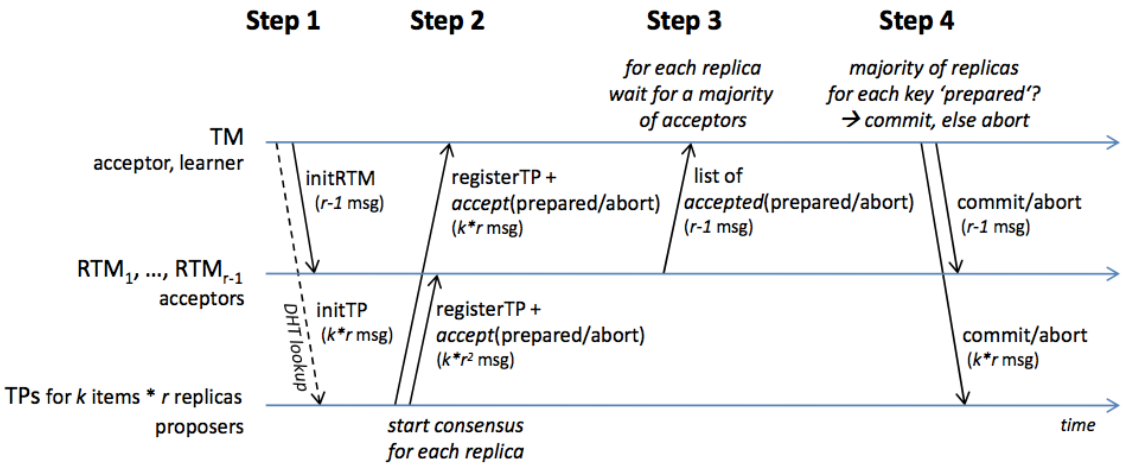

Abbildung 2.1: Zeitdiagramm eines Scalaris-Commits [23]

Eine Annahme, die bei dem hier skizzierten Vorgehen getätigt wurde, ist, dass eine Mehrheit der Akzeptorenmenge (RTMs plus TM) und eine Mehrheit der Replikate für jeden Datensatz korrekt sind. Die folgenden Fehlerfälle können nun auftreten:

- Wenn der TM ausfällt, kann jeder RTM die Rolle übernehmen, indem er eine neue Runde für jede beteiligte *Paxos Consensus* Instanz initiiert. In Scalaris arbeiten die Fehlerdetektoren der RTMs mit unterschiedlichen Grenzen für die

Zeitüberschreitung, so dass niemals mehrere RTMs gleichzeitig die Anführer-Rolle übernehmen wollen. Daher ist der Einsatz eines verteilten Algorithmus zur Wahl eines Anführers nicht notwendig. Der neue TM kann das Protokoll fortführen da der aktuelle Status bezüglich des Konsens in jedem RTM gespeichert ist.

- Sollte ein RTM ausfallen, kann das Protokoll auch mit einer reduzierten Menge von RTMs fortfahren.

- Wenn ein TP in Schritt (b) ausfällt, erhalten TM bzw. ein RTM innerhalb der Zeitüberschreitungsgrenze keine *accept*-Nachricht. In diesem Fall übernimmt der TM bzw. RTM die Rolle in der entsprechenden *Paxos Consensus* Instanz mit Rundenzahl > 1 und stimmt für *abort*, falls noch kein Konsens gefunden wurde, bevor der TP ausfiel. Wenn eine Mehrheit der Repikate für *prepared* stimmt, kann die Transaktion trotzdem ausgeführt werden.

Zuletzt widmen wir uns der Entscheidungsfindung von TPs in der Validationsphase von Transaktionen. Ein TP erhält in Schritt (a) einen Eintrag des Transaktionslogs. Um zwischen *prepared* und *abort* entscheiden zu können, werden folgende Bedingungen überprüft:

- Ist die Versionsnummer noch gültig?

  - *Leseoperationen:* Ist die Versionsnummer in der lokalen Datenbank die selbe wie die im Eintrag aus dem Transaktionslog?

  - *Schreiboperationen:* Ist die Versionsnummer in der lokalen Datenbank gleich der Versionsnummer aus dem Eintrag des Transaktionslogs minus eins?

- Ist die entsprechende Sperre für den Schlüssel verfügbar?

  - *Leseoperationen:* Ist keine Schreibsperre gesetzt?

  - *Schreiboperationen:* Ist weder eine Lese- noch eine Schreibsperre gesetzt?

Enden beide Überprüfungen mit positivem Ergebnis, schlägt der TP *prepared* vor und inkrementiert bei Leseoperationen den Zähler der Lesesperren bzw. setzt bei Schreiboperationen eine Schreibsperre. Andernfalls schlägt er *abort* vor. Empfängt ein TP ein *write commit* in Schritt (d), schreibt er den Wert und die zugehörige Versionsnummer aus dem Transaktionslog in die lokale Datenbank. Für alle Operationen der Transaktion werden zum Schluss, unabhängig von *commit* oder *abort*, die gehaltenen Sperren freigegeben.

Implementierung

Reinefeld et al. beschreiben in [22], dass sich auf Grund der Fehleranfälligkeit von Implementierungen verteilter Algorithmen mit Hilfe imperativer Programmiersprachen und Multithreading in diesem Bereich das Paradigma des *Message Passing*, entsprechend dem *Actor Model*, durchgesetzt hat [11]. Scalaris wurde also auf Basis dieses Modells entwickelt, in dem die grundlegenden Entitäten *Akteure* und *Nachrichten* sind. Jeder Akteur hat einen Zustand, kann Nachrichten empfangen und versenden und weitere Akteure erzeugen. Diese Grundlagen können leicht auf Prozesse und Nachrichten in der Programmiersprache Erlang abgebildet werden [2]. Dadurch wird ein fließender Übergang zwischen dem theoretischen Modell und der eigentlichen Implementierung gewährleistet.

Jeder Scalaris-Knoten besteht aus 8 Komponenten [22]:

- Ein *Fehlerdetektor* überwacht andere Knoten und informiert über Ausfälle.
- Der *Konfigurationsspeicher* gewährt Zugang zur aktuellen Konfiguration und erlaubt die Modifikation von Systemparametern.
- Der *Schlüsselhalter* speichert den Bezeichner des Knotens im SON.
- Der *Statistik-Sammler* sammelt Statistiken und leitet sie an zentrale Statistik-Server weiter.
- Die Komponente *Chord#-Knoten* setzt sich aus Einzelkomponenten zur Wartung des SON und der Routing-Informationen zusammen. Sie verwaltet unter Anderem die Nachfolger-Liste und die Routing-Tabelle und stellt die Funktionalität der SON-Schicht bereit.
- Die *Datenbank* speichert die Attribut-Wert-Paare des Knotens.
- Der *Transaktionsmanager* ist für das Transaktionsprotokoll zuständig.
- Die *Replikations-Komponente* sorgt dafür, dass alle gespeicherten Werte den gewünschten Replikationsgrad aufweisen.

Die Erlang-Prozesse sind in einem Supervisor-Baum angeordnet und werden bei Bedarf vom Erlang-Supervisor neu gestartet. Wenn entweder der Chord#-Knoten oder die Datenbank ausfällt, wird die jeweils andere Komponente explizit beendet und beide werden neu gestartet. Dies ist notwendig um Konsistenz sicherzustellen und ist äquivalent zum Beitritt eines neuen Knotens zum Netzwerk (*crash-stop* Fehlermodell).

Persistenz in Scalaris

Der für potentielle Anwender der Software wahrscheinlich wichtigste Faktor ist die Tatsache, dass die in der DHT gespeicherten Daten nicht persistent auf einen Festspeicher geschrieben werden. Dadurch könnte man den Eindruck gewinnen, dass die gespeicherten Daten in einem gewissen Sinne „flüchtig“ sind. Es sollte jedoch zuerst die Frage gestellt werden, welche Vorteile Persistenz denn überhaupt bringen würde.

Ein Grund könnte sein, dass man sich mit dieser Funktion vor ausfallenden Knoten schützen will, also eine klassische Sicherungskopie anlegen möchte. Wie in Sektion 2.1.2 beschrieben, werden Mehrheitsentscheidungen benutzt. Eine Voraussetzung für diese Herangehensweise ist, dass ausgefallene Knoten die Arbeit vollständig und für immer einstellen (*crash-stop* Fehlermodell). Eine weitere Bedingung sieht vor, dass die Mehrheit der replizierten Datensätze immer verfügbar ist. Ist die zweite Voraussetzung nicht erfüllt, gibt es keine Mehrheit für den entsprechenden Datensatz, d.h. er kann nicht gelesen werden und ist verloren. An dieser Stelle kann auch Persistenz nicht direkt helfen. Fällt jedoch eine Minderheit der Replikate aus, wird der Grad der Replikation wiederhergestellt (Replikate werden auf anderen Knoten neu erstellt).

Fällt ein einzelner Knoten aus und ist nach einiger Zeit wieder verfügbar - was nach dem Scalaris-Fehlermodell (*crash-stop*) bisher nicht vorgesehen ist - so gibt es folgende Herangehensweisen, um den Knoten wieder ins System einzugliedern: Zuerst könnte man die persistent gespeicherten Daten löschen und als neuer Knoten ins System zurückkehren. Dies ist der derzeitige Ansatz und entspricht dem *crash-stop* Modell. Als zweite Möglichkeit könnte der Knoten mit den persistent gespeicherten Daten ins System zurückkehren. Dies würde jedoch Inkonsistenzen hervorrufen, da entsprechende Replikate der Daten des Knotens möglicherweise schon neu erzeugt wurden. Es gäbe dann mehr Replikate als erwünscht, was die Mehrheitsentscheidungen negativ beeinflussen kann. Im Falle eines ausfallenden Knotens, der in das System zurückkehren soll, bringt eine persistente Ablage der Daten auf einem Festspeicher also keine unmittelbaren Vorteile hinsichtlich Robustheit oder Verfügbarkeit des Systems.

Ein zweiter möglicher Grund für die Einführung dieser Funktion könnte sein, dass ein Administrator in der Lage sein möchte, das System als Ganzes herunterzufahren und neu zu starten. Dies wäre mit einem verteilten Snapshot-Mechanismus möglich. Die Snapshots könnten benutzt werden, um das gesamte System nach einer Auszeit wieder hochzufahren und in den Zustand des Zeitpunktes des Snapshots zurückzuversetzen. Nötig werden könnte so etwas beispielsweise wenn der Administrator einen älteren Stand der Daten einspielen will, weil ein Nutzer in der Anwendung versehentlich Daten gelöscht hat oder wenn auf allen Systemen ein Software-Update angewendet werden soll, welches den Neustart der Knoten notwendig macht.

Im Gegensatz zu klassischen relationalen Datenbanken kann Scalaris jedoch nicht Zustände eines beliebigen Zeitpunktes wiederherstellen. Dies ist vom Systemmodell her nicht vorgesehen. Stattdessen könnte man mit den in Kapitel 3 beschriebenen Algorithmen periodische Snapshots implementieren.

2.2 Ereignisse und konsistente Schnitte

Im Folgenden Abschnitt wollen wir die theoretischen Grundlagen, die für die Einführung der Snapshot-Algorithmen im nächsten Kapitel notwendig sind, näher erläutern.

2.2.1 Ereignismodelle

Prozesse in einem verteilten System können bei abstrakter Betrachtung als eine Sequenz von Ereignissen angesehen werden, wobei Ereignisse atomare Operationen eines Prozesses sind, die in Zeit 0 stattfinden [20]. Es gibt in Modellen, die verteilte Systeme beschreiben, 3 Klassen von Ereignissen: Senden von Nachrichten, Empfangen von Nachrichten und interne Ereignisse, die den lokalen Zustand beeinflussen (z.B. Berechnungen). Die Abbildung 2.2 zeigt beispielhaft eine Abfolge von Ereignissen in einem verteilten System. Weiterhin stehen Ereignisse in Beziehung zueinander. In einem Prozess können die lokalen Ereignisse in eine totale Ordnung „$\rightarrow_i$“ gebracht werden, indem man Sie entsprechend der Zeit ihres Auftretens anordnet. Es gilt $e \rightarrow_i e'$ genau dann, wenn e vor e' im Prozess p_i stattfindet. Diese

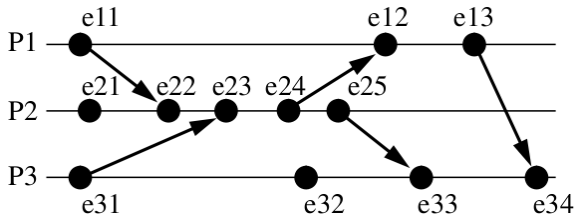

Abbildung 2.2: Beispiel für eine Ereignisabfolge in einem verteilten System [20]

(totale) Ordnung entspricht der kausalen Reihenfolge des Auftretens der Ereignisse. Zusätzlich muss es zu jeder empfangenen Nachricht eine gesendete Nachricht geben. Das zentrale Konzept hinter diesen beiden Beziehungen ist die Kausalitätsrelation, die besagt, dass die Zukunft nicht die Vergangenheit beeinflussen kann. Formal definiert ist eine Ereignisstruktur ein Paar $(E, \rightarrow)$, wobei E die Menge der Ereignisse

in einem verteilten System und $\rightarrow$ eine irreflexive partielle Ordnung auf dieser Menge ist [20]. Mit Hilfe dieser Strukturen lassen sich verteilte Berechnungen abstrakt beschreiben.

Definition 2.1 (Happens-Before-Relation)**.** *Es gilt $e \rightarrow e'$ genau dann, wenn eine der nachfolgenden Regeln zutrifft.*

(a) Es existiert ein Prozess p_i mit $p_i : e \rightarrow_i e'$

(b) e ist das Sende-Ereignis einer Nachricht und e' das entsprechende Empfangs-Ereignis

(c) Es existiert ein e'', so dass $e \rightarrow e''$ und $e'' \rightarrow e'$

Diese Definition der Relation geht auf Lamport [17] zurück. Stehen zwei Ereignisse e und e' in Happens-Before-Relation, ist es möglich, dass e' von e beeinflusst wurde.

2.2.2 Konsistente Schnitte

Der folgende Abschnitt orientiert sich stark an der Arbeit von Mattern [20], der die Definition konsistenter Schnitte wie folgt motiviert: Wenn ein Prozess Nachrichten an alle anderen Prozesse des Systems schickt, um dort lokale Aktionen auszulösen oder Informationen anzufordern, werden diese Nachrichten typischerweise zu unterschiedlichen Zeitpunkten bei Ihren Empfängern eintreffen. Wegen diesen unvorhersehbaren Verzögerungen bei der Nachrichtenübermittlung ist es nicht möglich zu garantieren, dass alle Aktionen, die von den versendeten Nachrichten ausgelöst wurden zum gleichen Zeitpunkt stattfinden. Intuitiv ist ein Schnitt (*cut*) eine ge-

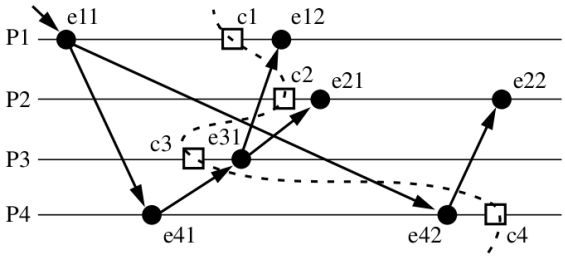

Abbildung 2.3: Beispiel für einen Schnitt [20]

zackte Linie, die ein Ereignisdiagramm wie in Abbildung 2.2 in zwei Teile teilt. Um eine formale Notation zu erhalten, fügen wir der Menge der Ereignisse E für jeden Prozess p_i ein Schnitt-Ereignis (*cut event*) c_i hinzu: $E' = E \cup \{c_1, \ldots, c_n\}$. In Abbildung 2.3 findet sich ein einfaches Beispiel. Ein Schnitt teilt die Ereignismenge E

in zwei Teilmengen: P (die Ereignisse vor dem Schnitt) und F (die Ereignisse nach dem Schnitt), womit wir folgende Definition erhalten:

Definition 2.2 (Schnitt)**.** *Ein Schnitt C einer Ereignismenge E ist eine endliche Teilmenge $C \subseteq E$, so dass aus $e \in C$ und $e' \rightarrow_i e$ folgt $e' \in C$.*

In dieser Form der Notation gingen zwar die Schnitt-Ereignisse c_i vorerst verloren, jedoch ist es trivial, diese hinzuzufügen, sollte man sie benötigen (das Selbe gilt für die Konvertierung in die andere Richtung).

Nach der jetzigen Definition kann es jedoch vorkommen, dass ein Schnitt das Empfangs-Ereignis zu einer Nachricht enthält, nicht aber das entsprechende Sende-Ereignis. Da wir Schnitte benutzen wollen, um (konsistente) Zustände eines verteilten Systems aufzunehmen, reicht diese Definition nicht aus. Die folgende Anpassung löst das Problem.

Definition 2.3 (konsistenter Schnitt)**.** *Ein <u>konsistenter Schnitt</u> C einer Ereignismenge E ist eine endliche Teilmenge $C \subseteq E$, so dass aus $e \in C$ und $e' \rightarrow e$ folgt $e' \in C$.*

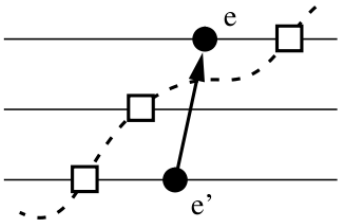

Abbildung 2.4: Beispiel für einen inkonsistenten Schnitt [20]

Ein Schnitt ist also konsistent, wenn jede empfangene Nachricht gesendet wurde (aber nicht notwendigerweise umgekehrt). Der Schnitt in Abbildung 2.3 ist konsistent und in Abbildung 2.4 findet sich ein Beispiel für einen Schnitt, der nicht konsistent ist (e ist Teil des Schnittes, aber der nach der Happens-Before-Relation unmittelbare Vorgänger e' nicht).

Im Folgenden führen wir noch informell (für eine korrekte formale Notation sei auf [20] verwiesen) ein weiteres Konzept zur Prüfung auf Konsistenz eines Schnittes ein. Man nehme an, ein Schnitt in einem Ereignisdiagramm sei auch ein Gummiband. Nun dehne man das Band so, dass es genau vertikal steht. Wenn nun ein Pfeil (der eine Nachricht darstellt) von rechts nach links durch das Gummiband hindurchführt, ist der Schnitt nicht konsistent, ansonsten ist er konsistent. Die Abbildung 2.5 zeigt den *rubber band inconsistency test* am Beispiel.

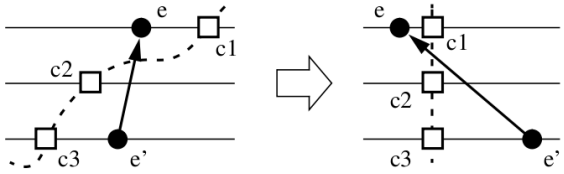

Abbildung 2.5: Der Gummiband-Test [20]

Für einen konsistenten Schnitt gibt es eine mögliche Ausführungsreihenfolge der Berechnungen in dem zu Grunde liegenden verteilten System, es ist also sinnvoll das Konzept des globalen Zustandes eines konsistenten Schnittes einzuführen (ein kausal konsistenter globaler Zustand). Dieser enthält den Zustand aller Kanäle (also diejenigen Nachrichten, die die Linie kreuzen, also in P gesendet und in F empfangen werden) und die Menge der lokalen Zustände der einzelnen Prozesse, aufgenommen zum Zeitpunkt des entsprechenden Schnitt-Ereignisses c_i. Der lokale Zustand eines Prozesses ist dabei der Zustand nach dem letzten lokalen Ereignis in C, oder dem initialen Zustand, falls es kein solches Ereignis gibt. In [14] wird ein konsistenter globaler Zustand formal noch etwas korrekter definiert. Da wir diese Definition in Kapitel 3 mehrfach zur Überprüfung der Eigenschaften der Algorithmen benötigen, geben wir sie hier an.

Definition 2.4 (Konsistenter globaler Zustand)**.** *Sei V die Menge der Prozesse in einem verteilten System und seien $i, j \in V$. Sei weiterhin LS_i der lokale Zustand von Prozess i und SC_{ij} der Zustand des Nachrichtenkanals zwischen den Prozessen i und j. Dann definieren wir den <u>globalen Zustand</u> GS wie folgt:*

$$GS = \{\cup_{k \in V} LS_k, \cup_{k,l \in V} SC_{kl}\}$$

Ein <u>konsistenter globaler Zustand</u> ist ein globaler Zustand, der folgende Eigenschaften erfüllt:

- ***C1**:* $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$
- ***C2**:* $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$

In einem konsistenten globalen Zustand ist also jede als empfangen aufgezeichnete Nachricht auch als gesendet aufgezeichnet und er erfüllt die Kausalitätseigenschaft (keine Nachricht die nicht gesendet wurde wird empfangen).

3 Snapshot-Algorithmen

Kshemkalyani, Raynal und Singhal fertigten 1995 eine grobe Übersicht [14] über die existierenden Snapshot-Algorithmen an. Die Vorstellung eines Teils der ersten 3 Abschnitte (Chandy-Lamport, Lai-Yang, Mattern) orientiert sich an dieser Zusammenstellung und wird vervollständigt durch Hinzuziehen der entsprechenden Primärliteratur.

3.1 Der Algorithmus von Chandy und Lamport

1985 präsentierten Chandy und Lamport [3] ihren Algorithmus zur Erstellung von Snapshots in einem verteilten System. Die Hauptanwendung, mit der sie ihre Arbeit damals motivierten, war das Erkennen von stabilen globalen Zustandsprädikaten (eine Funktion, die die Menge aller globalen Zustände auf wahr oder falsch abbildet, und deren Wert sich, nachdem sie einmal einen angenommen hat, nicht mehr ändert), wie zum Beispiel „die Berechnung ist terminiert“ oder „es liegt ein Deadlock vor“.

3.1.1 Modell

Es mussten eine Reihe von Annahmen getroffen werden, damit bestimmte Eigenschaften des Algorithmus bewiesen werden konnten. Zuerst werden alle Prozesse und Nachrichtenkanäle in einem verteilten System als fehlerfrei angesehen. Jede Nachricht erreicht den Adressaten unverändert und sie erreicht ihn genau ein mal. Weiterhin werden die Kommunikationskanäle als unidirektional und nach dem FIFO-Prinzip[1] aufgebaut angenommen. Bezüglich der Vernetzung der Knoten wird zuletzt angenommen, dass es zwischen zwei Prozessen im System immer einen Pfad gibt (d.h. das Netzwerk ist nicht partitioniert). Im weiteren Verlauf des Kapitels werden wir sehen, dass sich diese Voraussetzungen etwas aufweichen lassen.

Weiterhin können Chandy und Lamport nicht garantieren, dass der Algorithmus

[1] *first-in-first-out* bedeutet, dass die Nachrichtenreihenfolge erhalten bleibt. Nachrichten können auf dem Kanal nicht „überholt“ werden.

einen globalen Zustand aufnimmt, in dem sich das System tatsächlich einmal befand, sondern lediglich, dass er einen aufnimmt, in dem sich das System befunden haben könnte. In Sektion 3.1.3 werden wir diese Aussage formalisieren.

3.1.2 Der Algorithmus

Laut Chandy und Lamport sollte ein solcher Snapshot-Algorithmus zwar in der Lage sein, Nachrichten zu versenden und lokale Aktionen in jedem Prozess anzustoßen, jedoch sollte er dabei niemals die darunterliegende Berechnung beeinflussen (bis auf wenige Verzögerungen). In Abschnitt 3.4 werden wir diese Formulierung in ein formales Kriterium fassen und sehen, dass Chandy und Lamport ihr Ziel erreicht haben.

Der eigentliche Algorithmus läuft dann wie folgt ab:

(a) Ein Prozess entscheidet sich, den Algorithmus zu starten und

  (i) Schreibt den lokalen Zustand nieder

  (ii) Sendet einen „Marker“ auf allen ausgehenden Kanälen (nachdem der Zustand aufgenommen wurde und bevor weitere Nachrichten gesendet werden)

(b) Empfängt ein Prozess einen „Marker“ und hat er seinen lokalen Zustand noch nicht niedergeschrieben

  (i) Schreibt er den lokalen Zustand nieder

  (ii) Sendet einen „Marker“ auf allen ausgehenden Kanälen (nachdem der Zustand aufgenommen wurde und bevor weitere Nachrichten gesendet werden)

  (iii) Startet die Aufnahme für alle eingehenden Kanäle, auf denen noch kein „Marker“ empfangen wurde

(c) Empfängt ein Prozess einen weiteren „Marker“

  (i) Stoppt er die Aufnahme für den entsprechenden Kanal und schreibt die aufgenommenen Nachrichten nieder

3.1.3 Eigenschaften des lokalen Zustandes

Angenommen wir haben eine noch andauernde Berechnung *seq* in einem verteilten System, deren letztes auftretendes Ereignis e_i mit $i \geq 0$ war. Sei nun S_i der globale Zustand des Systems unmittelbar vor dem Ereignis e_i in der Sequenz der Ereignisse. Wir nehmen nun an, dass der Algorithmus von Chandy und Lamport im globalen Zustand S_i initiiert wird und dass er in S_ϕ mit $\phi > i$ terminiert. Wie zu Beginn erwähnt unterscheidet sich der tatsächlich aufgenommene Zustand S^* möglicherweise von allen S_k mit $i \leq k \leq \phi$. Die Autoren wollten folgende Aussagen verifizieren:

(a) S^* ist von S_i aus erreichbar und

(b) S_ϕ ist von S_* aus erreichbar.

Genauer ausgedrückt: Es existiert eine Berechnung seq' wobei

(a) seq' eine Permutation von *seq* ist, so dass S_i, S^* und S_ϕ als globale Zustände in seq' auftreten,

(b) $S_i = S^*$ oder S_i tritt vor S^* auf,

(c) $S_\phi = S^*$ oder S^* tritt in seq' vor S_ϕ auf.

Chandy und Lamport formulierten dann das folgende Theorem.

Theorem 3.1 (Eigenschaften des globalen Zustandes)**.** *Es existiert eine Berechnung seq', so dass*

(a) Für alle k mit $k < i$ oder $k \geq \phi$ gilt: $e'_k = e_k$ und

(b) die Teilsequenz $(e'_k, i \leq k < \phi)$ ist eine Permutation der Teilsequenz $(e_k, i \leq k < \phi)$ und

(c) für alle k mit $k \leq i$ oder $k \geq \phi$ gilt: $S'_k = S_k$ und

(d) es gibt ein l mit $i \leq l \leq \phi$ so dass $S^* = S'_l$.

Der Beweis kann in der Primärliteratur [3] nachgelesen werden.

3.1.4 Variationen des Chandy-Lamport Algorithmus

Nachdem Chandy und Lamport also 1985 die Pioniere waren, folgten später einige Optimierungen des bestehenden Algorithmus, die jeweils einzelne Aspekte verbesserten [14]. Spezialetti und Kearns optimierten in ihrem Algorithmus [31] zum Beispiel die gleichzeitige Initiierung eines Snapshots durch mehrere Prozesse und sorgten für eine effiziente Verteilung der niedergeschriebenen Daten. Ein anderes Beispiel ist der Algorithmus von Venkatesan [35]. Hier wurde das periodische Aufnehmen von Snapshots optimiert, welches für die Anwendung von Wiederherstellungsalgorithmen von Nöten sein kann. Im Algorithmus von Hélary [12] wird das Konzept von „Nachrichten-Wellen“, einem Fluss von Kontrollnachrichten, so dass jeder Prozess im System genau einmal erreicht wird und so dass mindestens ein Prozess entscheiden kann, wann der Fluss terminiert ist, in den Chandy-Lamport-Algorithmus eingeführt, um Funktionen auf dem globalen Zustand ausführen zu können. Shah und Toueg präsentierten eine Variante [29] bei der die Notation eines konsistenten Schnittes etwas abgeschwächt wurde, die jedoch auch mit ausgefallenen Prozessen oder Nachrichtenkanälen im System in gewisser Weise „sinnvolle“ Ergebnisse liefert. Weitere interessante Modifikationen stammen von Mattern [21] und Taylor [33, 5].

7 Die Varianten von Venkatesan, Shah und Toueg, Mattern und Taylor werden im weiteren Verlauf des Kapitels detailliert besprochen.

Venkatesan

Viele Anwendungen benötigen periodisch erstellte globale Snapshots des Systems, zum Beispiel Wiederherstellungsalgorithmen für synchrones Checkpointing oder die Erkennung stabiler Zustandsprädikate. Dies kann mit dem wiederholten Einsatz des Chandy-Lamport-Algorithmus umgesetzt werden, jedoch werden dabei a) zu viele Nachrichten verschickt (wir werden sehen, dass es wirklich zu viele sind) und b) viele redundante Informationen in die Snapshots aufgenommen, die sich seit der letzten Ausführung gar nicht geändert haben. Venkatesan [35] führte aus diesem Grund 1989 das Konzept der inkrementellen Snapshots ein.

Das zu Grunde liegende Modell bei seiner Arbeit besteht aus bidirektionalen FIFO-Kanälen (die Ordnung der Nachrichten, die auf einem Kanal gesendet werden, bleibt erhalten), die zusammen mit den Knoten/Prozessen formal einen Graphen $G = (V, E)$ mit Knotenmenge V und Kantenmenge E formen. Die Güte eines verteilten Algorithmus wird mit der *worst-case communication complexity* gemessen, einer Funktion der Anzahl der Knoten und/oder Kanälen im Netzwerk. Zugeordnet wird die maximale Anzahl an Nachrichten, die vom Algorithmus generiert wird (eine beliebige Ausführung auf einem beliebigen Netzwerk).

Sei nun i ein Knoten im System und sei $\text{snap}_{t_0}(i)$ ein Snapshot des Systems, initiiert von i als die Zeit auf der lokalen Uhr von i t_0 war. Ein inkrementeller Snapshot ist nun ein Status $\text{snap}_t(i)$ für $t > t_0$, welcher auf Basis von $\text{snap}_{t_0}(i)$ ermittelt wurde. Das Ziel ist es, die hohe Wahrscheinlichkeit auszunutzen, dass sich am Status des Systems in der Zwischenzeit nicht viel geändert hat. Der Vereinfachung wegen wird von nun an angenommen, das jeder Snapshot eine Revisionsnummer hat und dass diese mit jeder neuen Instanz streng monoton ansteigt.

Im Folgenden beschäftigte sich Venkatesan mit unteren Schranken bezüglich der Anzahl der versendeten Nachrichten in Protokollen für inkrementelle Snapshots. Der erste inkrementelle Snapshot wird dabei erst geholt, wenn ein kompletter „normaler“ Snapshot vorliegt, der nach dem Algorithmus von Chandy und Lamport ermittelt wurde. Weitere inkrementelle Snapshots werden jeweils erst ermittelt, wenn der vorige Durchlauf beendet ist. Sei nun $U \subseteq E$ die Menge der Kanäle, auf denen seit dem letzten Snapshot Nachrichten versendet wurden. Venkatesan zeigte nun folgende 3 Aussagen, deren Beweise wir hier nicht zitieren. Sie können in [35] nachgelesen werden.

Lemma 3.1. *Jedes Protokoll für inkrementelle Snapshots benötigt im schlechtesten Fall mindestens $|U|$ Nachrichten.*

Lemma 3.2. *Eine untere Schranke für die Anzahl der Nachrichten ($\text{mc}(n)$), die in einem Protokoll für verteilte Snapshots benötigt wird ist in $\Omega(|V|)$, d.h. es existieren $k > 0$ und j_0, so dass für alle $j > j_0$ gilt: $|V| \cdot k \leq |\text{mc}(n)|$.*

Theorem 3.2. *Eine untere Schranke für die Anzahl der Nachrichten, die im schlechtesten Fall in jedem Protokoll für inkrementelle Snapshots verschickt werden müssen ist Element von $\Omega(|U| + |V|)$.*

Beweis: Folgt sofort aus den Lemmata 3.1 und 3.2. □

Nachdem diese theoretischen Grenzen ermittelt wurden, entwickelte Venkatesan ein Protokoll, dass diese Grenzen asymptotisch erfüllen sollte. Zur Vereinfachung ging er davon aus, dass es nur einen Initiator gibt (den *central processor*). Mit den Ergebnissen von [31] lässt sich diese Annahme aber eliminieren. Jeder Prozessor kann beim Initiator jederzeit einen Snapshot anfordern, jedoch wird dieser mit der Ausführung immer warten, bis ein eventuell noch nicht beendeter Durchlauf terminiert und dann einen (inkrementellen) Snapshot initiieren. Sollte kein *central processor* vorhanden sein, so kann es nötig sein, einen Anführer zu wählen. Zu diesem Zweck lassen zum Beispiel Algorithmen verwenden, die Spannbäume über dem Systemgraphen aufbauen [4, 7]. Venkatesan nimmt zudem an, es existiere ein Spannbaum über dem Netzwerk, um die Kommunikationskomplexität seines Protokolls zu minimieren.

Von nun an ist $p \in V$ der *central processor* und $q, r \in V$ repräsentieren beliebige Prozesse im System. Jeder Knoten weiß welche seiner ausgehenden Kanäle in U_q (die

Menge der Kanäle auf denen nach der Aufnahme des lokalen Zustandes Nachrichten verschickt wurden) sind, jedoch kann er nicht exakt bestimmen, welche der eingehenden Kanäle zu dieser Menge gehören. Bei Chandy und Lamport wird angenommen, dass alle ausgehenden Kanäle in U_q sind und es werden entsprechend Marker auf allen ausgehenden Kanälen versendet (und auf allen eingehenden empfangen). Marker auf allen ausgehenden Kanälen zu versenden führt allerdings zu $O(|E|)$ Nachrichten. Um diese Anzahl zu reduzieren, werden nun lediglich auf allen ausgehenden Kanälen $c \in U_q$ Marker verschickt. Für jeden Marker wird eine Bestätigung zurückgesendet. Wenn q eine Bestätigung für alle versendeten Marker erhält, ist sichergestellt, dass die entsprechenden Kanäle korrekt aufgezeichnet wurden. Wenn nun jeder Prozess im Netzwerk Bestätigungen für alle versendeten Marker erhält, ist der inkrementelle Snapshot komplett, da der Zustand jedes Kanals in U korrekt aufgenommen wurde. Das Protokoll verhält sich im konkreten Ablauf sehr ähnlich dem Echo-Algorithmus [4], den auch Mattern später zum Einsammeln von lokalen Zuständen benutzte [21] (vgl. Sektion 3.3.2). Kontrollnachrichten des Typs *init_snap* werden dabei an alle Knoten geschickt um sie zu informieren, dass nun das Protokoll eingeleitet wurde. Nachrichten des Typs *snap_completed* werden von Prozess q im Spannbaum aufwärts geschickt, wenn sichergestellt ist, dass alle Nachfolger von q (q eingeschlossen) ihre lokalen Zustände und die Zustände aller Kanäle in U korrekt aufgenommen haben (wie zuvor beschrieben). Die Nachrichtentypen *marker* und *ack* (Bestätigung) wurden vorher schon erläutert.

In [35] wird das komplette Protokoll sehr implementierungsnah erläutert, worauf wir hier verzichten, da der geneigte Leser die entsprechenden Prozeduren und Implementierungsdetails in der Primärliteratur nachlesen kann.

Venkatesan hat die Korrektheit des Protokolls in [35] bewiesen, indem er gezeigt hat, dass jede Transitnachricht vom Empfänger aufgenommen wird und dass jede Nachricht, die vom Protokoll aufgenommen wird auch tatsächlich zu dem entsprechenden Schnitt gehört.

Die Nachrichtenkomplexität des Protokolls lässt sich relativ leicht analysieren: Auf jedem Kanal, der zum Spannbaum gehört (der Baum hat $|V| - 1$ Kanten) werden jeweils eine *init_snap*- eine *snap_completed*-Kontrollnachricht verschickt, womit die Anzahl bei $2(|V| - 1)$ steht. Zudem werden auf jedem Kanal $c \in U$ genau eine *marker*- und genau eine *ack*-Nachricht verschickt, womit die Gesamtanzahl der Nachrichten, die vom Protokoll generiert werden Element von $O(|V| + |U|)$ ist.

Shah und Toueg

Shah und Toueg präsentierten noch im Erscheinungsjahr der Arbeit von Chandy und Lamport eine Variante [29] von deren Algorithmus, die auch im Falle eines Ausfalls

von Knoten oder Kanälen einen konsistenten (wobei das Konsistenzkriterium etwas abgeschwächt wurde) globalen Zustand liefert.

Auch hier werden unidirektionale Kommunikationskanäle zwischen den Knoten vorausgesetzt. Chandy und Lamport nahmen bei der Vorstellung ihrer Methode an, dass die Kanäle fehlerfrei (es gibt keine Totalausfälle), verlustfrei (es gehen keine Nachrichten verloren), störungsfrei (es werden keine Nachrichten verändert) und reihenfolgeerhaltend (FIFO) sind. In der Arbeit von Shah und Toueg wurden die Voraussetzungen auf die letzteren zwei reduziert. Sowohl Kanäle als auch Prozesse dürfen also „gutartig“ ausfallen, d.h. sie liefern nach dem Ausfall keine falschen Daten o.ä., sondern stellen ihren Dienst komplett ein. Dabei ist es ebenfalls erlaubt, dass ausgefallene Prozesse wiederbelebt werden und das Kanäle nach dem Verlust einiger Nachrichten wieder ordnungsgemäß funktionieren.

Im Algorithmus von Chandy und Lamport war die erfolgreiche Teilnahme jedes Prozesses fundamental, um den globalen Zustand zu bestimmen. In einer fehleranfälligen Umgebung kann dies jedoch unter Umständen nicht möglich sein, da Prozesse ausfallen oder Nachrichten verloren gehen können. Ein „echter“ konsistenter Schnitt nach Definition 2.3 ist also gegebenenfalls nicht möglich. Das Ziel muss also etwas adaptiert werden.

Definition 3.1 (Zwei-Prozess-Schnitt)**.** *Ein Zwei-Prozess-Schnitt für Prozesse i und j ist eine Liste (e_i, e_j) von Ereignissen, wobei e_i und e_j Ereignisse in den jweiligen Prozessen i und j sind.*

Definition 3.2 (Wechselwirkung)**.** *Wir definieren eine Funktion C_{ij} für einen Zwei-Prozess-Schnitt, die dazu dient, die Wechselwirkungen zwischen i und j darzustellen:*

$C_{ij}((e_i, e_j)) = \{e : e$ *ist ein Ereignis in i, welches vor e_i stattfindet oder ein Ereignis in j, welches vor e_j stattfindet und e ist entweder ein internes Ereignis oder ein Kommunikationsereignis zwischen i, j*$\}$

$C_{ij}((e_i, e_j))$ *wird Wechselwirkung genannt, wenn für alle (e, e') in i und j gilt: $e \rightarrow e'$ und $e' \in C_{ij}((e_i, e_j)) \Rightarrow e \in C_{ij}((e_i, e_j))$.*

Definition 3.3 (ϕ-Schnitt)**.** *Ein ϕ-Schnitt ist eine Liste von Ereignissen $(e_1, \ldots, e_n)$ in der einige der e_i unbekannt sein können und dann als ϕ geschrieben werden. Ein ϕ-Schnitt wird als korrekt bezeichnet, wenn für alle $k, l, k \neq l$ gilt: $e_k \neq \phi$ und $e_l \neq \phi \Rightarrow C_{ij}((e_i, e_j))$ ist Wechselwirkung.*

Der eigentliche Algorithmus, der in [29] vorgestellt wurde, sieht in Pseudocode-Darstellung wie folgt aus:

- Initiator i:

```
inst_i = inst_i + 1;
e_i  = das letzte Ereignis;
Prozessstatus = aktueller Status;
Sende (signal_i,inst_i) an alle anderen Prozesse;
Starte fuer jeden Prozess einen Timer;
```

- für alle j:

```
Wird eine Nachricht m empfangen:
if(m == (signal_k, inst_k)) or (m == (msg_k,inst_k)):
        if inst_k > inst_j:
                inst_j = inst_k
                e_j  = das letzte Ereignis;
                Prozessstatus = aktueller Status;
                Status von Kanal (k,j) = leere Sequenz;
                Sende (signal_j,inst_j) an alle anderen
                        Prozesse;
                Starte fuer jeden Prozess einen Timer;
        else if inst_k == inst_k:
                Status von Kanal (k,j) = Sequenz von
                        Nachrichten die nach e_j und
                        vor m empfangen wurden;
                Breche Timer fuer k ab;
        else if inst_k < inst_j:
                noop;
        fi;
else if m == (time-out_k,inst_j):
        Status von Kanal (k,j) = Sequenz von Nachrichten
                die nach e_j und vor m empfangen wurden;
        Wirf alle Nachrichten von k weg, bis
                (signal_k,inst_k) oder (msg_k,inst_k)
                empfangen wird mit inst_k >= inst_j;
fi;
```

Listing 3.1: Der Algorithmus von Shah und Toueg

Der Algorithmus nimmt einen globalen Zustand des Systems mit einem korrekten ϕ-Schnitt auf. Dieser Schnitt wird unter Zuhilfenahme von Signal-Nachrichten erzeugt, die durch das System propagiert werden (`signal_k`). Diese Signale können, wie jede andere Nachricht, unterwegs verloren gehen. Es ist ebenso erlaubt, dass Prozesse `time-out` Nachrichten an sich selbst versenden. Diese `time-out` Nachrichten gehen nicht verloren, es sei denn der Prozess, der sie aktiviert, stürzt ab.

Weiterhin wird jeder Instanz des Algorithmus ein eindeutiger Bezeichner zugewiesen. Es wird, ohne Beschränkung der Allgemeinheit angenommen, dass ein Prozess i den Algorithmus initiiert. Daher ist ein einfaches Zahlenschema ausreichend, um die Instanzen voneinander zu unterscheiden. Die Instanznummer wird bei jedem Signal und auch bei jeder „normalen“ Nachricht mitgesendet. Daher wird die Notation `(Nachrichtentyp, Instanznummer)` für fast alle Nachrichten benutzt, die von einem Prozess versendet werden. Wenn der Nachrichtentyp `msg_k` oder `signal_k` ist, bedeutet dies, dass der Prozess eine normale Nachricht oder ein Signal von Prozess k erhalten hat. Der Nachrichtentyp `time-out_k` beschreibt den Empfang einer Zeitüberschreitung die zu Prozess k gehört. Weiterhin wird jeweils eine lokale Variable

`inst_j` verwaltet, welche die Instanznummer der neuesten Instanz des Algorithmus enthält, an der der Prozess j teilnahm.

Intuitiv ist der korrekte ϕ-Schnitt, der von einer Instanz des Algorithmus gefunden wird, die Liste der Ereignisse $(e_1, \ldots, e_n)$, wobei jedes e_i das Ereignis von Prozess i ist, welches unmittelbar vor dem Empfang des ersten Signals stattfand. Da Nachrichten (insbesondere Signale) verloren gehen können, müssen sie (mit Hilfe der Instanznummer) auch an normale Nachrichten angeheftet werden, da sonst gegebenenfalls kein korrekter ϕ-Schnitt entsteht.

Um die Terminierung des Algorithmus zu erzwingen wurde ein ad-hoc Mechanismus benutzt, der Zeitüberschreitungen involviert. Notwendig ist dies, da keine Annahmen über die Geschwindigkeit eines Prozesses im System gemacht werden. Da es keinen gemeinsamen Speicher gibt, hat ein Prozess i keine andere Möglichkeit zu prüfen, ob Prozess j abgestürzt ist oder der entsprechende Kommunikationskanal ausgefallen ist. Man muss sich, wenn man ein solches Schema benutzt, jedoch Gedanken über „falsche“ Zeitüberschreitungen (Prozesse werden als tot erkannt, obwohl sie es nicht sind) machen. Dieses Problem wurde bei diesem Algorithmus dadurch behoben, dass einem Prozess i erlaubt wird, Nachrichten eines Prozesses j zu verwerfen, wenn i eine Zeitüberschreitung beim Warten auf ein Signal von j festgestellt hat. Es werden erst wieder Nachrichten von j akzeptiert, wenn ein Signal von j eintrifft, welches zu dieser oder einer späteren Instanz des Algorithmus gehört. Im Systemmodell ist der Verlust von Nachrichten erlaubt. Daher gelten diejenigen Nachrichten, die nach dem oben beschriebenen Schema weggeworfen werden, als verloren. Diese Strategie führt nicht zu Inkonsistenzen im ϕ-Schnitt [29].

In [29] wurde die Korrektheit des Algorithmus in folgendem Theorem definiert und bewiesen. Wir werden den Beweis hier nicht zitieren, sondern verweisen auf die Primärliteratur.

Theorem 3.3 (Korrektheit des Algorithmus von Shah und Toueg)**.** *Der Algorithmus ist korrekt, d.h.*

(a) Der Algorithmus terminiert.

(b) Der sich ergebende Schnitt $(e_1, \ldots, e_n)$ ist ein korrekter ϕ-Schnitt

(c) Wenn im ϕ-Schnitt $(e_1, \ldots, e_n)$ $e_j \neq \phi$ dann sorgt der Algorithmus für die Niederschrift des lokalen Zustandes und des Zustandes der Kanäle bezüglich dieses korrekten ϕ-Schnittes.

Mattern

Eine weitere (etwas umfangreichere) Modifikation des Algorithmus von Chandy und Lamport nahm Mattern [21] vor. Diese wird erst in Sektion 3.3.2 besprochen, da dafür die Kenntnis des Algorithmus von Lai und Yang [15] (vgl. Sektion 3.2) notwendig ist.

Taylor

Taylor stellte in ihren Arbeiten über die Beeinflussung von Berechnungen in einem verteilten System durch einen parallel ausgeführten Snapshot-Algorithmus [33, 5], welche wir in Abschnitt 3.4 vorstellen werden, zwei minimale Varianten des Algorithmus von Chandy und Lamport vor.

3.2 Der Algorithmus von Lai und Yang

Zur Einführung in den Algorithmus werden die Erläuterungen aus [14] und [21] verwendet.

Der Algorithmus von Lai und Yang [15] ist für nicht-FIFO Systeme entwickelt worden und basiert auf zwei Beobachtungen bezüglich der Rolle eines Markers in einem FIFO-System (wie er zum Beispiel bei Chandy und Lamport benutzt wird). Die Erste ist die Sicherstellung der Bedingung $C1$ aus Definition 2.4. Beim Lai-Yang-Algorithmus wird diese Rolle von einem Farbschema von Nachrichten und Prozessen übernommen. Dieses Schema sieht wie folgt aus:

(a) Jeder Prozess ist anfangs weiß und wird rot während er einen Snapshot aufnimmt.

(b) Jede Nachricht, die von einem weißen (roten) Prozess gesendet wird, ist weiß (rot) gefärbt. Entsprechend ist weiße (rote) Nachricht gesendet worden, bevor (nachdem) der Sender einen lokalen Snapshot aufnahm.

(c) Jeder weiße Prozess kann jederzeit einen lokalen Snapshot aufnehmen, bevor eine rote Nachricht empfangen und verarbeitet wird.

Offensichtlich ist der entstandene Schnitt konsistent, da keine Nachricht, welche nach dem Schnitt gesendet wurde, vor dem Schnitt empfangen wird. Ein expliziter Marker wird nicht benötigt. Stattdessen werden die (bisher) nötigen Informationen

„Huckepack“ (mittels *piggybacking*) in den ohnehin existierenden Nachrichtenfluss eingebaut. Es muss lediglich sichergestellt werden, dass ein Prozess in der Lage ist, zu erkennen, ob eine eingehende Nachricht rot oder weiß gefärbt ist, bevor diese Nachricht in irgendeiner Weise den lokalen Zustand beeinflusst. Auch muss (vor allem in Systemen mit viel Nachrichtenverkehr) garantiert werden, dass die Entscheidung, ob eine Nachricht rot oder weiß gefärbt ist, nur wenig Zeit kostet.

Um zu garantieren, dass der Algorithmus in endlicher Zeit terminiert, muss sichergestellt werden, dass jeder Prozess irgendwann einen lokalen Snapshot aufnimmt. Erreicht werden kann dies zum Beispiel mit einem Broadcast von roten Kontrollnachrichten an alle anderen Prozesse nach der Aufnahme des lokalen Snapshots [21]. Solche Broadcast-Schemata können auf verschiedenste Weise implementiert werden [34]. Weiterhin hat man auch bei diesem Algorithmus, ähnlich wie bei Chandy und Lamport, das Problem, dass man die lokalen Snapshots einsammeln muss, um einen globalen Zustand zu erhalten.

Die zweite der zu Beginn erwähnten Beobachtungen ist, dass Marker dafür zuständig sind, den empfangenden Prozess direkt oder indirekt über diejenigen Nachrichten zu informieren, deren *send*-Ereignisse im lokalen Snapshot des sendenden Prozesses enthalten sind. Auf diese Weise ist der empfangende Prozess in der Lage, jene Nachrichten zu finden, die gerade übertragen werden (deren Sendeereignis also im lokalen Zustand des sendenden Prozesses enthalten ist, aber deren Emfpangsereignis nicht im lokalen Zustand des empfangenden Prozesses vorkommt). Diese „Transitnachrichten“ werden momentan im Algorithmus nicht aufgefangen. Lai und Yang [15] schlagen zur Lösung folgendes Schema vor:

(a) Jeder weiße Prozess führt für jeden Kanal eine History aller weißen Nachrichten (gesendet oder empfangen).

(b) Wenn ein Prozess rot wird, werden diese Histories, zusammen mit dem lokalen Zustand an den Initiator versendet, welcher den globalen Zustand einsammelt.

(c) Der Initiator kann dann nachträglich, mit einem Blick in den lokalen Zustand zweier Prozesse, die Transitnachrichten berechnen: $SC_{ij} = \{\text{send}(m_{ij}) : \text{send}(m_{ij}) \in LS_i\} - \{\text{rec}(m_{ij}) : \text{rec}(m_{ij}) \in LS_j\}$

Wir berufen uns nun wieder auf die Definition 2.4. Die Bedingung $C2$ ist erfüllt, weil eine rote Nachricht nicht in den lokalen Zustand des empfangenden Prozesses aufgenommen wird und weil der Status des Kanals die Differenz zweier Mengen von weißen Nachrichten ist. $C1$ gilt, da eine weiße Nachricht m_{ij} in den Zustand von j aufgenommen wird, falls j die Nachricht empfängt, bevor er den eigenen Zustand aufnimmt. Sonst ist $m_{ij} \in SC_{ij}$.

Bei dieser Herangehensweise muss jeder Prozess die Histories aller Kanäle aufneh-

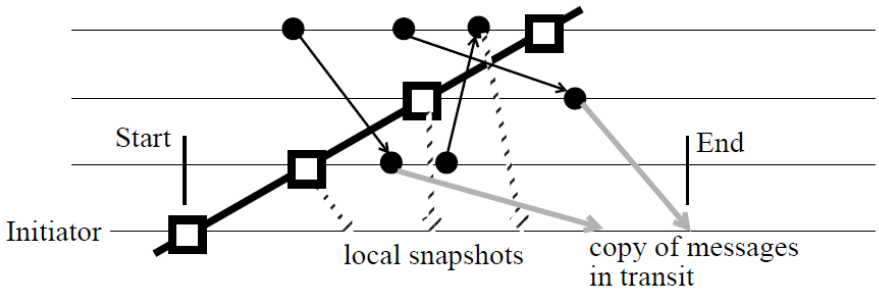

Abbildung 3.1: Das Prinzip des Algorithmus von Mattern [21]

men und speichern, da diese Teil der Datenmenge sind, die zum Initiator geschickt wird. Dies kann zu einem erheblichen Datenaufkommen führen (sowohl bezüglich des lokalen Speicherbedarfs als auch im Hinblick auf den Netzwerkverkehr). Lai und Yang beschreiben daher in ihrer Arbeit weiterhin, wie dieser Speicherbedarf reduziert werden kann. Dabei werden nur diejenigen Nachrichten gespeichert, die seit dem letzten Snapshot gesendet und empfangen wurden, sofern der letzte Snapshot vorhanden ist. Für Anwendungen, die periodische Snapshots benötigen, kann dies von großer Bedeutung sein.

3.3 Die Algorithmen von Mattern

3.3.1 Adaption von Lai und Yang

Mattern motiviert die Einführung seiner ersten Methode [21] mit der Ineffizienz der Handhabung der „Transitnachrichten“ bei Lai und Yang. Er schlägt vor, eine Kopie der weißen Nachrichten, die von roten Prozessen empfangen werden (dies sind genau die, die lokal gespeichert werden müssen um im Endeffekt einen konsistenten globalen Zustand nach Definition 2.4 zu erhalten) direkt zum Initiator weiterzuleiten. Nachdem der Initiator die letzte Kopie aller Transitnachrichten erhalten hat, hat er den kompletten globalen Zustand vorliegen. Mattern illustriert dies mit Hilfe der Abbildung 3.1. Das Farbschema von Nachrichten und Prozessen und die damit zusammenhängenden Regeln wurden von Lai und Yang [15] übernommen (vgl. Sektion 3.2). Auch hier existiert das Problem der Terminierungserkennung. Der Initiator bekommt zwar alle Transitnachrichten als Kopie, jedoch kann er nicht entscheiden, welche die Letzte ist.

deficiency counting

Mattern argumentiert, dass prinzipiell jeder Algorithmus zur Terminierungserkennung bei nicht-FIFO-Systemen anwendbar ist, dass aber die Methode *deficiency counting* besonders attraktiv sei. In dieser Methode wird jeder Prozess mit einem Zähler ausgestattet, der folgenden Status beinhaltet: die Anzahl an Nachrichten, die der Prozess zu irgendeinem anderen geschickt hat minus die Anzahl der Nachrichten die er von irgendeinem Prozess empfangen hat. Wird der Status dieser Zähler von jedem Prozess eingesammelt, erhält der Initiator eine konsistente Sicht auf alle Zähler und kann berechnen, wie viele Nachrichten für diesen Schnitt Transitnachrichten sind und wie viele Kopien er entsprechend erhalten müsste.

Vektorzähler

Mattern argumentiert weiterhin [21], dass es unter Umständen - insbesondere wenn es unpassend ist, den Initiator beim Empfang jeder einzelnen Transitnachricht zu informieren - nötig sei, auf eine andere Strategie zurückzugreifen: das Prinzip der Vektorzähler [18, 19]. Damit ist es möglich, eine Terminierung nach insgesamt zwei Runden mit Kontrollnachrichten zu erreichen. Er setzt für die Verteilung dieser Kontrollnachrichten die Existenz eines Rings als Kontrollstruktur über dem Netzwerk voraus. Bei der Vektorzähler-Methode zählt jeder Prozess i die Anzahl der weißen Nachrichten die an Prozess $j \neq i$ versendet wurden in der j-ten Komponente eines lokalen Vektors V_i der Länge n (die Anzahl der Prozesse). Die „eigene“ Komponente $V_i[i]$ wird immer dann dekrementiert, wenn eine weiße Nachricht von irgendeinem Prozess empfangen wird. Eine den Ring zirkulierende Kontrollnachricht mit einem Kontrollvektor C sorgt für die Aufrechnung (mittels Vektoraddition) der lokalen Vektoren und setzt die lokalen Vektoren auf Null. In der ersten Kontrollrunde werden zudem alle noch weißen Prozesse rot gefärbt und die lokalen Zustände eingesammelt. Nach Beendigung der ersten Runde enthält $C[i]$ die Anzahl der weißen Nachrichten, die zu Prozess i unterwegs sind (weiß bezüglich des Schnittes, der von der Kontrollrunde induziert wurde). Wenn es nach Runde eins ein $i \in \{1, \ldots, n\}$ mit $C[i] > 0$ gibt, dann ist eine zweite Runde notwendig. In dieser zweiten Runde wartet die Kontrollnachricht bei jedem Prozess bis alle weißen Transitnachrichten empfangen wurden (nach der ersten Runde werden keine neuen weißen Nachrichten mehr erzeugt). Weiterhin wird in dieser zweiten Runde der eigentliche Inhalt der Transitnachrichten eingesammelt und zum Initiator weitergeleitet, eventuell nach selbst definierten Kriterien gefiltert.

Mattern [21] diskutiert weiterhin mögliche Verbesserungen und Variationen. Zum einen müssen in Runde zwei nur diejenigen Prozesse nochmals besucht werden, für die $C[i] > 0$ gilt. Weiterhin können die Vektoren so kodiert werden, dass nur die

Komponenten gespeichert und versendet werden, die ungleich Null sind. Soll der Algorithmus mehrfach ausgeführt werden, ist es zudem sinnvoll, zwei lokale Vektoren zu benutzen, um weiße und rote Nachrichten zu zählen. Im Falle einer Wiederholung müssten dann die Farben vertauscht werden.

3.3.2 Adaption von Chandy und Lamport

Eine weitere in [21] vorgestellte Methode benötigt keine spezifische Kontrollstruktur und kann direkt aus dem Algorithmus von Chandy und Lamport abgeleitet werden. Es werden keine Vektorzähler benutzt und es ist lediglich eine Runde nötig, um zu terminieren. Ansonsten werden die Prinzipien des im vorigen Abschnitt beschriebenen Protokolls beibehalten (Färbung von Nachrichten und Prozessen, Warten auf die Transitnachrichten nach dem ersten Schnitt).

Will man den Algorithmus von Chandy und Lamport [3] (vgl. 3.1) auf nicht-FIFO-Kommunikation anpassen, müssen zwei Fälle berücksichtigt werden.

(a) Eine Nachricht von Prozess i zu Prozess j, die nach dem Marker gesendet wurde, wird vor dem Marker empfangen.

(b) Eine Nachricht von i and j, die vor dem Maker gesendet wurde, wird nach dem Marker empfangen.

Beim Umgang mit Fall (a) greifen wir auf bekannte Methoden zurück: Ein Prozess wird in dem Moment rot, in dem er seinen lokalen Zustand aufnimmt und die Farbe wird als Statusinformation an jede Nachricht angeheftet. Empfängt also ein Prozess eine rote Nachricht ohne vorher einen Marker empfangen zu haben, weiß er dass er in einem FIFO-System einen Marker empfangen hätte. Empfängt ein weißer Prozess eine rote Nachricht, werden die üblichen Aktionen ausgelöst (Verteilung von Markern, Änderung der Farbe, Aufnahme des lokalen Zustandes, Start der Aufnahme der Transitnachrichten) und der daraufhin eintreffende Marker wird einfach ignoriert.

Der Fall (b) lässt sich auch in folgende Frage fassen: Wann kommt die letzte weiße Nachricht in einem Kanal an? Im FIFO Fall ist das einfach, weil die Marker das Ende der aufzunehmenden Kette von Nachrichten signalisieren. Im nicht-FIFO-Fall kann das Selbe erreicht werden, indem man die Marker „langsamer“ als die Basisnachrichten macht: Dem Marker wird die Anzahl der (weißen) Nachrichten, die vor ihm versendet werden angeheftet. Die Verarbeitung eines Markers kann dann verzögert werden, bis alle vor ihm abgesendeten Nachrichten empfangen wurden. Um dies umzusetzen muss ein Prozess für jeden Kanal die Anzahl der gesendeten weißen und roten Nachrichten zählen. Die Lösung für beide Fälle ((a) und (b)) kann durch Hinzufügen der Anzahl der entlang eines Kanals gesendeten weißen Nachrichten zu

jeder roten Nachricht, die durch den entsprechenden Kanal geht, erreicht werden. In einem nicht-FIFO-System entspricht der Status eines Kanals der Menge aller weißen Nachrichten die von einem roten Prozess empfangen werden.

Um einen globalen Snapshot zu erhalten, müssen, wie gehabt, die lokalen Zustände (inklusive der jeweiligen Nachrichtenkanäle) eingesammelt werden. Wenn es nur einen einzigen Initiator gibt und das System mit bidirektionalen Kommunikationskanälen ausgestattet ist, lässt sich das Problem mit Hilfe des Echo-Algorithmus [4] lösen: Wenn ein Prozess den ersten Marker oder die erste rote Nachricht erhält, sendet er Marker an alle Nachbarn bis auf den, von dem der Marker oder die entsprechende Nachricht kam. Ein Prozess, der keinen anderen Kanal hat, wartet (nach obiger Methode), bis alle weißen Nachrichten empfangen wurden und sendet dann eine *echo*-Nachricht (die den lokalen Zustand enthält) an den entsprechenden Prozess. Dabei verhalten sich die *echo*-Nachrichten wie Marker, sie enthalten also die Anzahl der weißen Nachrichten, die vor ihnen versendet wurden. Wenn ein Prozess nun Marker oder Echos auf jedem angrenzenden Kanal erhalten hat und wenn alle verbleibenden weißen Nachrichten empfangen wurden, wird ein *echo* mit den zusammengefassten Snapshots (inklusive des eigenen) an den Prozess gesendet, von dem er den ersten Marker bzw. die erste rote Nachricht erhielt. Diese „Echo-Welle“ kommt dann irgendwann beim Initiator an.

3.4 Hemmung und die Algorithmen von Taylor

1989 präsentierte Taylor die erste Arbeit [33], in der ein Kriterium für die eventuell vorhandene Einflussnahme eines Protokolls zur Erstellung eines konsistenten globalen Zustandes definiert wurde. Nicht-hemmende (*non-inhibitory*) Algorithmen sind demnach die, die keine Aktionen in der Ausführung des verteilten Systems verhindern. 1996 erschien dann eine darauf aufbauende Arbeit von Taylor und Chritchlow [5], in der die das Kriterium etwas verfeinert wurde und in der weitere theoretische Grenzen für die Eigenschaften von Snapshot-Protokollen erarbeitet und bewiesen wurden.

In dieser zweiten Arbeit haben Taylor und Chritchlow drei Protokolle näher auf ihre Eigenschaften untersucht: Zwei leichte Varianten des Chandy-Lamport-Algorithmus (*Flood1* und *Flood2*) und eine Variante des *Two-Phase-Commit*-Protokolls für Transaktionen (von nun an *Tree* genannt). Die letzten beiden Algorithmen wurden auch in der ersten Arbeit von Taylor erwähnt und analysiert. Dabei fasste Sie die Ergebnisse wie folgt zusammen:

(a) Es gibt kein nicht-hemmendes (*non-inhibitory*) Protokoll, welches konsistente Zustände in asynchronen, nicht-FIFO-Systemen herstellt.

(b) Jedes nicht-hemmende Protokoll zur Aufnahme von konsistenten Zuständen in einem asynchronen FIFO-System benötigt eine Nachricht auf jedem Kanal -Paar zwischen benachbarten Knoten.

(c) Es existiert ein nicht-hemmendes Protokoll zur Aufnahme von konsistenten Zuständen in asynchronen FIFO-Systemen, welches bis zu zwei Nachrichten zwischen benachbarten Knoten benötigt.

(d) Es gibt ein hemmendes Protokoll zur Aufnahme von konsistenten Zuständen in asynchronen FIFO- und nicht-FIFO-Systemen, welches genau $3(n-1)$ Nachrichten benötigt.

Taylor beweist alle Aussagen, auf das Zitat der Beweise verzichten wir an dieser Stelle jedoch und verweisen den Leser auf [33]. Die Aussagen (a) und (b) werden dabei mit Hilfe formaler Techniken gezeigt und die Aussagen (c) und (d) jeweils unter Zuhilfenahme der hier auch schon angesprochenen Algorithmen (*Flood2* und *Tree*).

In [5] wird das Hemmungs-Kriterium noch etwas verfeinert. Dazu wurde eine Freigabe-Relation eingeführt, die für lokale Zustände beschreibt, welche Ereignisse als nächstes auftreten könnten (entsprechend den Transitionen die laut dem lokalen Prozesszustand erlaubt sind).

Definition 3.4 (Sperrung)**.** *Ein System-Ereignis (also ein Ereignis, welches nicht zur Ausführung des Protokolls gehört) e_i wird in einem Zustand vom Protokoll gesperrt genannt, wenn die Menge der Systemereignisse e_i freigibt, aber die Menge aller Ereignisse (inklusive der durch das Protokoll verursachten) e_i nicht freigibt.*

Definition 3.5 (Hemmung)**.** *Ein Protokoll wird nicht-hemmend genannt, wenn kein Systemereignis von irgendeiner Ausführung des Protokolls gesperrt wird. Ein Protokoll, welches Systemereignisse sperrt wird hemmend genannt.*

Ein nicht-hemmendes Protokoll behindert also das darunterliegende System nicht. Darüber hinaus war eine Unterscheidung zwischen Blockierungen durch lokale Ereignisse und Blockierungen durch Warten auf Nachrichten von anderen Prozessen erwünscht.

Definition 3.6 (Lokale Verzögerung)**.** *Sei e_i ein Ereignis, welches von einem Protokoll in Zustand l_i in Ausführung r gesperrt wird. Wir nennen e_i lokal verzögert, wenn es eine Ausführung r' gibt, so dass*

(a) r'_i enthält eine Erweiterung l'_i von l_i.

(b) e_i wird nicht vom Protokoll gesperrt (in l'_i)

(c) $l'_i - l_i$ *enthält keine Empfangs-Ereignisse*

Damit erhalten wir die gewünschte Unterscheidung.

Definition 3.7 (globale/lokale Hemmung)**.** *Ein Protokoll weist lokale (globale) Hemmung auf, wenn ein (kein) in irgendeiner Ausführung gesperrtes Ereignis lokal verzögert ist.*

Dies impliziert, dass in allen Ausführungen eines global hemmenden Protokolls Ereignisse durch Warten auf Nachrichten eines anderen Prozessors verzögert werden. Zuletzt soll die eventuell vorhandene Verzögerung von Sende- oder Empfangs-Ereignissen durch ein hemmendes Protokoll Beachtung finden.

Definition 3.8 (Sende-/Empfangs-Hemmung)**.** *Ein Protokoll weist Sende-Hemmung (Empfangs-Hemmung) auf, wenn es lokal oder global verzögerte Sende-Ereignisse (Emfpangs-Ereignisse) gibt.*

Taylor und Critchlow [5] beweisen dann, mit dieser geänderten Notation drei weitere Aussagen (zusätzlich zu (a) bis (d), die nur unter der „einfachen“ Hemmungs-Definition aus der ersten Arbeit gelten).

Theorem 3.4. *Es gibt kein Protokoll für nicht-FIFO-Systeme, welches globale Empfangs-Hemmung und lokale Sende-Hemmung aufweist. Diese Aussage bleibt auch bestehen, wenn nur Systemnachrichten (nicht aber die Protokoll-Nachrichten) konsistent aufgenommen werden sollen.*

Theorem 3.5. *Es gibt kein nicht-hemmendes Protokoll für FIFO-Systeme, selbst wenn Protokoll-Nachrichten nicht konsistent in den Schnitt aufgenommen werden müssen.*

Theorem 3.6. *Es gibt kein Protokoll für FIFO-Systeme welches lokale Sende-Hemmung, nicht aber Empfangs-Hemmung aufweist, wenn Protokoll-Nachrichten konsistent aufgenommen werden sollen.*

3.4.1 Protokolle

In diesem Abschnitt soll es nun um die bereits erwähnten Protokolle *Flood1*, *Flood2* und *Tree* und deren Eigenschaften gehen.

Zuerst befassen wir uns mit dem Protokoll *Flood1*, welches fast dem Algorithmus von Chandy und Lamport entspricht. Es werden Nachrichten entlang jedes Kanals

im System versendet, beginnend mit einem dedizierten Initiator I. Ein konsistenter Schnitt entsteht, weil jede Nachricht, die nach den vom Protokoll initiierten Nachrichten versendet wird, auch nach ihnen ankommen muss (wegen den hier vorausgesetzten FIFO-Kanälen). Die lokalen Zustände, die den konsistenten globalen Zustand formen, werden wie bei Chandy und Lamport unmittelbar nach dem Empfang des ersten „Markers“ aufgenommen (dies schließt das Aufnehmen der Transitnachrichten ein). Man beachte, dass eine explizite Hemmung von Sende-Ereignissen eingesetzt wird. Der Algorithmus sieht (in stark vereinfachter Form) wie folgt aus:

- Initiator I:
  - Nimm den lokalen Zustand auf. Sende (Cut) an alle Nachbarn j, bevor weitere Systemnachrichten an j versendet werden.
- Andere Prozesse i, unmittelbar nach dem ersten Empfang einer Nachricht Cut von einem Nachbarn k:
  - Nimm den lokalen Zustand auf. Sende Cut zu allen Nachbarn $j \neq k$ bevor weitere Systemnachrichten an j versendet werden.

Nun widmen wir uns dem Protokoll, welches zum Beweis der Aussage (c) herangezogen wurde (*Flood2*). Dieser Algorithmus ist eine Variante des Echo-Algorithmus von Chang [4] und des Snapshot-Algorithmus von Chandy und Lamport [3]. Man beachte, dass eine explizite Hemmung von Empfangs-Ereignissen eingesetzt wird.

- Initiator I:
  - Sende (Cut) an alle Nachbarn j, bevor weitere Nachrichten empfangen werden. Nimm den lokalen Zustand auf.
- Andere Prozesse i, unmittelbar nach dem ersten Empfang einer Nachricht Cut von einem Nachbarn k:
  - Sende Cut zu allen Nachbarn $j \neq k$ bevor weitere Nachrichten empfangen werden. Nimm den lokalen Zustand auf.

Taylor beweist, dass die lokalen Zustände einen konsistenten Schnitt in FIFO-Systemen bilden [33]. Weiterhin dient der Algorithmus zum Beweis der Aussage (c) aus der ersten Arbeit. In der dort verwendeten Variante wurde statt der expliziten Hemmung von Empfangs-Ereignissen ein anderes Konzept eingesetzt: Das atomare Send-Receive. Es wurde angenommen, dass unmittelbar nach dem Empfang einer Nachricht (noch bevor weitere Ereignisse ausgelöst werden) weitere Nachrichten in einer atomaren Aktion versendet werden konnten. Beide Konzepte sind - wie Taylor und Critchlow begründeten [5] - äquivalent. Das atomare Send-Receive wurde nicht

als Hemmung angesehen, womit die Aussage (c) aus der Korrektheit des Algorithmus folgt.

Nun wenden wir uns dem Protokoll zu, welches Taylor zum Beweis der Aussage (d) dient. Es ist eine Variante des klassischen Zwei-Phasen-Commit-Protokolls [9]. Dort wird ein Spannbaum über dem Netzwerk benutzt, um die Kommunikation zu minimieren. Es ist hemmend, weil zwischen den beiden Phasen das Senden von Nachrichten verhindert wird. Auch hier gibt es wieder einen dedizierten Initiator I. Drei Nachrichten, namentlich $PrepareCut$, Cut und $Resume$ werden den Baum hinunter, hoch und wieder hinab gesendet. Das Senden von Nachrichten wird ausgeschaltet, wenn sich Cut den Baum hinab bewegt und wieder angeschaltet, wenn $Resume$ aufwärts gesendet wird. Ein Spannbaum kann zum Beispiel mit dem Algorithmus von Chang [4] gewonnen werden. Seine Existenz wird von nun an vorausgesetzt.

(a) Sei T ein Spannbaum über dem Netzwerk mit Wurzel I. In jedem Prozess i gibt es eine Variable $parent(i)$ und eine Liste $children(i)$, die das Elternteil bzw. die Kinder von i in T enthalten. Ein interner Knoten hat ein Elternteil und Kinder, ein Blatt hat nur ein Elternteil, die Wurzel hat nur Kinder.

(b) Initiator I: Sende $PrepareCut$ zu allen Knoten $children(I)$

(c) Jeder interne Knoten i, nach Empfang von $PrepareCut$: Sende $PrepareCut$ an alle Knoten $children(i)$

(d) Jeder Blattknoten i, nach Empfang von $PrepareCut$: Schalte das Senden von Nachrichten ab, sende Cut an $parent(i)$ und nimm den lokalen Zustand auf.

(e) Jeder interne Knoten i, nach Empfang von Cut von allen $children(i)$: Schalte das Senden von Nachrichten ab, sende Cut an $parent(i)$ und nimm den lokalen Zustand auf.

(f) Initiator I, nach Empfang von Cut von allen $children(I)$: Nimm den lokalen Zustand auf und sende $Resume$ an alle $children(I)$.

(g) Interne Knoten i, nach Empfang von $Resume$ von $parent(i)$: Sende $Resume$ an alle $children(i)$, schalte das Senden von Nachrichten wieder an.

(h) Blätter i, nach Empfang von $Resume$ von $parent(i)$: Schalte das Senden von Nachrichten wieder an.

Wie bereits erwähnt, benutzt Taylor [33] den obigen Algorithmus um Aussage (d) der zu Beginn aufgeführten Aussagen-Aufzählung nachzuweisen. Dieser hemmende Algorithmus funktioniert also für FIFO- und nicht-FIFO-Systeme und benötigt lediglich $3(n-1)$ Nachrichten.

Weiterhin könnte man mit einer Änderung am Systemmodell erreichen, dass ein symmetrisches Protokoll zu *Tree*, bei dem statt Sende-Hemmung globale Empfangs-Hemmung verwendet wird, das selbe Ergebnis erreicht. Je nach Anwendung kann dies vorteilhaft sein. Die Modell-Änderung würde vorsehen, dass Prozesse Nachrichten selektiv empfangen können (also dass sie vor dem Empfang in die Nachricht „reinschauen“ können) [5].

Koo und Toueg stellten in [13] einen Algorithmus vor, der *Tree* stark ähnelt. Es gibt jedoch zwei verschiedene Snapshot-Typen (bei Koo und Toueg heißen sie *Checkpoints*): provisorisch und permanent. In der ersten Phase wird vom Initiator propagiert, dass alle Prozesse im Spannbaum einen provisorischen Checkpoint aufnehmen sollen. Die Welle, die im Anschluss den Baum hinauf gesendet wird, stellt sicher, dass jeder Prozess seinen provisorischen Checkpoint aufgenommen hat. Erhält der Initiator positive Rückmeldung von allen seinen Kindern (diese versenden die Rückmeldung nur, wenn sie positive Rückmeldung von ihren Nachkommen erhielten), entscheidet er, dass das soeben aufgenommene Provisorium permanent gemacht wird. Diese Entscheidung wird in der zweiten Welle den Baum hinab propagiert. Erhält der Initiator von einem seiner Nachkommen keine positive Rückmeldung, entscheidet er sich gegen die aktuelle Checkpoint-Menge und propagiert in Schritt zwei die Entscheidung, das Provisorium zu verwerfen. Die Blockade des Sendens von Systemnachrichten erfolgt im Übrigen analog zum *Tree*-Protokoll. In [13] werden noch einfache Optimierungen hinsichtlich Fehlertoleranz und Anzahl der aufgenommenen Snapshots vorgenommen. Die Anzahl der aufgenommenen Checkpoints wird wie folgt reduziert: Ein Prozess p nimmt nur dann (auf Zuruf von q) einen provisorischen Checkpoint auf, wenn q's provisorischer Checkpoint den Empfang einer Nachricht von p ausweist und p's neuester permanenter Checkpoint das Sende-Ereignis zu dieser Nachricht noch nicht enthält. Dies wird durch das Hinzufügen von Sequenznummern zu jeder Systemnachricht erreicht. Hinsichtlich Ausfalltoleranz wird die Existenz eines Fehlerdetektors angenommen und festgelegt, dass ein Prozess, wenn einer seiner Nachkommen ausfällt, bevor dieser seine Entscheidung nach oben versenden konnte, selbst keine positive Rückmeldung nach oben weitergibt. Erhält ein Prozess die Entscheidung bezüglich des Provisoriums nicht, blockiert er, da er sonst riskiert die Entscheidung des Initiators zu ignorieren. Um dieses Problem zu umgehen, müsste die aktuelle Struktur (die ja dem eines Zwei-Phasen-Commit entspricht) durch die eines nichtblockierenden Drei-Phasen-Commits [30] ersetzt werden. Dies würde jedoch zu einem massiven Anstieg der Komplexität führen.

3.4.2 Zusammenfassung

Die letzten drei der vier gezeigten Aussagen aus der ersten Arbeit von Taylor [33] illustrieren einen notwendigen Kompromiss, den man beim Entwurf eines Systems beachten sollte: Entweder die eigentliche Berechnung des Systems wird durch ver-

zögerte Sende-Ereignisse nicht oder nur wenig beeinflusst - in diesem Fall kann und sollte man sich für ein Protokoll mit weniger Nachrichten entscheiden - oder das System wird durch das kurzzeitige Verzögern von Nachrichten massiv beeinträchtigt - dann sollte man die zusätzlichen Nachrichten bei der Erstellung eines konsistenten Schnittes in Kauf nehmen und sich für ein nicht-hemmendes Protokoll entscheiden. Konkret kann das erste Protokoll bei einem voll vernetzten System bis zu $(n^2 - n)$ Nachrichten benötigen (nach Aussage (b) ist gibt es in diesem Fall kein Protokoll mit weniger als $\frac{1}{2}(n^2 - n)$ Nachrichten), was für hinreichend große Netze deutlich schlechter ist als $3(n-1)$ beim vorgestellten hemmenden Protokoll. Die Entscheidung hängt darüber hinaus primär von der Architektur des Systems ab.

4 Analyse der Algorithmen

Nach der theoretischen Einführung der Algorithmen im vorigen Kapitel werden wir uns nun der Evaluierung der Protokolle anhand einer Reihe ausgewählter und für unseren Anwendungsfall interessanter Kriterien zuwenden. Zunächst werden wir dazu die Kriterien vorstellen und diese allgemein auf die Algorithmen anwenden. Weiterhin werden wir eine Betrachtung im Kontext der Implementierung von Scalaris vornehmen. Zu guter Letzt werden wir die Ergebnisse dieser beiden Abschnitte benutzen, um eine Empfehlung für die praktische Implementierung abzugeben.

4.1 Bewertungskriterien

Systemmodell

Die meisten der vorgestellten Algorithmen wurden von jeweiligen Entwicklern auf ein bestimmtes Systemmodell [1] maßgeschneidert. Die entsprechenden Korrektheitsbeweise gelten dann entsprechend auch nur innerhalb der Grenzen des jeweiligen Modells. Es gilt hierbei herauszufiltern, was die Anforderungen an das Modell sind, und ob sie im Kontext von Chord# und Scalaris erfüllt sind bzw. leicht erfüllt werden können.

Weiterhin fällt auch der Punkt Fehlertoleranz unter dieses Kriterium, da die meisten Autoren vorweg im Modell bestimmte Fehlerklassen ausschließen. Da Scalaris jedoch auch für den Einsatz in Umgebungen mit viel *Churn* entwickelt wurde, muss diesem Punkt Beachtung geschenkt werden. Wir werden uns bei der Betrachtung jedoch auf nicht-byzantinische[2] Fehlerquellen beschränken (Ausfall von Knoten und Ausfall von Netzwerkkomponenten).

[1] Die Eigenschaften des *Systemmodells* beziehen sich in diesem Forschungsfeld vorwiegend auf die Gestalt der Nachrichtenkanäle

[2] Nicht-byzantinische Fehler bezeichnen Ausfälle von Systemkomponenten, die zum kompletten Stillstand des jeweiligen Subjektes führen (insbesondere liefern die Komponenten keine falschen Daten an andere Knoten).

Konsistenzmodell

Ein Großteil der vorgestellten Algorithmen orientiert sich bezüglich des von dem von ihnen aufgenommenen Zustandes erfüllten Konsistenzkriteriums an der Happens-Before-Relation (Definition 2.1) von Lamport und an der daraus abgeleiteten Definition eines konsistenten Schnittes, die wir in einer Variante von Mattern als Definition 2.3 niedergeschrieben haben. Die Algorithmen garantieren dann, unter gewissen Voraussetzungen, dass sie einen Zustand aufnehmen, der einem solchen konsistenten Schnitt entspricht. Dies ist ein Zustand in dem sich das System einmal befunden haben könnte. Für unseren Anwendungsfall - ein Transaktionssystem mit Replikation - ist dies auch zwingend notwendig, da es sonst zu nicht erwünschten Effekten kommen kann:

Angenommen ein Client stellt eine *put*-Anfrage an einen Knoten des Ringes. Der Knoten nimmt die Anfrage entgegen und leitet sie an die eigentlichen Adressaten, entsprechend der Aufteilung des Namensraumes, entsprechend des Routing-Schemas und entsprechend des Replikations-Schemas, weiter. Wird nun auf den Adressaten, die die Daten eigentlich speichern sollen, vor dem Empfang der Nachricht ein Snapshot-Ereignis ohne Behandlung der Transitnachrichten ausgelöst, so sind die Daten, die kurze Zeit später eintreffen, verloren. Selbst wenn nicht alle der angesprochenen Nachrichten verloren gehen, kann es dazu kommen, dass zum betroffenen Datensatz nicht genügend Replikate für eine Mehrheit existieren, so dass die Daten nicht abgefragt werden können.

Topologie

Bei diesem Kriterium geht es darum, zu bewerten, ob und wie stark sich ein Algorithmus auf die Existenz einer bestimmten Topologie (zum Beispiel Ring, Stern oder Existenz eines Pfades zwischen je zwei Knoten im Netzwerk) des Netzwerkes verlässt.

Zusätzlich muss in Erwägung gezogen werden, dass die Umgebung hochgradig dynamisch ist, es kann also vorkommen, dass zu einem für den Algorithmus „ungünstigen“ Zeitpunkt ein neuer Knoten dem System beitritt oder dass ein Knoten das System verlässt. Auch solche Fälle sollen hier beleuchtet werden.

Kommunikationskomplexität

Die Kommunikationskomplexität ist das wahrscheinlich am einfachsten und am objektivsten zu bewertende Kriterium. Gemessen wird, wie viele Nachrichten (in Ab-

hängigkeit von der Anzahl der Knoten im System und deren Vernetzung) zur Erstellung eines Snapshot benötigt werden.

Die Arbeiten von Taylor, die wir in Abschnitt 3.4 beleuchtet haben, stellten als ein Ergebnis heraus, dass es einen Konflikt zwischen der Kommunikationskomplexität eines Protokolls und des Grades der Beeinflussung der eigentlichen Aufgaben des darunterliegenden Systems durch das Protokoll gibt. Zu bewerten ist also auch (zusammenhängend mit dem folgenden Kriterium „Hemmung“), inwiefern tiefgreifende Einschnitte, wie zum Beispiel das Sperren des Versandes von Systemnachrichten, akzeptabel sind.

Hemmung von Aktionen

Das Kriterium „Hemmung“ steht in enger Verbindung mit der Kommunikationskomplexität. Hier gilt es zu untersuchen, ob der gewählte Algorithmus Aktionen im darunterliegenden Programm beeinflusst oder hemmt (im Sinne von Taylor aus Sektion 3.4) und inwiefern diese Hemmung die Benutzbarkeit bzw. die eigentliche Funktionalität des Systems negativ beeinträchtigt.

Minimal-invasive Implementierung

Das Ziel der vorliegenden Analyse ist es, eine Empfehlung auszusprechen, die in der Implementierung des entsprechenden Protokolls mündet. Entsprechend muss untersucht werden, wie tief die „Einschnitte“ in das bestehende System sein müssten, um dies umzusetzen. Das Hauptziel ist es, den bestehenden Quellcode weitestgehend unverändert zu belassen, um Regressionen[3] zu vermeiden.

4.2 Bewertung der Algorithmen

Im folgenden Abschnitt sollen nun die im vorigen Kapitel vorgestellten (und unten nochmal mit den fortan für sie verwendeten Kürzeln aufgelisteten) Algorithmen anhand der Kriterien nochmals näher beleuchtet werden, ohne jedoch Bezug auf Scalaris-Implementierungsdetails zu nehmen.

[**CL**] Chandy und Lamport

[3] Als *Regression* bezeichnet man Nebenwirkungen einer Modifikation in bereits getesteten und validierten Teilen von Software

[CLV] Venkatesan (Modifikation von Chandy und Lamport)

[CLST] Shah und Toueg (Modifikation von Chandy und Lamport)

[LY] Lai und Yang

[MLY] Mattern 1 (Modifikation von Lai und Yang)

[MCL] Mattern 2 (Modifikation von Mattern 1 und Chandy und Lamport)

[TF1] Taylor *Flood1*

[TF2] Taylor *Flood2*

[TT] Taylor *Tree*

4.2.1 Systemmodell

In den Tabellen 4.1 und 4.2 findet sich eine Auflistung der zu Grunde liegenden Systemmodelle (i.e. der Eigenschaften der Nachrichtenkanäle) der vorgestellten Algorithmen. Allen Modellen gemeinsam ist, dass es sich um asynchrone Systeme handelt, in denen keine globale Uhr oder irgendeine andere Möglichkeit der Synchronisation existiert.

Algorithmus	unidirektional oder bidirektional	FIFO oder nicht-FIFO
CL	unidirektional	FIFO
CLV	bidirektional	FIFO
CLST	unidirektional	FIFO
LY	bidirektional	nicht-FIFO
MLY	bidirektional	nicht-FIFO
MCL	bidirektional	nicht-FIFO
TF1	unidirektional	FIFO
TF2	unidirektional	FIFO
TT	bidirektional	nicht-FIFO

Tabelle 4.1: Zu Grunde liegende Systemmodelle

Die Tabelle 4.2 gibt eine Übersicht über die an die Nachrichtenkanäle gestellten Anforderungen. Fehlerfreiheit bedeutet dabei, dass es keine Totalausfälle gibt, während Verlustfreiheit suggeriert, dass keine Nachrichten verloren gehen. Weiterhin ist anzumerken, dass alle Algorithmen von störungsfreien Nachrichtenkanälen ausgehen, d.h. eine einwandfreie Funktion kann nicht garantiert werden, wenn Nachrichten während

der Übertragung verändert werden. Bei der Analyse der Algorithmen hinsichtlich der

Algorithmus	fehlerfrei	verlustfrei
CL	terminiert ggfs. nicht	terminiert ggfs. nicht
CLV	terminiert ggfs. nicht	terminiert ggfs. nicht
CLST	Inkonsistenz	ggfs. Inkonsistenz
LY	terminiert ggfs. nicht	ggfs. Inkonsistenz
MLY	terminiert ggfs. nicht	ggfs. Inkonsistenz
MCL	terminiert ggfs. nicht	terminiert ggfs. nicht
TF1	terminiert ggfs. nicht	terminiert ggfs. nicht
TF2	terminiert ggfs. nicht	terminiert ggfs. nicht
TT	terminiert ggfs. nicht	terminiert ggfs. nicht

Tabelle 4.2: Anforderungen an Nachrichtenkanäle

Eigenschaften der Kanäle fällt auf, dass diejenigen, die explizit Steuernachrichten verschicken, besonders anfällig für Ausfälle von Nachrichtenwegen bzw. Verlust von Nachrichten sind. Beim Algorithmus **CLST** nahmen sich die Autoren explizit diesem Thema an, bei allen anderen kann der Verlust von Steuernachrichten dazu führen, dass der Algorithmus nicht terminiert. **CLST** toleriert beide Fehlerklassen (er terminiert auch, wenn Kanäle komplett ausfallen oder einzelne Nachrichten verloren gehen), jedoch kann er keinen konsistenten Schnitt garantieren (daher die Notation „ggfs. Inkonsistenz“). Beim Algorithmus von Lai und Yang und dessen Verbesserung von Mattern kann es im Fall eines Totalausfalls eines Kanals passieren, dass der Algorithmus nicht terminiert (da eventuell ein einzelner Prozess nie anders gefärbt wird), jedoch toleriert er (gegebenenfalls auf Kosten des konsistenten Schnittes) den Verlust einzelner Nachrichten.

4.2.2 Konsistenzmodell

Bei der genaueren Untersuchung der Algorithmen hinsichtlich dieses Kriteriums stellten wir fest, dass sich bis auf **CLST** alle Algorithmen an dem Kriterium für konsistente Schnitte orientieren, welches wir in Definition 2.3 niedergeschrieben haben.

Die fehlertolerante Modifikation des Algorithmus von Chandy und Lamport, die Shah und Toueg vornahmen, liefert unter identischen Voraussetzungen analoge Ergebnisse. Es wurden allerdings zusätzlich die Anforderungen an das Systemmodell abgeschwächt, um gewisse Fehlerklassen tolerieren zu können. Der Algorithmus garantiert auch unter schlechteren Voraussetzungen, dass ein so genannter korrekter ϕ-Schnitt aufgenommen wird, also ein Schnitt, bei dem unter bestimmten Bedingungen (vgl. Definitionen 3.1, 3.2 und 3.3) der Zustand einzelner Prozesse unbekannt

sein darf.

4.2.3 Topologie

Tabelle 4.3 gibt eine Übersicht über die Voraussetzungen hinsichtlich der Netzwerktopologie, die für den Betrieb der verschiedenen Algorithmen notwendig sind. Alle Algorithmen haben eine Voraussetzung gemein: die Existenz eines Pfades zwischen je zwei Knoten im Netzwerk. In Sektion 4.3.3 werden wir allerdings feststellen, dass diese Einschränkung für unsere Anwendung nur bedingt relevant ist.

Algorithmus	Anforderungen an Topologie
CL	Existenz eines Pfades zwischen zwei beliebigen Knoten
CLV	Existenz eines Pfades zwischen zwei beliebigen Knoten
CLST	Routing-Protokoll für Kommunikation zwischen zwei Knoten
LY	Existenz eines Pfades zwischen zwei beliebigen Knoten
MLY	Kontrollstruktur zum Einsammeln der Transitnachrichten
MCL	Existenz eines Pfades zwischen zwei beliebigen Knoten
TF1	Existenz eines Pfades zwischen zwei beliebigen Knoten
TF2	Existenz eines Pfades zwischen zwei beliebigen Knoten
TT	Spannbaum über dem Systemgraphen

Tabelle 4.3: Anforderungen an die Netzwerktopologie

Weiterhin widmen wir uns wie angekündigt noch dem in unserem Kontext sehr wichtigen Aspekt der dynamischen Umgebung. Tabelle 4.4 gibt eine oberflächliche Übersicht über die Fähigkeiten der Algorithmen, während der Ausführung mit hinzukommenden und wegfallenden Knoten umzugehen.

Die Algorithmen, die stark an **CL** angelehnt sind (**CL**, **MCL**, **TF1**, **TF2**), zeigen bezüglich der hinzukommenden Knoten alle das selbe Verhalten: In Abhängigkeit vom Zeitpunkt des Beitretens des neuen Knotens kann es vorkommen, dass er nicht im Snapshot aufgenommen wird (er erhält keinen Marker auf irgendeinem eingehenden Kanal). Dies wäre durchaus tolerabel, da sich der frisch hinzugefügte Knoten wahrscheinlich ohnehin noch sehr nahe am Standard-Zustand befindet. Alternativ kann es passieren, dass der Knoten zu einem sehr unpassenden Zeitpunkt hinzugefügt wird, so dass er keinen Marker erhält, aber andere Knoten auf den eingehenden Kanälen des neuen Knotens lauschen. In diesem Fall terminieren die angesprochenen Algorithmen (zumindest ohne Modifikation) nicht.

Der Algorithmus von Venkatesen toleriert zwar das Hinzufügen von Knoten, jedoch geht die Logik des Algorithmus bei der Erstellung inkrementeller Snapshots von einer konstanten Umgebung aus. Bei **CLST** wird eine konstante Prozessmenge an-

Algorithmus	Beitretende Knoten	Ausfallende Knoten
CL	terminiert ggfs. nicht	terminiert ggfs. nicht
CLV	nicht vorgesehen	nicht toleriert
CLST	nicht vorgesehen	toleriert
LY	toleriert	toleriert
MLY	Kontrollstruktur aktualisieren	Kontrollstruktur aktualisieren
MCL	terminiert ggfs. nicht	terminiert ggfs. nicht
TF1	terminiert ggfs. nicht	terminiert ggfs. nicht
TF2	terminiert ggfs. nicht	terminiert ggfs. nicht
TT	nicht im Snapshot	terminiert nicht

Tabelle 4.4: Die Algorithmen in dynamischen Umgebungen

genommen, da jeder Prozess an jeden anderen Prozess Nachrichten verschickt und entsprechende Timer vorhält. Mit wenigen Modifikationen wäre es wahrscheinlich möglich, hinzukommende Knoten zu tolerieren und sinnvoll in die Snapshot-Menge einzufügen, in seinem Urzustand sind solche Topologie-Veränderungen jedoch nicht vorgesehen. Beim Algorithmus von Lai und Yang hingegen werden überhaupt keine Annahmen bezüglich der Topologie gemacht, so dass neue Knoten in keinster Weise ein Problem darstellen. Sie werden sich wahrscheinlich in absehbarer Zeit rot färben und entsprechend in den Snapshot aufgenommen. Für die Lai-Yang-Modifikation von Mattern gilt mit Einschränkungen das Selbe. Hier muss jedoch der neue Knoten vor Ausführung des rundenbasierten Algorithmus zum Einsammeln der Transitnachrichten in die Kontrollstruktur (den Ring) eingefügt werden. Idealerweise würde man die Kontrollstruktur erst zu diesem Zeitpunkt generieren, wenn die Snapshot-Frequenz nicht zu hoch ist. Beim Algorithmus **TT** hängt die Frage, ob der neue Knoten in den Snapshot aufgenommen wird, davon ab, ob er zu Beginn der Ausführung in die Kontrollstruktur (den Spannbaum) aufgenommen wurde oder nicht. Toleriert wird das Hinzufügen in jedem Fall.

Widmet man sich dem Entfernen von Knoten während der Ausführung der Algorithmen, fällt erneut auf, dass **CL**, **MCL**, **TF1** und **TF2** analoges Verhalten an den Tag legen. Hier führt das Entfernen dazu, dass der Algorithmus nicht terminiert, wenn gerade jemand auf einem eingehenden Kanal lauscht, der von dem entfernten Knoten ausging. Ein simpler Timeout-Mechanismus könnte dieses Problem allerdings beheben. Lauscht niemand auf einem entsprechenden Kanal und werden die Kanäle nach dem Entfernen des Knotens ebenfalls entfernt, werden die Algorithmen nicht beeinträchtigt.

Der Algorithmus von Venkatesan operiert, wie schon erwähnt, unter der Annahme, dass die Umgebung konstant ist. Er toleriert den Ausfall eines Knotens nicht ohne Modifikationen. **CLST** und **LY** tolerieren Ausfälle hingegen ohne Probleme; bei **MLY** muss lediglich die Kontrollstruktur zum Einsammeln der Transitnach-

richten aktualisiert (bzw. auf Basis der neuen Topologie neu generiert) werden. **TT** terminiert ohne zusätzliche Modifikationen nicht, wenn ein Knoten entfernt wird.

4.2.4 Kommunikationskomplexität

Um die Kommunikationskomplexität der verschiedenen Algorithmen vergleichbar zu machen, gehen wir davon aus, dass zwischen zwei verbundenen Knoten immer bidirektionale Kommunikation möglich ist, i.e. es existiert ein bidirektionaler Kommunikationskanal, sofern dies im Systemmodell des jeweiligen Algorithmus vorgesehen und erlaubt ist, bzw. zwei entgegengesetzt gerichtete unidirektionale Kommunikationskanäle. Unter diesen Voraussetzungen erhalten wir die in der Tabelle 4.5 aufgelisteten Ergebnisse. E ist dabei die Menge der Kommunikationskanäle, V die Menge der Knoten im Systemgraphen $G = (V, E)$. Weiterhin erfassen wir nur die zur Aufnahme eines Snapshots nötigen Nachrichten. Das Einsammeln der lokalen Zustände wird nur dann berücksichtigt, wenn dies zur Berechnung des endgültigen globalen Zustandes notwendig ist (also bei **LY** und **MLY**).

Algorithmus	Komplexität
CL	$2\|E\|$
CLV	$2(\|V\| - 1) + 2\|E\|$
CLST	$O(\|V\|^2 \log(\|V\|))$
LY	$O(\|V\| \log(\|V\|))$
MLY	$O(\|V\|)$
MCL	$2\|E\|$
TF1	$2\|E\|$
TF2	$2\|E\|$
TT	$3(\|V\| - 1)$

Tabelle 4.5: Kommunikationskomplexität der Algorithmen

Beim Algorithmus von Chandy und Lamport wird auf jedem unidirektionalen Kanal ein Marker versendet, womit sich für ein bidirektional vernetztes System eine Nachrichtenanzahl von $2|E|$ ergibt. Die Variante von Venkatesan erfordert es, auf jedem Kanal in einem Spannbaum zwei Nachrichten zu versenden ($2(|V| - 1)$) und auf jedem bidirektionalen Kanal eine Marker- und eine Bestätigungsnachricht ($2|E|$). Die Chandy-Lamport-Variante von Shah und Toueg sieht vor, dass jeder Prozess einmalig eine Nachricht an alle anderen Prozesse versendet. Da wir nicht davon ausgehen, dass das System voll vernetzt ist, müssen die Nachrichten also geroutet werden. Wir wissen zudem, dass wir in Chord# in höchstens $\log(|V|)$ Schritten jeden Zielknoten erreichen können. Somit erhalten wir die Komplexitätsklasse $O(|V|^2 \log(|V|))$. Der Algorithmus von Lai und Yang benötigt zur Aufnahme der lokalen Snapshots eigentlich gar keine Nachrichten, da die entsprechenden Informationen an „normale“

Systemnachrichten angeheftet werden. Will man die Terminierung des Algorithmus etwas beschleunigen, so könnte ein einmaliger Broadcast von roten Signalnachrichten an alle Knoten ($O(|V|\log(|V|))$) in Erwägung gezogen werden. Zum Versand der aufgenommenen Nachrichtenlogs an den Initiator sind jedoch bis zu $|V-1|\log(|V|)$ (eventuell sehr umfangreiche) Nachrichten notwendig. Bei der Adaption dieses Algorithmus durch Mattern wurde die Strategie auf eine Kontrollstruktur (einen Ring) maßgeschneidert, wodurch sich die notwendige Nachrichtenanzahl zum Einsammeln der Nachrichtenlogs in $O(|V|)$ befindet und $2|V|$ nicht übersteigt (höchstens 2 Runden im Ring). Matterns Adaption des Algorithmus von Chandy und Lamport hingegen ändert an der Nachrichtenanzahl nichts, sie bleibt bei $2|E|$. Das Selbe gilt für die Algorithmen *Flood1* und *Flood2* von Taylor, die auch nur minimale Variationen von Chandy und Lamport sind. Taylors Algorithmus *Tree* hingegen benötigt 3 Nachrichtenwellen entlang eines Spannbaumes (runter, hoch, und wieder runter), womit die Anzahl der Nachrichten bei $3(|V|-1)$ liegt.

Darüber hinaus muss erneut angemerkt werden, dass die Algorithmen **CLV** und **TT** einen Spannbaum über dem Systemgraphen benötigen, der vorab erzeugt werden muss. Das Erstellen eines solchen Baumes mit dem Echo-Algorithmus benötigt $2|E|$ Nachrichten [4]. **MLY** benötigt ebenfalls eine Kontrollstruktur. Da an dieser Stelle jedoch die Existenz eines Ringes vorausgesetzt wird, müssen wir keine zusätzliche Struktur verwalten (und entsprechend keine weiteren Nachrichten versenden). Der Ring existiert bei Chord# ohnehin.

4.2.5 Hemmung von Aktionen

Analysiert man die Algorithmen hinsichtlich der Hemmung von Systemereignissen (siehe Tabelle 4.6), fällt auf, dass diejenigen Algorithmen, die auf dem Algorithmus von Chandy und Lamport basieren (und natürlich auch **CL** selbst) kurzzeitig lokal Ereignisse hemmen müssen, um die Konsistenz des Schnittes zu gewährleisten. So muss nämlich die Ausführung einzelner Sende- bzw. Empfangsereignisse gegebenenfalls so lange verzögert werden, bis die Aufnahme des lokalen Zustandes beendet ist (vgl. Sektion 3.4.1). Das Selbe gilt im Übrigen für den Algorithmus von Lai und Yang und dessen Modifikationen. **TT** hingegen ist der einzige Algorithmus, bei dem global Ereignisse gehemmt werden, d.h. es entstehen Blockaden durch das Warten auf Nachrichten von anderen Prozessen.

4.2.6 Minimal-invasive Implementierung

Bezüglich der Implementierung der Algorithmen auf einem existierenden System gilt es zuerst festzuhalten, dass es eine Reihe von Anforderungen gibt, die die vorgestell-

Algorithmus	Komplexität
CL	lokale Hemmung
CLV	lokale Hemmung
CLST	lokale Hemmung
LY	lokale Hemmung
MLY	lokale Hemmung
MCL	lokale Hemmung
TF1	lokale Hemmung
TF2	lokale Hemmung
TT	globale Hemmung

Tabelle 4.6: Hemmung von Systemereignissen

ten Strategien gemein haben:

- Eine Prozedur zum Versand von Protokollnachrichten (z.B. Marker oder gesammelte Snapshots)
- Eine Prozedur zur Aufnahme des lokalen Zustandes
- Eine Prozedur zur Aufnahme von Transitnachrichten, getrennt nach Absender bzw. Kanal, ggfs. reihenfolgeerhaltend bzw. mit Sequenznummer oder Zeitstempel versehen
- Die Möglichkeit, auf eingehende Nachrichten (zum Beispiel Marker oder anders gefärbte Systemnachrichten) reagieren zu können, bevor weiterer Code des darunterliegenden Systems ausgeführt wird.

Darüber hinaus verlangen die Algorithmen **CLV**, **LY**, **MLY** und **MCL** die Möglichkeit, zusätzliche Informationen an jede Systemnachricht anheften zu können (*piggybacking*). **MLY**, **CLV** und **TT** benötigen außerdem zur Funktion spezielle Kontrollstrukturen oberhalb des Overlay-Netzes (zum Beispiel einen Spannbaum oder einen Ring) um betrieben werden zu können. Entsprechend muss eine Komponente existieren, die diese Strukturen erzeugt und verwaltet.

4.3 Betrachtung der Algorithmen im Kontext von Scalaris

Im Folgenden Abschnitt wollen wir die oben verwendeten Kriterien noch einmal genauer beleuchten, diesmal jedoch mit Hinblick auf die Implementierung der Al-

gorithmen im Kontext von Scalaris. Es soll also die Relevanz der Einschränkungen bzw. Vorzüge der Algorithmen in diesem Feld untersucht werden.

4.3.1 Systemmodell

Auch wenn die Implementierung von Erlang in Scalaris vermuten lässt, dass eine Erhaltung der Nachrichtenreihenfolge nicht garantiert werden kann (auf Grund der Eigenheiten der Nachrichten-Warteschlange und der *receive*-Anweisung von Erlang), kann die Implementierung dieses Kriterium erfüllen. Dies kommt durch die Verwendung einer eigens entwickelten Komponente zum Empfang von Nachrichten zu Stande. Dort werden zuerst alle Nachrichten empfangen, eine Selektion anhand der Nachrichtenstruktur findet erst später statt. Genau an dieser Stelle könnte die Implementierung eines Snapshot-Algorithmus ansetzen, so dass davon ausgegangen werden darf, dass die Nachrichtenreihenfolge erhalten bleibt.

Weiterhin gibt es in Chord# bidirektionale Kommunikation zwischen Prozessen. Es lässt sich allerdings problemlos eine Betrachtungsweise anwenden, bei der die Kommunikation auf zwei getrennten unidirektionalen Kanälen stattfindet. Setzt man den Snapshot-Algorithmus an der eben erwähnten Stelle an, erhält man ohnehin lediglich alle eingehenden Nachrichten.

4.3.2 Konsistenzmodell

Da wir die Anwendung eines der Snapshot-Algorithmen auf ein Transaktionssystem mit Replikation vorgesehen haben, müssen wir zwingend ein Kriterium analog Definition 2.3 fordern. Andernfalls kann es dazu kommen, dass entweder ganze Datensätze verloren gehen oder (dies ist der wahrscheinlichere Fall) dass der Replikationsgrad einzelner Datensätze nicht stimmt, so dass es zum Beispiel zwei disjunkte Mehrheiten geben kann, womit das Transaktionsprotokoll keine strenge Konsistenz mehr sicherstellen kann. Es sind allerdings auch Strategien denkbar, um die Replikationsstruktur im Nachhinein zu reparieren. Im Wesentlichen gibt es also zwei Herangehensweisen: strikte Erzwingung des konsistenten Schnittes oder Reparatur der Replikate nach dem Einspielen eines Snapshots. In Abhängigkeit vom konkreten Anwendungsfall für den Snapshot an sich kann die eine oder die andere Strategie als die zu bevorzugende herausstellen.

4.3.3 Topologie

Die von mehreren Algorithmen an das Overlay-Netzwerk gestellte Anforderung, dass zwischen zwei Knoten immer ein Pfad existieren muss (also dass das Netzwerk nicht partitioniert ist), können wir weitestgehend ignorieren, da Scalaris zwar in gewissem Sinne tolerant gegenüber Partitionierung ist (vgl. [26, 8]), sich jedoch in diesem Falle zwei (bzw. mehrere) neue geschlossene Systeme (Ringe) bilden, welche für sich betrachtet werden können. So lang einer dieser Ringe so strukturiert ist, dass von allen Datensätzen noch eine Mehrheit der Replikate verfügbar ist, bleibt dieser voll funktionstüchtig.

Mehrere der Algorithmen benötigen zudem eine zusätzliche Kontrollstruktur über dem eigentlichen Systemgraphen, zum Beispiel einen Ring (**MLY**) oder einen Spannbaum (**TT**). Ein Ring mit Routing-Protokoll und Nachfolger-Listen ist in der Grundstruktur ohnehin vorhanden und ein Protokoll zur Erzeugung eines Spannbaumes existiert ebenfalls bereits. Dieses wird zum Versand von Broadcasts innerhalb des Systems verwendet. Weiterhin sei erwähnt, dass die Voraussetzung der Existenz eines Protokolls zur Kommunikation zwischen zwei beliebigen Knoten erfüllt ist.

Bezüglich der dynamischen Umgebung (hinzukommende und das System verlassende Knoten) stellt sich die Frage, ob man solche Veränderungen am System zulassen will, da in so gut wie jedem Fall das Kriterium des konsistenten Schnittes nicht mehr erfüllt werden kann (vgl. Abschnitte 4.2.2 und 4.3.2). Hat man eine Vorrichtung zur Korrektur des der Replikationsstruktur, kann man dies in Kauf nehmen und die entsprechende Prozedur nach Einspielen des Snapshots ausführen.

4.3.4 Kommunikationskomplexität und Hemmung von Aktionen

Bezüglich der Kommunikationskomplexität muss festgehalten werden, dass es fast immer besser ist, wenn die Ausführung eines verteilten Algorithmus unter Zuhilfenahme weniger Protokollnachrichten geschieht. Im Grunde hat dieses Kriterium in unserem Kontext jedoch weniger Bedeutung, da minimale Verzögerung von Systemereignissen (sei es durch die interne Hemmung von Ereignissen bzw. durch Last auf den Kommunikationskanälen) durchaus toleriert werden kann. Sollten intern oder extern Ereignisse kurzzeitig verzögert werden, kann dies schlimmstenfalls zu einer verspäteten Sichtbarkeit der Ergebnisse einer Transaktion oder zu langsamer beantworteten Antworten führen. Viel wichtiger sind Kriterien wie Toleranz gegenüber Fehlern oder Konsistenz. Letzteres kann eher verloren gehen, wenn die Ausführung des Algorithmus auf Grund hoher Kommunikationskomplexität sehr lange dauert und sich zwischenzeitlich die Umgebung ändert.

4.3.5 Minimal-invasive Implementierung

Wir haben in Abschnitt 4.2.6 bereits die notwendigen Vorrichtungen für den generellen Betrieb der Algorithmen erfasst. Nach genauerer Betrachtung der anderen Kriterien im Scalaris-Kontext können wir nun optional eine Vorrichtung zur Erhaltung bzw. Wartung der Replikationsstruktur hinzufügen.

Was die schon angeführten Voraussetzungen angeht, lässt sich die Implementierung der fünf obligatorischen Prozeduren (Versand von Protokollnachrichten, Aufnahme des lokalen Zustandes, Aufnahme von Transitnachrichten, unmittelbare Reaktion auf Protokollereignisse) unter keinen Umständen umgehen und steht daher nicht zur Diskussion. Das Anfügen eines Statusbits an Systemnachrichten hingegen ließe sich umgehen. Die Implementierung ist jedoch auch nicht unmöglich, da der Nachrichtenverkehr im System zentral gehandhabt wird.

Darüber hinaus gibt es zwei Algorithmen (**CLV** und **TT**), die einen Spannbaum über dem Systemgraphen benötigen. Die Implementierung eines Protokolls zur Erzeugung eines solchen Baumes liegt bereits vor, jedoch müssten noch Vorkehrungen zur Erhaltung, Wartung und Zugänglichkeit dieser Struktur geschaffen werden. **MLY** benötigt zum Einsammeln der Informationen ebenfalls eine Kontrollstruktur, ist jedoch auf einen Ring maßgeschneidert. Dieser existiert bei der Verwendung von Chord# ohnehin und muss nicht extra erstellt werden.

4.4 Destillation einer Implementierungsempfehlung

Im Folgenden werden wir die Ergebnisse der Abschnitte 4.2 und 4.3 versuchen in stark vereinfachter aber übersichtlicher Form zusammenzufassen. Obwohl die vorgestellten Algorithmen nicht in jeder Hinsicht miteinander verglichen werden sollten (da ihr Entwurf zum Teil mit Fokus auf sehr unterschiedlichen Ziele geschah), haben wir die wichtigsten Kriterien in Tabelle 4.7 zusammengetragen und eine Bewertung nach einem sehr einfachen Schema vorgenommen. Zur Erläuterung folgt anschließend eine Übersicht über die angewandten Kriterien mit einem kurzen Kommentar zur Bewertung.

SM Die Punkte wurden hier in Abhängigkeit von den Anforderungen an das **Systemmodell** vergeben. Der neutrale Zustand entspricht dem Modell beim Algorithmus von Chandy und Lamport (fehlerfreie FIFO-Kanäle, fehlerfreie Prozesse). Wird von diesen Kriterien eines entfernt, fällt die Bewertung positiv aus (zum Beispiel nicht-FIFO-Kanäle, Fehlertoleranz).

Alg	SM	KM	T	KK	H	MI	DU
CL	0	0	0	0	0	+1	0
CLV	0	0	0	-1	0	-1	-1
CLST	+1	+1	-1	0	0	-1	+1
LY	+1	0	0	+1	0	0	+1
MLY	+1	0	0	+1	0	0	+1
MCL	+1	0	0	0	0	0	0
TF1	0	0	0	0	0	+1	0
TF2	0	0	0	0	0	+1	0
TT	+1	0	-1	+1	-1	-1	-1

Tabelle 4.7: Zusammenfassung der Eigenschaften der Algorithmen

KM Hier fand eine Bewertung hinsichtlich des **Konsistenzmodells** statt. Bis auf **CLST** folgen hier alle Algorithmen der Definition 2.3. **CLST** hingegen liefert im fehlerfreien Fall das selbe Kriterium, kann aber auch im Fehlerfall ein sinnvolles Konsistenzkriterium erfüllen. Daher fällt die Bewertung positiv aus.

T Bezüglich der **Topologie** fällt die Bewertung nur bei denjenigen Algorithmen negativ aus, die eine zusätzliche Kontrollstruktur über dem eigentlichen Systemgraphen benötigt. Bemerkenswert ist, dass **MLY** ebenfalls eine solche Struktur benötigt. Dort ist jedoch eine negative Bewertung nicht gerechtfertigt, da die benötigte Ringstruktur in unserem Anwendungsfall ohnehin vorliegt, gewartet wird und immer verfügbar ist.

KK Bei der Bewertung hinsichtlich der **Kommunikationskomplexität** wurde willkürlich eine Grenze zwischen neutraler und positiver bzw. negativer Bewertung gezogen.

H Hinsichtlich der **Hemmung von Aktionen** weisen bis auf **TT** alle Algorithmen sehr ähnliche Eigenschaften auf. Daher fällt die Bewertung bis auf den einen Algorithmus, bei dem global Ereignisse verzögert werden, neutral aus.

MI Ein sehr einflussreiches Kriterium ist das der möglichen **minimal-invasiven Implementierung**. Hier erhalten die auf dem Algorithmus von Chandy und Lamport basierenden Verfahren eine positive Bewertung, da hier die wenigsten Eingriffe in die bestehende Software-Architektur notwendig sind. **CLST** und **TT** erhalten eine negative Bewertung, da bisher keine Vorrichtung zum Erstellen und Warten eines Spannbaums über dem Systemgraphen vorgesehen ist und diese zusätzlich implementiert werden müsste. **CLV** erhält eine negative Bewertung, da es bei diesem Verfahren notwendig ist, dass jeder Prozess mit jedem kommunizieren kann. Es gibt jedoch in Scalaris bisher keine Vorrichtung, um alle aktiven Prozesse herauszufinden, geschweige denn mit ihnen zu

kommunizieren. **LY**, **MLY** und **MCL** erhalten lediglich neutrale Bewertungen, da es bei deren Implementierung notwendig ist, Statusinformationen an jede Systemnachricht anzuheften (das Selbe gilt im Übrigen auch für **CLV**).

DU Hier wurde das Verhalten der Algorithmen hinsichtlich des Betriebes in **dynamischen Umgebungen** bewertet. Die auf dem Algorithmus von Chandy und Lamport basierenden Algorithmen sind hier der Maßstab für eine neutrale Bewertung. Alle Algorithmen, die hinzukommende oder verlassende Knoten während der Ausführung besser verkraften, erhalten eine positive Bewertung (vgl. Tabelle 4.4) **TT** ist der einzige Algorithmus, der schlechtere Eigenschaften aufweist und erhält daher eine negative Einschätzung.

Gewichtet man nun der Einfachheit halber alle Kriterien gleich, erhalten **LY** und **MLY** die besten Bewertungen (Summe gleich +3). An dieser Stelle zeigt sich dann auch schon die Schwäche des Verfahrens, da **MLY** seinem Vorgänger hinsichtlich der Behandlung von Transitnachrichten trotz gleichem Ergebnis eindeutig überlegen ist. Nichtsdestotrotz ist die Tabelle ein annehmbarer Anhaltspunkt.

4.4.1 Wahl des Verfahrens

Auf Grund der in den vorigen Abschnitten vorgestellten Analyse der vorgestellten Algorithmen hinsichtlich der für uns interessanten Aspekte sprechen wir eine Empfehlung für die Implementierung der Mattern'schen Adaption des Lai-Yang-Algorithmus aus Sektion 3.3.1 aus. Der einzige Nachteil den dieses Verfahren gegenüber anderen konkurrenzfähigen Protokollen hat, ist, dass gegebenenfalls die Sendemethode für Systemnachrichten etwas abgeändert werden muss, da an jede Nachricht die Farbe des jeweiligen Prozesses angeheftet werden muss.

Die Vorteile überwiegen und wurden im Verlauf der Arbeit hinreichend behandelt, so dass hier nur noch eine kurze Aufzählung erfolgt:

- Es werden keine FIFO-Kanäle benötigt
- Ausfallende und beitretende Knoten werden toleriert (führen jedoch gegebenenfalls zu Inkonsistenzen im aufgenommenen Zustand), so lang die entsprechenden Kontrollstrukturen zeitnah aktualisiert werden
- Die geforderte Meta-Struktur zum Einsammeln der lokalen Zustände und der gesammelten Transitnachrichten (ein Ring) ist in Chord# ohnehin vorhanden
- Geringe Kommunikationskomplexität

- Das Einsammeln der lokalen Zustände und der Transitnachrichten ist bereits Teil des in der Primärliteratur vorgestellten Verfahrens und muss nicht extra entworfen werden (was z.B. beim Algorithmus von Chandy und Lamport der Fall wäre)

5 Anpassung des Algorithmus

Nachdem wir uns im vorigen Kapitel für die Anwendung eines Protokolls entschieden haben, versuchen wir in den folgenden Abschnitten, den Algorithmus so abzuwandeln, dass er sich möglichst gut in die existierende Struktur einfügt und dass die Komplexität der Implementierung reduziert wird.

5.1 Grundlegendes Design

Snapshot-Nummer statt Farbe

Wir ersetzen die binäre Färbung der Knoten des Systems aus dem Algorithmus von Lai und Yang aus Sektion 3.2 durch eine fortlaufende Nummerierung mit natürlichen Zahlen. Jeder Knoten hält lokal ein Snapshot-Status Tupel *s3state*[1] vor, in dem die lokale Snapshot-Nummer, die Initatoren einzelner Snapshot-Instanzen und ein Zustands-Bit (lokale Snapshot-Verarbeitung läuft gerade oder nicht) abgelegt werden. Diese *S3*-Nummer wird entsprechend an die in Frage kommenden Nachrichten angehängt. Ist die Snapshot-Nummer einer eingehenden Nachricht größer als die lokale, wird die lokale erhöht. Neue Knoten treten dem System mit $s_i := 0$ bei.

Behandlung von Transitnachrichten

Wir haben uns bei der Entwicklung des Verfahrens frühzeitig gegen dagegen entschieden, die sogenannten „Transitnachrichten“, also Nachrichten, die von einem Knoten vor der Änderung seiner S3-Nummer abgeschickt werden, aber vom Adressaten nach der Änderung dessen Nummer empfangen werden, auf klassische Weise zu behandeln. Die meisten generischen Snapshot-Algorithmen für verteilte Systeme sehen vor, diese Nachrichten in irgendeiner Form aufzunehmen und zusätzlich zu den lokalen Zuständen der Prozesse niederzuschreiben. Sie gehören zum Snapshot. Einen auf diese Weise erstellten Snapshot zur Wiederherstellung eines Systems

[1] Der von uns angepasste Algorithmus von Lai und Yang trägt den Namen *ScalarisSaveSnapshot* (*S3*).

zu verwenden ist gerade dann, wenn man ein vielschichtiges Software-Projekt wie *Scalaris* vor sich hat, sehr aufwändig.

Daher verzichten wir bei unserem Verfahren auf diese Handhabung der Transitnachrichten im Sinne des Algorithmus von Chandy und Lamport aus Sektion 3.1 oder des Algorithmus von Lai und Yang aus Sektion 3.2. Stattdessen nutzen wir das Wissen über die Interna des Transaktionssystems um auf der Datenbank-Ebene für einen Knoten einen konsistenten Zustand herzustellen, ohne alle Transitnachrichten extra aufzunehmen und abzuspeichern.

Handhabung der lokalen Datenbank

Wenn sich die lokale *S3*-Nummer eines Knotens erhöht, wird im Sinne des Algorithmus von Lai und Yang der lokale Zustand niedergeschrieben. In unserem Fall wird eine Kopie der lokalen Datenbank angelegt, die auch nach dem Kopierprozess bearbeitet werden kann. Es existieren dabei maximal 2 Datenbankversionen gleichzeitig, ältere Versionen werden gegebenenfalls verworfen, die entsprechende Snapshot-Instanz wird aus Sicht des Knotens übersprungen. Der Initiator der übersprungenen Instanz erhält eine Nachricht.

Definition 5.1 (konsistenter Zustand einer Scalaris-Datenbank)**.** *Eine Scalaris-Datenbank d befindet sich in einem* ***konsistenten Zustand****, wenn auf ihr weder Schreib- noch Lesesperren existieren.*

Unsere Strategie, die sicherstellt, dass sich die Datenbank-Kopie irgendwann in einem konsistenten Zustand im Sinne der Definition 5.1 befindet, besteht aus zwei Teilen.

Zum einen verhindern wir, dass sich Transaktionen auf ungünstige Weise mit dem *S3*-Algorithmus überlappen, indem wir zwei Änderungen an dem in Sektion 2.1.2 beschriebenen Transaktionssystem von Scalaris vornehmen:

- Ein Client fügt in der Lesephase der Transaktion zu jedem Eintrag im Transaktionslog die lokale *S3*-Nummer hinzu. Der Eintrag wird also ein 6-Tupel, bestehend aus (1) der Operation, (2) dem Schlüssel, (3) einem Statusbit, welches den Erfolg der Operation anzeigt, (4) dem zugehörigen Wert, (5) der Versionsnummer und (6) der *S3*-Nummer.

- Die Validationsstrategie, die ein *transaction participant* (TP) verwendet, um ein Votum für eine Transaktion abzugeben, wird um eine weitere Regel erweitert: Wenn die im Transaktionslog enthaltene *S3*-Nummer streng kleiner ist als die lokale *S3*-Nummer, entscheide *abort*

Definition 5.2 (*S3-i*-aktive Transaktionen)**.** *Sei i eine S3-Nummer. Eine Transaktion, an der ein Knoten als TP teilnimmt, ist aus seiner Sicht S3-**i-aktiv**, wenn der TP vor der Erhöhung der lokalen S3-Nummer auf i bereits ein Votum für prepared oder abort abgegeben hat, aber deren Entscheidung durch den TM noch aussteht. Für diese Menge von Transaktionen existieren also Sperren auf beiden Datenbankversionen.*

Zum anderen stellen wir sicher, dass die Anzahl der Sperren auf der Kopie der Datenbank monoton fällt. Dazu betrachten wir für eine Snapshot-Instanz i die aus Sicht eines TP[2] *S3-i*-aktiven Transaktionen separat. Die Verarbeitung des Transaktionslogs (bzw. die Freigabe der Sperren) im Falle einer positiven (bzw. negativen) Entscheidung durch den TM muss für diese Menge von Transaktionen auf beiden Datenbankversionen ausgeführt werden. Abbilung 5.1 illustriert die Men-

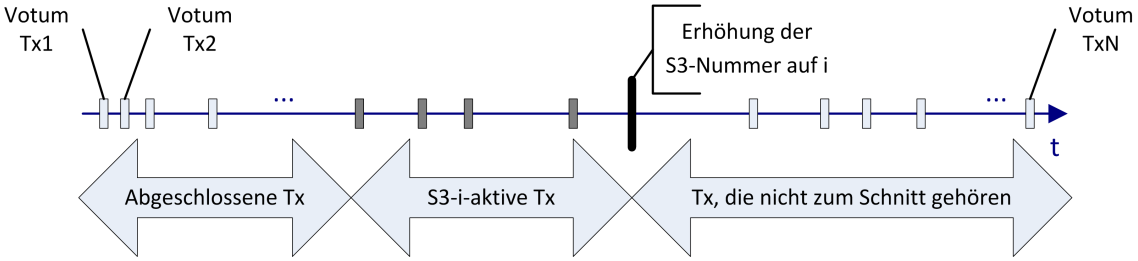

Abbildung 5.1: Transaktionen, an denen ein Knoten als TP teilnimmt als Zeitdiagramm

ge der Transaktionen, an denen ein Knoten als TP teilnimmt (also für die er eine Entscheidung abgibt) auf einem Zeitstrahl. Es wird deutlich, dass die *S3-i*-aktiven Transaktionen ein geschlossenes Intervall besetzen. Zwischen der ersten und der letzten *S3-i*-aktiven Transaktion liegt keine, die nicht zu dieser Menge gehört. Alle Transaktionen, die vorher stattfanden, sind bereits abgeschlossen und ihre Änderungen in der Datenbank festgeschrieben. Sie gehören ohnehin zum durch den Snapshot erzeugten Schnitt. Alle Transaktionen, die später (also nach dem Erhöhen der *S3*-Nummer) ankommen, gehören nicht zum Schnitt und müssen nur auf der Live-Datenbank ausgeführt werden.

Um in der Implementierung herauszufinden, zu welcher Snapshot-Instanz eine Transaktion gehört, fügen wir der Abstimmungs-Nachricht des TP an den TM (*commit* oder *abort*) einen „Cookie“ hinzu. Mit Hilfe dieser *Scalaris*-Funktion können in einer Nachricht beliebige Daten abgelegt werden, die dann vom Kommunikationspartner im Falle einer Antwort *unverändert* zurückgesendet werden. Dort können

[2]Die Teilnahme an einer Transaktion als TP ist die einzige Aktionsfolge, die Änderungen an der Datenbank nach sich zieht, also auch die einzige Aktionsfolge, die wir betrachten müssen

wir also hinterlegen, welche lokale *S3*-Nummer zum Zeitpunkt des Votums aktuell war. Ist diese kleiner als die zum Zeitpunkt des Commits aktuelle (dort wird der Cookie dann zum Vergleich wieder ausgepackt), muss die Transaktion auf beiden Datenbank-Versionen ausgeführt werden.

Lemma 5.1. *Wenn sich die lokale Datenbank-Kopie zu einer Snapshot-Instanz i in einem konsistenten Zustand befindet, finden dort keine Änderungen mehr statt. Der durch i erzeugte Schnitt ist aus Sicht dieses Knotens vollständig.*

Beweis: Alle Änderungen bezüglich der *S3*-i-aktiven Transaktionen werden sowohl bei positiver als auch bei negativer Entscheidung durch den TM auch auf der Datenbank-Kopie ausgeführt. Wenn ein TP also für alle *S3*-i-aktiven Transaktionen irgendwann eine Entscheidung erhält, werden die entsprechenden Sperren auch irgendwann freigegeben. Dies ist durch das Transaktionssystem sichergestellt. □

Lokales Überspringen von Snapshot-Instanzen

Mit dem Nummernschema ist auf Grund von möglicherweise sehr langsamen Nachrichten möglich, dass ein Knoten von einer Snapshot-Instanz $i + 2$ erfährt, bevor er an deren Vorgänger $i + 1$ teilgenommen hat. In diesem Fall ist die Instanz $i + 1$ gegebenenfalls unbrauchbar. Der Knoten, der eine Instanz übersprungen hat, benachrichtigt den Initiator von $i + 1$ darüber.

Initiierung und Terminierung

Wir führen bei unserer Algorithmus-Variante explizit die Rolle des Initiators einer Snapshot-Instanz ein. Dieser verteilt die durch ihn neu vergebene Nummer mittels eines expliziten Broadcasts einer *doSnapshot*-Nachricht. Damit lernen alle Knoten den Initiator kennen. Im Fall des Verlustes dieser Nachricht kann ein Knoten einen Nachbarn nach dem Initiator der aktuellen Instanz befragen. Über deren Existenz erfährt er (früher oder später) durch das implizite Verteilen der Nummern.

Der Initiator hat zudem die Aufgabe, die lokalen Zustände der Teilnehmer des Netzwerkes zu sammeln und zusammenzuführen. Er erhält nach obigem Schema von jedem Knoten eine konsistente Datenbank-Kopie oder eine Fehlernachricht. Mit Hilfe dieser Menge von Datenbanken kann er allein entscheiden, wann der Algorithmus terminieren kann: Wenn er für alle Schlüsselbereiche eine Mehrheit erhalten hat. In diesem Fall kann er allein und ohne zusätzliche Informationen unter Zuhilfenahme der bekannten Kriterien aus der lokalen Validationsphase (vgl. Sektion 2.1.2)

aus der Menge der Datenbanken eine einzige große Hashtabelle konstruieren, die einen konsistenten Schnitt (also einen Zustand, in dem sich das System aus Sicht eines Nutzers einmal befunden haben könnte) repräsentiert. Im Falle zahlreicher Fehlermeldungen durch einzelne Knoten (z.B. verursacht durch hochfrequente Initialisierung mehrerer Snapshots) kann er auch autonom die Frage beantworten, ob dieser Zustand noch erreichbar ist und, wenn zu viele Fehlermeldungen eingegangen sind, eine Fehlermeldung an den Nutzer weitergeben.

5.2 Der Algorithmus

Die Algorithmen 1 und 2 zeigen implementierungsnahe und ereignisbasierte Beschreibung der involvierten Verfahren in Pseudocode-Form.

Algorithmus 1 *S3* participant (every node)

```
1: initialize
2:     s3state.number ← 0
3:     s3state.inProgress ← 0
4:     s3state.leaders ← ⊥
5: end initialize

6: on client working phase
7:     add s3state.number to each tuple in transaction log
8: end
```

```
9: on receipt of ∗ from i
10:     if s3state.number < ∗.s3number then                ▷ begin atomic block
11:         if s3state.inProgress > 0 then
12:             abort active S3 instance, delete d_s3
13:             error message to s3state.leaders[s3state.number]
14:         end if
15:         s3state.number ← ∗.s3number
16:         s3state.inProgress ← 1
17:         if ∗ = doSnapshot then
18:             s3state.leaders[s3state.number] ← i
19:         end if
20:         copy database d_live to d_s3
21:         subscribe to locks on d_s3
22:     end if                                             ▷ end atomic block
23: end
```

```
on subscription trigger from d_s3
    if d_s3 is lock-free then                                  ▷ db is in desired state
        if s3state.leaders[s3state.number] = ⊥ then
            ask other node for leader of this instance
        end if
        send (snapshot_complete, s3state.number, d_s3)
        to s3state.leaders[s3state.number]
        s3state.inProgress ← 0
    end if
end
```

```
on transaction validate for Tx      ▷ this is added to existing validation code
    for all applicable elements entry of transaction log for Tx do
        if Tx.entry.s3number < s3state.number then
            vote abort
        end if
    end for
    do rest of the validation logic
    send vote to TP, add s3state.number as cookie to message
end
```

```
on commit transaction Tx
    j ← extracted snapshot number from cookie in commit message from TM
    if j < s3state.number then
        apply transaction log entries for Tx to d_s3
    end if
    apply transaction log entries for Tx to d_live
end
```

```
on abort transaction Tx
    j ← extracted snapshot number from cookie in abort message from TM
    if j < s3state.number then
        free locks for Tx on d_s3
    end if
    free locks for Tx on d_live
end
```

Algorithmus 2 *S3* initiator

```
1: initialize
2:     s3state.number ← s3state.number + 1
3:     s3leader.snapshots ← ⊥
4:     s3leader.inProgress ← 1
5:     s3leader.number ← s3state.number
6:     broadcast doSnapshot to all nodes in network
7: end initialize
```

```
8: on receipt of (snapshot_complete, snapshot_number, d_i) from i
9:     if s3leader.inProgress > 0 ∧ s3leader.number = snapshot_number then
10:        s3leader.snapshots[i] ← d_i
11:        if s3leader.snapshots contains majority for all key ranges then
12:            merge elements of s3leader.snapshots
13:            using transaction validation criteria          ▷ global snapshot done
14:            save snapshot
15:            s3leader.snapshots ← ⊥
16:            s3leader.number ← ⊥
17:            s3leader.inProgress ← 0
18:        end if
19:    end if
20: end
```

```
21: on receipt of snapshot_error from i
22:     s3leader.snapshots[i] ← ∅
23:     if s3leader.snapshots will never contain a majority for all key ranges then
                                ▷ the above statement has to consider the best case
                                  ▷ in which all remaining nodes answer positively
24:         log error message                              ▷ snapshot instance failed
25:         s3leader.snapshots ← ⊥
26:         s3leader.inProgress ← 0
27:         s3leader.number ← ⊥
28:     end if
29: end
```

5.3 Zusammenfassung des Algorithmus

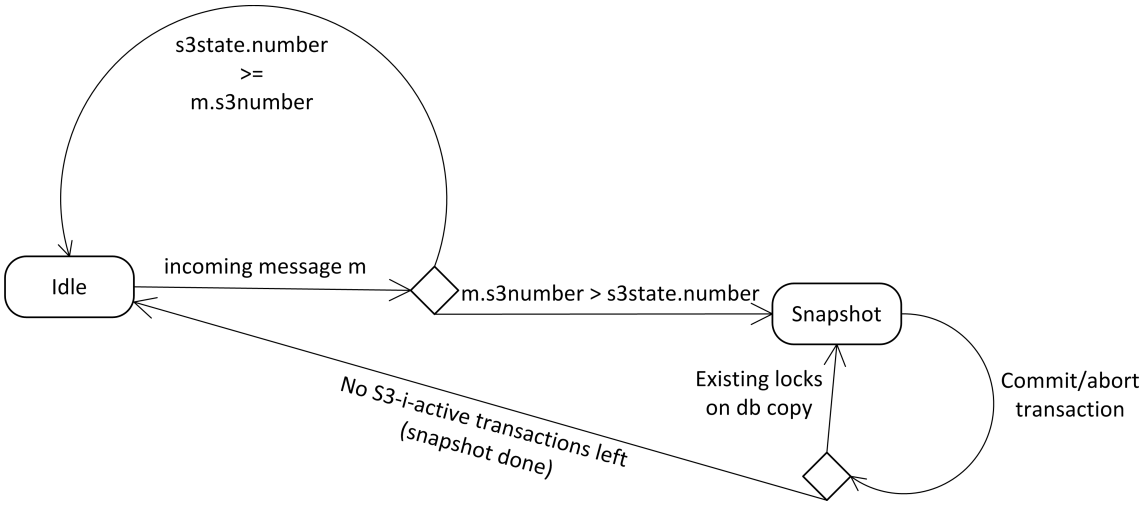

Abbildung 5.2: Zustandsdiagramm S3-Teilnehmer

In Abbildung 5.2 findet sich eine stark vereinfachte Darstellung des Ablaufs des Algorithmus als Teilnehmer (vgl. Algorithmus 1) in Form eines Zustandsdiagramms. Nach der Initialisierung des *s3state*-Tupels aus den Zeilen 1-5 befindet sich der Knoten aus Sicht des *S3*-Algorithmus im Zustand „Idle“. Für jede eingehende Nachricht wird dabei geprüft, ob die *S3*-Nummer der Nachricht größer ist als die lokale (Zeile 10). Wenn dem nicht so ist, verweilen wir in diesem Zustand, es muss nichts getan werden. Ist die Nummer streng größer, begeben wir uns in den Zustand „Snapshot“. Dieser Übergang passiert in den Zeilen 11-21. Dabei werden folgende Schritte ausgeführt: 1) Abbruch eventuell aktiver *S3*-Instanzen und Benachrichtigung von deren Initiatoren; 2) Kopieren der Datenbank und 3) Setzen des *s3state*-Tupels für diese Instanz.

In der Validationsphase hängen wir, wie oben beschrieben, die aktuelle *S3*-Nummer als Cookie verpackt an die Nachricht, die dem TM die Entscheidung mitteilt, an. Beim Beenden der Transaktionen durch Festschreiben der Änderungen oder Freigeben der Sperren in den Zeilen 43-49 und 50-56 packen wir diesen vom TM unverändert zurückgesendeten Cookie wieder aus und können anhand eines Vergleichs mit der aktuellen lokalen Nummer entscheiden, auf welchen Datenbankversionen entsprechende Änderungen ausgeführt werden (Zeilen 45 und 52). Der Zustand „Snapshot“ für die *S3*-Instanz i kann nur verlassen werden, wenn keine *S3*-i-aktive Transaktion mehr existiert, also wenn die Datenbank-Kopie in einem konsistenten Zustand ist (Zeile 25). In diesem Fall findet der Übergang in den Ausgangszustand statt. Dabei wird in den Zeilen 26-31

- gegebenenfalls ein Nachbarknoten nach dem Anführer der in Frage kommenden

Instanz befragt

- der vollständige lokale Zustand an den Anführer versendet
- der lokale Zustand aufgeräumt

Am Ende der Ausführung steht wieder der Zustand „Idle“ und das Warten auf eine Nachricht mit einer *S3*-Nummer streng größer als i. Nicht dargestellt ist der Fall des Abbruchs einer aktiven Snapshot-Instanz. Erhalten wir im Zustand „Snapshot“ die Meldung über eine neuere Snapshot-Instanz, verweilen wir in dem Zustand, räumen die alte Snapshot-Instanz aber auf und nehmen die lokale Initialisierung der neuen vor.

Im stark vereinfachten Zustandsdiagramm aus Abbildung 5.2 werden zudem die Änderungen am Transaktionssystem nicht sichtbar. In den Zeilen 6-7 verdeutlichen wir, dass in der Arbeitsphase des Clients (beim Zusammensetzen des Transaktionslogs) die lokale *S3*-Nummer des Clients zu jedem Eintrag hinzugefügt wird. Bei der Validation der Transaktion in den Zeilen 34-42 wird diese Nummer für jeden in Frage kommenden Eintrag mit der lokalen Nummer des TP verglichen und, sollte sie kleiner sein, für *abort* gestimmt.

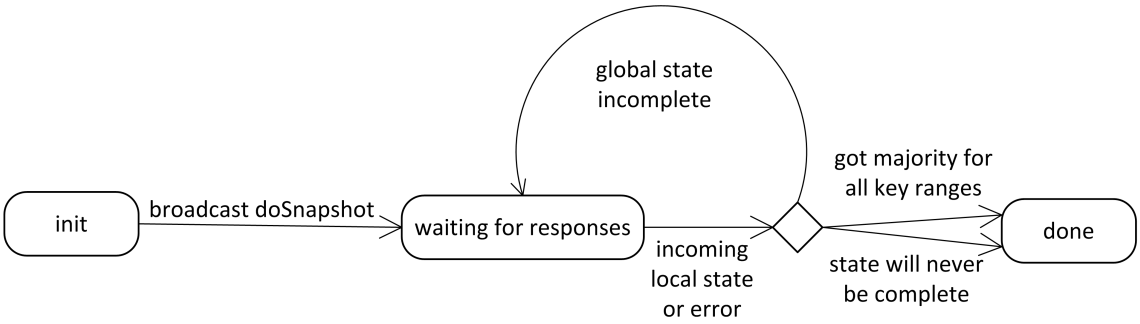

Abbildung 5.3: Zustandsdiagramm S3-Initiator im fehlerfreien Fall

Die Ausführung von *S3* aus Sicht des Initiators lässt sich dem Zustandsdiagramm in Abbildung 5.3 entnehmen (vgl. Algorithmus 2). Hier entscheidet sich der Knoten (auf Zuruf des Nutzers), seinen Initialzustand zu verlassen und führt dabei die in den Zeilen 1-7 dargestellten Aktionen aus. Insbesondere wird die lokale *S3*-Nummer erhöht und eine *doSnapshot*-Nachricht an alle Knoten im Netzwerk gesendet (eine Nachricht an sich selbst eingeschlossen). Daraufhin wartet der Anführer auf eingehende lokale Zustände oder Fehlermeldungen von den Netzwerkknoten. Erhält er von einem Knoten dessen lokalen Zustand (Zeile 8) zu dieser Snapshot-Instanz (Prüfung in Zeile 9), legt er diesen ab und kann allein entscheiden, ob der Algorithmus gegebenenfalls bereits beendet ist (Zeile 11). Ist dies nicht der Fall, verweilt er im Wartezustand. Falls aber bereits eine Mehrheit für alle Schlüsselbereiche vor-

liegt, kann er den globalen Zustand unter Zuhilfenahme der bekannten Kriterien aus der Validationsphase für Transaktionen zusammenfassen und niederschreiben (Zeilen 12-14) und die Ausführung des Algorithmus beenden (Zeilen 15-17). Erhält er statt eines lokalen Zustandes eine Fehlermeldung von einem beliebigen Knoten (Zeile 21), findet ein analoges Verfahren statt. Wenn der Status „Initiator hat für alle Schlüsselbereiche eine Mehrheit erhalten“ nicht mehr erreichbar ist (Zeile 23), wird der Algorithmus mit einer Fehlerausgabe abgebrochen (Zeilen 24-27).

5.4 Ausgewählte Implementierungsdetails

Im Rahmen dieser Diplomarbeit wurde der in diesem Kapitel erarbeitete Algorithmus in das Softwareprojekt *Scalaris* integriert. Während der Umsetzung mussten einige von der Papierform des Algorithmus unabhängige Entscheidungen getroffen werden, auf die hier in Auszügen eingegangen werden soll.

Kopieren der Datenbank via *Copy-On-Write*

Laut Algorithmus soll bei der lokalen Initialisierung eines neuen Snapshots der Zustand, also die Datenbank, kopiert werden. In der Praxis liegt die Datenbank bei den meisten Scalaris-Installationen vollständig im Hauptspeicher. Wenn wir also eine komplette Kopie der Daten vorsehen würden, würden wir die effektive Datenmenge, die im System abgelegt werden kann, halbieren. Wir haben uns also für einen Copy-On-Write-Ansatz entschieden, bei dem nur dann Daten in die Snapshot-Kopie der Datenbank d_{s_i} geschrieben werden, wenn sich Änderungen wirklich ausschließlich auf diese beziehen. Alle anderen Änderungen werden implizit von der Live-Datenbank d_{live} übernommen. Beim Abholen der Snapshot-Daten werden die Live-Daten und die Copy-On-Write-Daten dann zusammengeführt.

Buchführung über Sperren auf der Datenbank

Die Entscheidung, ob ein Snapshot aus Sicht eines Knotens bereits beendet ist, muss schnell getroffen werden können, da sonst der Transaktionsdurchsatz des Systems bei laufendem Snapshot-Algorithmus einbrechen könnte. Aus diesem Grund haben wir alle Datenbankoperationen, die Änderungen durchführen, so angepasst, dass zusätzlich zur eigentlichen Datenbank immer die aktuelle Anzahl von vorhandenen Sperren (Summe der Lese- und Schreibsperren) weitergegeben wird. Im Falle der Initialisierung eines Snapshots wird diese positive, ganze Zahl kopiert und, bis der

Snapshot-Algorithmus beendet ist (also bis die Zahl auf 0 gefallen ist), separat aktualisiert. Auf diese Weise kann mit einem einzigen Ganzzahlvergleich entschieden werden, ob der Snapshot fertig ist.

Der Initiator

Wir haben uns dafür entschieden, die Initiator-Komponente des Algorithmus als separaten Prozess, genauer als eigene `gen_component` (vgl. [26]), innerhalb jedes Scalaris-Knotens zu verankern. Auf diese Weise profitieren wir direkt vom Scalaris Supervisor-Baum und können sicher sein, dass der Prozess bei Ausfall neu gestartet wird. Weiterhin wird damit die Adressierung vereinfacht: Die teilnehmenden Knoten können für die Rücksendung ihrer lokalen Zustände explizit diesen Prozess als Ziel ihrer Nachricht angeben. Auf diese Weise ist es leichter, die Nachrichten zu verfolgen und die Funktion des restlichen Systems wird nicht beeinträchtigt.

Die Initialisierung eines Snapshots von außen erfolgt durch Senden speziellen Nachricht an diesen Prozess. Da die Sprache Erlang es erlaubt, Prozesse mit Namen zu versehen, kann ist dieser Vorgang in der Praxis bei einem laufenden Scalaris-System sehr einfach:

```
(firstnode@roger.local)1> snapshot_leader ! {init_snapshot,comm:this()}.
```

Nach Versand dieser Nachricht erhält der Absender nach Beendigung des Algorithmus eine von zwei möglichen Nachrichtenformen (Erfolg oder Fehler), auf die entsprechend reagiert werden kann.

Intern versendet der Initiator nach der lokalen Initialisierung seines Zustands an alle Scalaris-Knoten einen Broadcast (`init_snapshot`) und wartet dann auf Antworten. Im Optimalfall müsste er für jeden Schlüssel 4 Replikate erhalten. Gegebenenfalls, zum Beispiel bei der Verwendung von Chord (vgl. Sektion 2.1.1) als Routing-Verfahren, müssen die Schlüssel beim Zusammenfügen des Gesamt-Snapshots normalisiert werden, da der Wert unter 4 unterschiedlichen Hashwerten in der DHT gespeichert war. Es wird immer der Wert mit der höchsten Versionsnummer gewählt. Die Entscheidung der Terminierung treffen wir mit Hilfe des *Scalaris*-Moduls `intervals`. Für jeden Knoten lässt sich dessen Zuständigkeit innerhalb des Rings als Intervall im *key space* leicht herausfinden. Damit lässt sich entscheiden, wann für jedes Intervall eine Mehrheit vorliegt. Als Rückgabe versendet der Initiator-Prozess eine Liste von Tupeln bestehend aus dem (normalisierten, gegebenenfalls gehashten) Schlüssel, dem Wert und der Versionsnummer. Konkrete Ausführungsbeispiele werden im folgenden Kapitel behandelt.

Bei der Ausführung des Algorithmus ist weiterhin auf den Hauptspeicher-Bedarf

zu achten. Der Zusammenbau des globalen Snapshots findet mit Hilfe der Erlang-Hauptspeicherdatenbank `ets` statt. Das System benötigt also mindestens genug Hauptspeicher für die Daten des lokalen Knotens plus die Vereinigung aller Datenbanken der anderen Knoten im System.

6 Auswertung

Nachdem der Algorithmus in Scalaris implementiert und getestet wurde, werden wir uns in diesem Kapitel der Auswertung der Ergebnisse widmen und einige interessante Fakten bezüglich der Implementierung im Detail und am Beispiel analysieren.

6.1 Benutzung und Beispielabläufe

In Scalaris gibt es eine Gruppe von Funktionen, die die Modifikation des Systems von außen auf Transaktions-Level erlauben. Sie sind in dem Modul `api_tx` zusammengefasst. Da unser Snapshot-Algorithmus oberhalb der Transaktionsschicht ansetzt, haben wir diesem Modul eine weitere Funktion zur Initialisierung eines Snapshots hinzugefügt: `api_tx:get_system_snapshot/0`. Per Aufruf dieser Funktion wird dem zum angesprochenen Knoten gehörenden Prozess `snapshot_leader` mitgeteilt, dass ein Snapshot zu initialisieren ist. Der Prozess leitet dann die notwendigen Schritte entsprechend Algorithmus 2 aus Sektion 5.2 ein (vgl. auch Sektion 5.4). Im einfachsten Fall sieht ein Ablauf des Algorithmus aus Sicht des Nutzers dann wie folgt aus:

```
(firstnode@roger.local)1> api_tx:write("foo",1).
{ok}
(firstnode@roger.local)2> api_tx:write("bar",2).
{ok}
(firstnode@roger.local)3> api_tx:get_system_snapshot().
[{59467880073354024310100585614816879832,1,0},
 {74047935693191174550601131226829771250,2,0}]
```

Zuerst werden auf Transaktionsebene zwei Werte unter den Schlüsseln „foo“ und „bar“ in der DHT gespeichert. Im 3. Aufruf wird ein Snapshot des Systems angefordert. Die Rückgabe erfolgt als Liste von 3-Tupeln, bestehend aus:

- Dem Schlüssel. In den hier aufgelisteten Beispielen wird in Scalaris *Chord* anstelle von *Chord#* als Routing-Schema verwendet (vgl. Sektion 2.1.1), weshalb die Schlüssel hier in gehashter Form vorliegen. Die Art des Routings hat keinen Einfluss auf den Snapshot-Algorithmus und ist in Scalaris via Konfigurationswert austauschbar.

- Dem aktuellen Wert
- Der höchsten Versionsnummer

Verfolgung eines Ablaufs

Im Folgenden haben wir eine Reihe von Ausgaben in den Code eingebaut, mit deren Hilfe wir den Ablauf des Algorithmus in einem kleinen System verfolgen können.

In diesem Testlauf auf einem System mit 4 Knoten existieren 3 Threads, die parallel Anfragen an das System stellen: Zwei dieser Threads führen sequentiell Transaktionen auf dem System aus (Lese- und Schreiboperationen mit zufälligen Werten auf den Schlüsseln A und B). Ein dritter Thread führt sequentiell Snapshot-Anfragen aus.

```
1  {{127,0,0,1},14195,<0.892.0>} snapshot:on_do_snapshot: 2
2  {{127,0,0,1},14195,<0.892.0>} snapshot:snapshot_is_done: false
3  {{127,0,0,1},14195,<0.892.0>} snapshot:snapshot_is_done db data:
4  [{254907426297438654045810222752225607209,
5  2266003264,false,0,107},
6  {294251812994918625457073604598288790897,3025286486,true,0,107}]
7  {{127,0,0,1},14195,<0.892.0>} tx_tp:update_db_or_forward incoming operation:
8  {rdht_tx_write,abort}
9  {{127,0,0,1},14195,<0.892.0>} tx_tp:update_db_or_forward db after opteration:
10 [{254907426297438654045810222752225607209,
11 2266003264,false,0,107},
12 {294251812994918625457073604598288790897,3025286486,false,0,107}]
13 {{127,0,0,1},14195,<0.892.0>} tx_tp:update_db_or_forward snapshot is done
14 {{127,0,0,1},14195,<0.892.0>} snapshot:on_local_snapshot_is_done
15 snapshot_leader got local_snapshot_done from {{127,0,0,1},14195,<0.892.0>} for
16 range [{interval,'(',243381152945604023884860858376437092446,
17 328451744675838639750704510234379145310,']'}]
```

In obigem Beispiel findet sich der Ablauf des Algorithmus aus Sicht eines einzigen Knotens, die restlichen Ausgaben wurden herausgefiltert. Jede Ausgabe bis auf die Letzte beginnt mit dem Identifikator des Prozesses, bestehend aus IP-Adresse, Port und der Erlang-Prozessnummer. In Zeile eins erhält der Knoten vom Anführer die Aufforderung, einen neuen Snapshot mit der Nummer 2 zu starten. Daraufhin wird im `snapshot`-Modul überprüft, ob der lokale Snapshot eventuell bereits fertig ist (Zeile 2). Da dies nicht der Fall ist, wird der aktuelle Zustand der Datenbank ausgegeben. Dort den Zeilen 3-6 sehen wir, dass Werte für beide Schlüssel (A und B, gehasht) existieren, dass beide bei Version 107 stehen, dass auf keinem eine Lesesperre existiert und dass auf dem Schlüssel mit dem Hash 29425 . . . eine Schreibsperre existiert, auf dem anderen nicht. Die Entscheidung „Snapshot ist noch nicht beendet“ ist also korrekt.

Daraufhin verbleibt der Knoten im normalen Betrieb und das TP-Modul erhält eine

eingehende Anfrage nach Abbruch einer (Schreib-)Transaktion (Zeile 7-8). Da in diesem Modul auch die Änderungen an der Datenbank stattfinden, kann dort - nach Anwendung der Änderungen - auch entschieden werden, ob ein laufender Snapshot beendet ist. Im hiesigen Fall wird nach dem *abort* die Schreibsperre auf dem Schlüssel entfernt (Zeilen 9-12) und das Modul entscheidet, dass der Snapshort fertig ist (Zeile 13) und benachrichtigt das lokale `snapshot`-Modul (Zeile 14). Daraufhin wird der lokale Zustand an den Initiator versendet. Dieser gibt den Zustand, zusammen mit dem Ringintervall, für das der Knoten verantwortlich ist, aus (Zeilen 15-17).

6.2 Performanz des Gesamtsystems

Verschiedene Tests haben gezeigt, dass der Einfluss des Snapshot-Algorithmus auf die Performanz des Systems relativ gering ist. In Abbildung 6.1 findet sich eine gra-

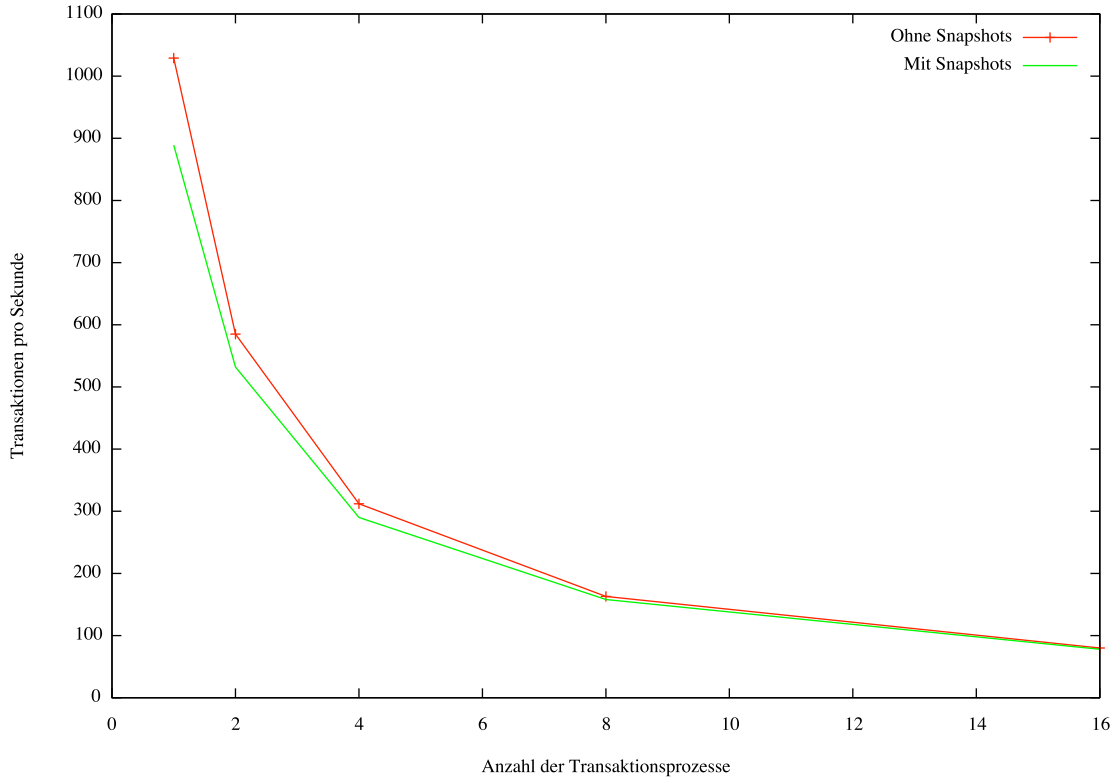

Abbildung 6.1: Transaktionsdurchsatz mit und ohne Ausführung von Snapshots

fische Darstellung eines solchen Aufbaus. Hier wurden auf einen Scalaris-Ring der Größe 4 (innerhalb einer einzigen Erlang-VM) unter Verwendung von Hilfsprozessen parallel Transaktionen ausgeführt. Jeder einzelne dieser Prozesse führte sequentiell

1000 einfache Transaktionen aus. Parallel führte ein weiterer Prozess sequentiell 200 Snapshots aus. Für den Moment blenden wir hier eventuell abgebrochene Transaktionen aus.

In diesem Fall lässt sich beobachten, dass die Ausführung von Snapshots je mehr Last auf dem System existiert immer weniger Einfluss auf den Durchsatz hat. Während die Abweichung bei einem Transaktionsprozess noch etwa 14% beträgt, ist sie bei einem mit 16 Prozessen deutlich überlasteten System[1] nur noch bei $2,5\%$.

In der Praxis sollte die Herstellung eines Snapshots vom Administrator eines Scalaris-Systems zwar periodisch ausgeführt werden, jedoch keinesfalls in einer so hohen Frequenz, dass das System ernsthaft beeinträchtigt wird. Je größer die lokalen Datenbanken der einzelnen Knoten werden, desto größer wird auch der Einfluss einer Snapshot-Ausführung auf die Performanz des Systems, da der Versand der lokalen Zustände je nach Bandbreite immer länger dauert und somit wichtigere Nachrichten (z.B. des Transaktionssystems) sehr langsam werden können. Theoretisch kann dies fälschlicherweise sogar zur Feststellung eines Ausfalls führen, da die Antwort auf Nachrichten von Ausfalldetektoren möglicherweise verzögert wird. Es ist daher ratsam, das Scalaris-System mit zunehmender Datenmenge entsprechend zu skalieren.

Die Beeinträchtigung durch die eigentliche Ausführung des Snapshot-Algorithmus auf einem Knoten ist hingegen relativ begrenzt. Zur Initialisierung muss einmalig eine Nachricht verarbeitet und lokaler Zustand angepasst werden. Daraufhin erfolgt, sollte der Snapshot nicht bei Initialisierung bereits beendet sein, für jedes eingehende *commit* oder *abort* ein Test, ob die zugehörige Transaktion zur vorigen Snapshot-Nummer gehört (ein Ganzzahl-Vergleich) und, falls dem so ist, müssen daraus resultierende Schreiboperationen im Hauptspeicher doppelt ausgeführt werden. Am Ende jeder dieser Ausführungen steht ein weiterer Ganzzahl-Vergleich, um zu entscheiden, ob der Snapshot fertig ist (vgl. Sektion 5.4).

6.3 Abbrüche von Transaktionen

Den größten Eingriff in das existierende Scalaris-System nahmen wir in der Validationsphase vor. Um vorzubeugen, dass Transaktionen nicht eindeutig einer Snapshot-Instanz zugeordnet werden können, legten wir fest, dass diejenigen, bei denen während der Erstellung des Transaktionslog eine andere Snapshot-Nummer als die gerade aktuelle vorlag, abgebrochen werden. Das Ganze passiert unabhängig davon, ob der Snapshot-Algorithmus auf dem Knoten gerade läuft.

Um uns einen Eindruck zu verschaffen, wie viele Transaktionen von dieser Ent-

[1] Laptop mit 1,8GHz Core 2 Duo Zweikern-Prozessor und 4GB Hauptspeicher

scheidung betroffen sind, wählten wir folgenden Testaufbau: Wir erstellten einen Scalaris-Ring der Größe 4 in einer lokalen Erlang VM, und erstellten dann eine variable Anzahl von Threads, in denen jeweils 1000 sequentielle, sehr einfache Transaktionen ausgeführt wurden. Bei 64 Threads wurde also versucht, 64000 Transaktionen auszuführen, davon je 64 „parallel“ (wir befinden uns nach wie vor auf einem Laptop mit Zweikern-Prozessor). Ein weiterer Thread forderte parallel dazu insgesamt 300 Snapshots vom System an. Abbildung 6.2 illustriert die Ergebnisse. Jeder der

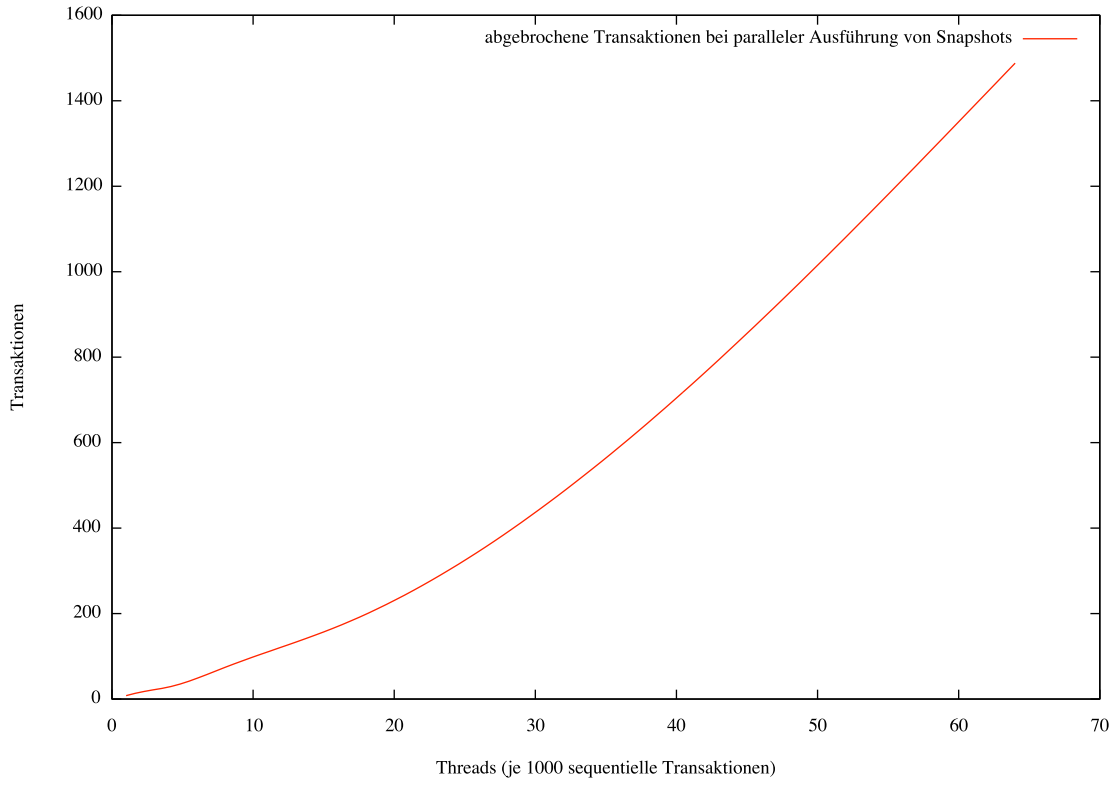

Abbildung 6.2: Durch Snapshot-Nummer-Vergleich abgebrochene Transaktionen

Transaktionsthreads wählte den Schlüssel zufällig, d.h. Konflikte sind ausgeschlossen. In mehreren zusätzlichen Testläufen wurde ohne die gleichzeitige Ausführung von Snapshots auch bei 64 Transaktionsthreads keine einzige Transaktion abgebrochen.

An Hand der Daten lässt sich beobachten, dass bei steigender Last immer mehr Transaktionen unter das angesprochene Kriterium fallen. Während es bei einem und zwei Prozessen noch $0,8\%$ sind, sind es bei 16 schon $1,0\%$ und bei 64 dann $2,5\%$. Dies lässt sich mit der bei Überlastung des Systems länger werdenden Zeitspanne zwischen Client-Arbeitsphase (Zuammenbau des Transaktionslogs) und Validation erklären. In Abbildung 6.1 ist der Grundaufbau vergleichbar und schon bei 16 Transaktionsthreads liegt der Durchsatz nur noch bei 80 Transaktionen pro Sekunde. An

dieser Stelle müssen wir also erneut die dringende Empfehlung aussprechen, das Scalaris-System entsprechend den Anforderungen zu skalieren. In einem überlasteten System führt die häufige Ausführung von Snapshots zu unerwünschten, jedoch nicht unerwarteten Effekten.

6.4 Privatsphäre und Sicherheit

Die Einführung eines Verfahrens zum Erstellen eines globalen Zustands eines solchen Systems hat den Nebeneffekt, dass nun eine Sicht auf die in der DHT gespeicherten Daten geschaffen wurde, die vorher nicht existierte. Bisher war es ohne unverhältnismäßigen Aufwand nicht möglich, eine Übersicht über alle im System gespeicherten Daten zu erhalten. Im Falle der Verwendung von *Chord#* als Routing-Schema sind sogar die Schlüssel im Klartext (ohne vorige Anwendung einer Hashfunktion) vorhanden. Bei *Chord* könnte durch die Verwendung einer sicheren Hashfunktion wenigstens die Zuordnung der Daten zu ihren ursprünglichen Schlüsseln verhindert werden. In einer Hosting-Umgebung, bei der sich viele Nutzer ein Scalaris-System teilen, wäre dann die durch die Anfrage-Schnittstelle gegebene Isolation (beispielsweise könnten alle vom Nutzer abgelegten Schlüssel mit einem Präfix - z.B. der Nutzer-ID - versehen werden) der Daten nicht mehr gewährleistet.

Weiterhin wären verschiedene Angriffsszenarien denkbar. Einerseits könnte sich ein bösartiger Knoten als Initiator eines Snapshots ausgeben, der gar nicht die Berechtigung dazu hat, bzw. der eventuell nicht einmal Teil des Scalaris-Rings sein muss. Andererseits könnte ein bösartiger Knoten gefälschte Daten an den Initiator liefern und damit leicht den Gesamtzustand verfälschen. Auch Man-in-the-Middle-Attacken auf einzelne Knoten oder den Initiator des Snapshots sind denkbar.

6.5 Mögliche Weiterentwicklungen

Der größte Entwicklungsbedarf bis zur endgültigen praktischen Einsatzbereitschaft des Verfahrens liegt im Bereich Privatsphäre und Sicherheit (vgl. Sektion 6.4). Die dort vorliegenden Schwächen des Verfahrens sind nicht nur theoretisch sondern auch praktisch sehr leicht auszunutzen und werden potentiellen Anwendern Probleme bereiten.

An vorderster Stelle steht wohl ein Verfahren zur Authentifizierung und Autorisierung von Knoten, die überhaupt als Snapshot-Initiator fungieren dürfen. Denkbar wäre hier bei einem von einem Unternehmen betriebenen System zum Beispiel ei-

ne klassische Public Key Infrastructure (PKI), bei der von einer zentralen Instanz Zertifikate für einzelne Knoten ausgestellt werden, die dann von Kommunikationspartnern überprüft werden können. Bei einem kollaborativ betriebenen System ist die Festlegung einer solchen Instanz meist problematisch, weshalb alternativ ein *Web Of Trust* zum Einsatz kommen könnte. Beide Modelle sind erprobt und in laufend aktualisierten Open-Source Softwarebibliotheken implementiert, so dass eine Umsetzung möglich wäre.

Ein fast ebenso wichtiger Punkt ist die Verschlüsselung der Punkt-zu-Punkt Kommunikation der Knoten innerhalb des Snapshot-Protokolls zur Verhinderung von Man-in-the-Middle-Angriffen und Abhöraktionen. Auch hier kann auf bekannte kryptografische Verfahren mit existierenden Bibliotheken wie TLS oder SSL zurückgegriffen werden. Das Vorhandensein einer PKI oder eines Web Of Trust ist dann allerdings Voraussetzung, da Kommunikationspartner vor Aufbau einer verschlüsselten Verbindung immer erst ihre Authentizität beweisen müssen.

Aus Sicht des Nutzers der DHT, der sich nun damit konfrontiert sieht, dass der Betreiber ohne Weiteres eine globale Sicht auf die Daten erhalten kann, wäre es zudem wünschenswert, wenn es - möglicherweise zusätzlich zu den existierenden Schnittstellen - eine Art sichere Transaktions-API gäbe, bei der der Nutzer darauf vertrauen kann, dass niemand die Daten zu Gesicht bekommt. Dafür wären unserer Meinung nach 3 Schritte notwendig

- Verschlüsselung der Kommunikation von Client zum (authentifizierten) Knoten
- Anwendung sicherer Verfahren zum Hashen der Schlüssel
- transparente Ver- und Entschlüsselung der eigentlichen Daten

Alle 3 Schritte könnten ebenfalls auf bekannten Verfahren und existierenden und erprobten Softwarebibliotheken basieren.

7 Rückblick

Die zentrale Problemstellung bei der Anfertigung dieser Arbeit war das Fehlen eines Verfahrens zur Aufnahme eines konsistenten Zustandes in einer skalierbaren DHT-Implementierung mit Unterstützung für Transaktionen namens *Scalaris*. Wir haben uns also zuerst den theoretischen Konstrukten gewidmet, die dem Aufbau dieser Software zu Grunde liegen und eine Einführung in strukturierte Overlay-Netze, DHTs, *Chord* und *Chord#* erarbeitet. Noch im selben Kapitel gaben wir einen Überblick über die klassischen Ereignismodelle, die in der Literatur zu verteilten Systemen verwendet werden. Insbesondere definierten wir, was ein konsistenter Schnitt ist.

Ausgehend von dieser Definition widmeten sich Wissenschaftler in den letzten fast 30 Jahren der Frage nach einem Algorithmus, der einen Zustand des Systems niederschreibt, der einem solchen konsistenten Schnitt entspricht. Wir gaben in Kapitel 3 einen ausführlichen Überblick über die existierenden Protokolle.

Die Analyse dieser Protokolle hinsichtlich der Anwendbarkeit auf das hier vorliegende Softwareprojekt und dessen Struktur führte dann zu einer knappen aber eindeutigen Entscheidung für die Implementierung eines dieser Protokolle, der Mattern'schen Adaption des Algorithmus von Lai und Yang. Ausgehend von den Grundlagen dieses Algorithmus warfen wir dann einen detaillierten Blick auf das Transaktionssystem von Scalaris, welches im Wesentlichen den lokalen Zustand eines Knotens in unserem System bestimmt, und kamen zu dem Schluss, dass sich die Implementierung noch deutlich vereinfachen und dass sich das Ergebnis leichter handhaben ließe, wenn einige sehr spezielle Anpassungen am Protokoll vorgenommen werden würden. Diese Anpassungen mündeten im Entwurf unseres eigentlichen Algorithmus in Kapitel 5.

Nachdem die Implementierung des Verfahrens am existierenden Software-Projekt und die Beschreibung ausgewählter Details abgeschlossen war, widmeten wir uns der Auswertung unserer Arbeit. Hier stellten wir fest, dass die korrekte Skalierung (Anzahl der Knoten im Vergleich zur Auslastung des Systems und dessen Datenmenge) des Scalaris-Rings entscheidend für gute Performanz des Systems in Gegenwart des laufenden Snapshot-Algorithmus ist. Zudem stellten wir fest, dass unsere Implementierung unter einer Reihe von Problemen im Zusammenhang mit der Privatsphäre der Nutzer und der allgemeinen Sicherheit der Daten leidet und stellten hier einige Lösungsansätze vor, mit denen sich weiterführende Arbeit beschäftigen könnte.

Alles in Allem ist das Ergebnis dieser Arbeit sehr zufriedenstellend und stellt eine große Verbesserung für die Nutzer der Software dar. Eine solide theoretische Grundlage kombiniert mit einigen sehr pragmatischen Anpassungen mündeten in einer Implementierung von überschaubarer Komplexität. Damit ist eines der größten Defizite von *Scalaris*, welches in der Praxis eine Reihe von Nutzern an einem Umstieg gehindert hat, endgültig beseitigt.

Literaturverzeichnis

[1] Alima, L.O. ; El-ansary, S. ; Br, P. ; Haridi, S.: DKS (N, k, f): A family of low communication, scalable and fault-tolerant infrastructures for P2P applications. In: *In 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2003

[2] Armstrong, J.: Programming Erlang: Software for a Concurrent World. (2007)

[3] Chandy, Kanianthra M. ; Lamport, Leslie: Distributed snapshots: determining global states of distributed systems. In: *ACM Trans. Comput. Syst.* 3 (1985), Nr. 1, S. 63–75. – ISSN 0734–2071

[4] Chang, E.J.H.: Echo algorithms: Depth parallel operations on general graphs. In: *IEEE Trans. Software Eng.* 8 (1982), Nr. 4, S. 391–401

[5] Critchlow, Carol ; Taylor, Kimberly: The inhibition spectrum and the achievement of causal consistency. In: *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing.* New York, NY, USA : ACM, 1990. – ISBN 0–89791–404–X, S. 31–42

[6] De Prisco, R. ; Lampson, B. ; Lynch, N.: Revisiting the Paxos algorithm. In: *Distributed Algorithms* (1997), S. 111–125

[7] Gallager, R. G. ; Humblet, P. A. ; Spira, P. M.: A Distributed Algorithm for Minimum-Weight Spanning Trees. In: *ACM Trans. Program. Lang. Syst.* 5 (1983), Nr. 1, S. 66–77. – ISSN 0164–0925

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

[9] Gray, James N.: Notes on data base operating systems. In: *Operating Systems* (1978), S. 393–481

[10] Gray, James N. ; Lamport, Leslie: Consensus on transaction commit. In:

ACM Trans. Database Syst. 31 (2006), Nr. 1, S. 133–160

[11] Guerraoui, R. ; Rodrigues, L.: *Introduction to reliable distributed programming.* Springer-Verlag New York Inc, 2006

[12] Hélary, J.M.: Observing Global States of Asynchronous Distributed Applications. In: *Proceedings of the 3rd International Workshop on Distributed Algorithms* Springer-Verlag, 1989, S. 135

[13] Koo, Richard ; Toueg, Sam: Checkpointing and rollback-recovery for distributed systems. In: *ACM '86: Proceedings of 1986 ACM Fall joint computer conference.* Los Alamitos, CA, USA : IEEE Computer Society Press, 1986. – ISBN 0–8186–4743–4, S. 1150–1158

[14] Kshemkalyani, AD ; Raynal, M. ; Singhal, M.: An introduction to snapshot algorithms in distributed computing. In: *Distributed systems engineering* 2 (1995), S. 224–233

[15] Lai, Ten H. ; Yang, Tao H.: On distributed snapshots. In: *Inf. Process. Lett.* 25 (1987), Nr. 3, S. 153–158. – ISSN 0020–0190

[16] Lamport, L.: The part-time parliament. In: *ACM Transactions on Computer Systems (TOCS)* 16 (1998), Nr. 2, S. 169

[17] Lamport, Leslie: Time, clocks, and the ordering of events in a distributed system. In: *Commun. ACM* 21 (1978), Nr. 7, S. 558–565. – ISSN 0001–0782

[18] Mattern, Friedemann: Algorithms for distributed termination detection. In: *Distributed Computing* 2 (1987), S. 161–175

[19] Mattern, Friedemann: Experience with a New Distributed Termination Detection Algorithm. In: *Distributed algorithms proceedings Lecture notes in computer science* (1987), S. 127

[20] Mattern, Friedemann: Virtual time and global states of distributed systems. In: *Parallel and Distributed Algorithms* (1989), S. 215–226

[21] Mattern, Friedemann: Efficient algorithms for distributed snapshots and global virtual time approximation. In: *Journal of Parallel and Distributed Computing* 18 (1993), Nr. 4

[22] Reinefeld, Alexander ; Schintke, Florian ; Schütt, Thorsten ; Haridi, S.: A Scalable, Transactional Data Store for Future Internet Services. In: *Towards the Future Internet* (2009), S. 148

[23] Schintke, F. ; Reinefeld, A. ; Haridi, S. ; Schütt, T.: Enhanced Paxos Commit for Transactions on DHTs.

[24] Schütt, Thorsten ; Moser, M. ; Plantikow, S. ; Schintke, Florian ; Reinefeld, Alexander: A Transactional Scalable Distributed Data Store: Wikipedia on a DHT. In: *Proc. 1st IEEE International Scalable Computing Challenge, Lyon*, 2008

[25] Schütt, Thorsten ; Schintke, Floarian ; Reinefeld, Alexander: Structured overlay without consistent hashing: Empirical results. In: *Sixth IEEE International Symposium on Cluster Computing and the Grid Workshops, 2006* Bd. 2, 2006

[26] Schütt, Thorsten ; Schintke, Florian ; Kruber, Nico: Scalaris: Users and Developers Guide, Version 0.4.0. (2012)

[27] Schütt, Thorsten ; Schintke, Florian ; Reinefeld, Alexander: Chord#: Structured Overlay Network for Non-Uniform Load-Distribution.

[28] Schütt, Thorsten ; Schintke, Florian ; Reinefeld, Alexander: Scalaris: Reliable Transactional P2P Key/Value Store - Web 2.0 Hosting with Erlang and Java. In: *Proc. 7th ACM SIGPLAN Erlang Workshop, Victoria*, 2008

[29] Shah, A. ; Toueg, S.: Distributed Snapshots in spite of Failures. (1984)

[30] Skeen, M.D.: *Crash recovery in a distributed database system*, University of California, Berkeley, Diss., 1982

[31] Spezialetti, M. ; Kearns, P.: Efficient distributed snapshots. In: *Proceedings of the International Conference on Distributed Computing Systems*, 1986, S. 382–388

[32] Stoica, Ion ; Morris, Robert ; Karger, David ; Kaashoek, M. F. ; Balakrishnan, Hari: Chord: A scalable peer-to-peer lookup service for internet applications. In: *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications.* New York, NY, USA : ACM, 2001. – ISBN 1–58113–411–8, S. 149–160

[33] Taylor, Kimberly: The Role of Inhibition in Asynchronous Consistent-Cut Protocols. Ithaca, NY, USA : Cornell University, 1989. – Forschungsbericht

[34] Tel, G.: *Topics in distributed algorithms.* Cambridge Univ Press, 1991

[35] Venkatesan, S.: Message-optimal incremental snapshots. In: *Proceedings of*

the International Conference on Distributed Computing Systems, 1989, S. 53–60

Abbildungsverzeichnis

2.1 Zeitdiagramm eines Scalaris-Commits [23] 11
2.2 Beispiel für eine Ereignisabfolge in einem verteilten System [20] . . . 15
2.3 Beispiel für einen Schnitt [20] 16
2.4 Beispiel für einen inkonsistenten Schnitt [20] 17
2.5 Der Gummiband-Test [20] 18

3.1 Das Prinzip des Algorithmus von Mattern [21] 30

5.1 Transaktionen, an denen ein Knoten als TP teilnimmt als Zeitdiagramm 58
5.2 Zustandsdiagramm S3-Teilnehmer 63
5.3 Zustandsdiagramm S3-Initiator im fehlerfreien Fall 64

6.1 Transaktionsdurchsatz mit und ohne Ausführung von Snapshots . . . 70
6.2 Durch Snapshot-Nummer-Vergleich abgebrochene Transaktionen . . . 72

Tabellenverzeichnis

4.1 Zu Grunde liegende Systemmodelle 43
4.2 Anforderungen an Nachrichtenkanäle 44
4.3 Anforderungen an die Netzwerktopologie 45
4.4 Die Algorithmen in dynamischen Umgebungen 46
4.5 Kommunikationskomplexität der Algorithmen 47
4.6 Hemmung von Systemereignissen 49
4.7 Zusammenfassung der Eigenschaften der Algorithmen 53