Fehlertolerante Replikation von Diensten mit schwacher Konsistenz mittels selbststabilisierender verteilter Algorithmen

Diplomarbeit
von
Felix Christoph Gärtner

18. Dezember 1997

Technische Universität Darmstadt
Fachbereich Informatik
Fachgebiet Betriebssysteme
Betreuer: Dr. Henning Pagnia
Prüfer: Prof. Dr. Peter Kammerer

DA-BS-1997-06

Zusammenfassung

In dieser Arbeit wird eine Familie von verteilten Algorithmen für schwach-konsistente Replikation entwickelt, die ein hohes Maß an Fehlertoleranz aufweisen. Die guten Eigenschaften der Algorithmen werden durch die konsequente Berücksichtigung eines Konzeptes erreicht, welches unter dem Namen "Selbststabilisierung" 1974 durch Dijkstra in die Informatik eingeführt wurde. Das zugrundeliegende Modell basiert auf asynchroner Kommunikation über Nachrichten und verlangt als Voraussetzung lediglich, daß die einzelnen Knoten mittels eindeutiger Namen unterschieden werden können und alle die Größe des Netzwerkes kennen. Durch die Selbststabilisierungseigenschaft sind die Algorithmen in der Lage, sowohl *jede* Form vorübergehender Fehler zu tolerieren, als auch eine begrenzte Anzahl permanenter Fehler (solange keine Netzpartitionen auftreten). Der Nachteil an dieser Form von Fehlertoleranz ist, daß fehlerhaftes Verhalten nicht maskiert wird und die Knoten darum ein begrenztes Maß an fehlerhaftem Verhalten ihrer Umgebung ertragen müssen. Die entwickelten Algorithmen werden mit bereits existierenden Verfahren für schwach-konsistente Replikation in verteilten Systemen verglichen und ihre praktische Einsetzbarkeit wird diskutiert.

Abstract

In this thesis a family of novel distributed algorithms aimed at performing weak-consistency replication in a highly fault-tolerant manner is presented. The fault-tolerance is achieved by following the paradigm of self-stabilization which was introduced by Dijkstra in 1974. The algorithms are fully distributed in the sense that all nodes run the same algorithm in a message-passing environment (only unique node identities are required) and all nodes have to know the approximate size of the network. Because the algorithms are self-stabilizing, they can tolerate any form of transient failures and also a limited number of permanent failures (as long as no network partitions occur). The drawback from using the paradigm is that the fault-tolerance is non-masking and that network nodes can thus be subject of a limited amount of errorneous behaviour once failures occur. Theoretical analysis and simulation results suggest that these algorithms perform well, compared to existing solutions to the problem of weak-consistency replication.

für Uli, meine Eltern und Lina

Inhaltsverzeichnis

1 Einführung

1.1 Allgemeine Vorbemerkungen

Angeblich besteht ein umgekehrt proportionaler Zusammenhang zwischen der Anzahl der Formeln in einer Arbeit und der Größe der Leserschaft [82, S. 7]. Bei den gut 150 Formeln auf den folgenden Seiten war eine breite Leserschaft offensichtlich auch nicht das Ziel, obwohl der Schnitt von einer Formel pro Seite im Vergleich zu anderen einschlägigen Publikationen [40, 83, 95] noch eher gering ist. Man fragt aber auch zu Recht, ob eine Diplomarbeit wie diese überhaupt eine Leserschaft finden kann, die über die akademischen Betreuer hinausgeht — nach dem Motto: "Sollen die Leute, die sich nicht in der Informatik auskennen, doch andere Texte lesen!" Trotzdem hatte ich von vornherein den Wunsch, eine Arbeit zu schreiben, die auch für Nicht-Informatiker verständlich sein sollte. Allerdings mußte ich beim Schreiben und vor allem bei der Überarbeitung vieler Textteile einsehen, daß dieser Wunsch erstens wegen der Komplexität des behandelten Themas und der Kürze der Zeit leider nicht in Erfüllung gehen konnte und zweitens wegen des wissenschaftlichen Anspruchs dieser Arbeit gar nicht in Erfüllung gehen durfte.

Nach wie vor halte ich es für eine der wichtigsten Aufgaben der Wissenschaft (und gerade der Informatik), ihre Gegenstände und Sachverhalte in allgemeinverständlicher Form auszudrücken und sie so auch den "Nicht-Experten" zugänglich zu machen. Dies ist umso wichtiger, je stärker die Gesellschaft durch die Technik bestimmt und abhängig gemacht wird. Darum habe ich mich bemüht, die zum Teil doch sehr formalen und theoretischen Konzepte so zu schildern, daß sie ein Informatik-Studierenden nach dem Grundstudium nachvollziehen können sollte (Kenntnisse: Beweise, formale Systeme, formale Logik). Die einzelnen Kapitel enthalten zudem anfangs jeweils kurze Absätze, die den folgenden Stoff auch für

Nicht-Informatiker verständlich machen sollen.

1.2 Thema dieser Arbeit

Heutige Einzelplatzrechner bestehen aus einer für den Laien unüberschaubaren Vielzahl von Einzelteilen, die größtenteils selbst wieder ein hochkomplexes Geflecht von Leiterbahnen und Schaltkreisen umfassen. Alle diese Bauteile sind mögliche Fehlerquellen, die zum Ausfall des gesamten Systems führen können [45, S. 57], und beinahe jeder, der schon mit Computern in Berührung gekommen ist, wird bestätigen, daß diese Komponenten auch tatsächlich ab und zu ausfallen und den Rechner zeitweise unbenutzbar machen (von der Tatsache, daß die eingesetzten Programme viel öfter fehlerhaft sind, ganz zu schweigen).

Daß Computerbauteile irgendwann ausfallen, ist nicht so sehr ein technisches Problem sondern steht im Zusammenhang mit der Haltbarkeit und Belastbarkeit physikalischer Materialien, die in der Realität eben doch nur begrenzt ist. Menschen, die Computersysteme entwickeln, müssen daher lernen, mit diesen Voraussetzungen umzugehen. Das bedeutet nicht, daß man diese Fehler als "notwendiges Übel" einfach hinnehmen muß. Da Computer zunehmend in sicherheitskritischen Bereichen (zum Beispiel Flugsicherung, Kraftwerksteuerung) eingesetzt werden und das Wohlergehen der Gesellschaft immer stärker von korrekt funktionierenden verteilten Rechnersystemen abhängt, wäre gerade im Computerbereich ein "Fehler-Fatalismus" verhängnisvoll.

Es gibt daher in der modernen Welt einen zunehmenden Bedarf an Rechnersystemen, die im Falle interner Fehler entweder ein wohldefiniertes Fehlerverhalten aufweisen (und beispielsweise vor dem Ausfall dafür sorgen, daß keine Daten verlorengehen) oder trotz interner Fehler weiterhin nach außen die von ihnen verlangten Leistungen erbringen. Gewöhnlich bezeichnet man derartig gebaute Systeme als *fehlertolerant* [45, S. 57]. Die Informatik beschäftigt sich in immer stärkerem Maße mit den Aspekten der Sicherheit und Zuverlässigkeit von Rechnernetzen [26]; diese Arbeit ist ein Teil davon.

In dieser Arbeit will ich ein ausgewähltes praktisches Problem betrachten und fehlertolerante Lösungen zu dessen Realisierung erarbeiten. Das konkrete Problem, um das es hier geht, ist *Replikation in verteilten Systemen*. Replikation meint, daß mehrere identische Kopien ein und derselben Sache zur Verfügung gestellt werden, um eine bestimmte Leistung zu erbringen. Das in diesem Zusammenhang oft benutzte Standardbeispiel ist ein Telefonbuch: Ohne die Verbreitung tausender Exemplare ein und derselben Ausgabe könnte eine Telefongesellschaft die Auskunftswünsche ihrer Kunden nicht befriedigen. In der Sprache der Informatik ist ein Telefonbuch ein "replizierter Auskunftsdienst." Weitere wichtige Anwendungsgebiete von Replikation sind alle Formen der verteilten Datenbanken.

Es gibt viele verschiedene Methoden, um Fehlertoleranz in verteilten Systemen zu realisieren [45] (vergleiche Abbildung 1.1). Ein bisher noch selten anzutreffendes Konzept heißt *Selbststabilisierung*. Informal gesprochen ist ein System selbststabilisierend, wenn es nach vorübergehenden Fehlern wieder ohne äußeres Zutun beginnt, korrekt zu funktionieren. Nach dem Auftreten eines Fehlers kann ein Benutzer vorübergehend auch nach außen hin ein gewisses Maß an Fehlverhalten beobachten, doch kann er darauf vertrauen, daß sich ein korrektes Systemverhalten irgendwann wieder einstellt, ohne daß er weiter eingreifen muß.

Abbildung 1.1: Fehlertoleranz durch Redundanz [137].

1.3 Aufbau der Arbeit

Die Arbeit ist wie folgt aufgebaut: Zunächst werden in einem Kapitel über Modelle für verteilte Algorithmen die formalen Grundlagen und Beschreibungsmöglichkeiten für die in dieser Arbeit auftretenden Probleme dargelegt. Anschließend befassen sich getrennte Kapitel zum einen mit dem Anwendungsgebiet, also Replikation, und zum anderen mit der Methode, also Selbststabilisierung. Im darauffolgenden Hauptteil dieser Arbeit werden beide Bereiche miteinander verschmolzen mit dem Ziel, fehlertolerante Replikationsverfahren zu entwickeln. Im abschließenden Kapitel befasse ich mit ich praktischen Erwägungen zur Realisierbarkeit der Verfahren, diskutiere Simulationsergebnisse und nenne Ansätze, wie die hier begonnene Arbeit weiter fortgeführt werden kann.

Die Kapitel beginnen jeweils mit einer eigenen “umgangssprachlichen”

Kurzzusammenfassung, und sind weiter durch eine etwas ausführlichere Einleitung und einen abschließenden Ausblick eingerahmt.

Natürlich kann diese Arbeit keine umfassende Darstellung eines der behandelten Gebiete erbringen, obwohl gerade bei meiner Beschäftigung mit der Selbststabilisierung die Tendenz zu einem solchen Unterfangen durchaus vorhanden war. Die Anzahl der Seiten, die daraus resultierte, ist teilweise größeren Kürzungen unterzogen worden; gegen die möglichen Risiken und Nebenwirkungen der trotzdem entstandenen hohen Seitenzahl empfehle ich kursorisches Lesen.

1.4 Anmerkung zu den Beweisen

In diese Arbeit werden an einigen Stellen Beweise der Korrektheit von Algorithmen geführt. Der Stil, in dem die Beweise zu Papier gebracht worden sind, folgt der Art und Weise, wie sie in den gängigen Fachzeitschriften Verwendung findet (*journal mathematics*). Die Erfahrung zeigt, daß solche Beweise sehr unzuverlässig sind und daß so schon oft die Korrektheit von Algorithmen "nachgewiesen" wurde, nur um später festzustellen, daß sie doch nicht funktionieren [16].

Die Lösung für dieses Problem ist die Verwendung eines formalen Kalküls, der es erlaubt, mit genau definierten Schlußregeln auf die Erfüllung der Spezifikation zu schließen [97]. Solche Kalküle gibt es durchaus schon für verteilte Algorithmen [104, S. 1167ff]. Meine Erfahrung zeigt, daß man eine große Genauigkeit nur mit einer ebensogroßen Unübersichtlichkeit erkaufen kann. Deshalb möchte ich in dieser Arbeit keines dieser Kalküle verwenden. Mir geht es darum, den Leser von der Korrektheit der aufgestellten Sätze zu überzeugen und gleichzeitig dabei so genau zu sein, daß ein Beweis mit geringem Zusatzaufwand in die formale Sprache eines Verifikationskalküls übersetzt werden könnte. Dennoch sollte man (nicht nur in dieser Arbeit) solchen Beweisen generell skeptisch gegenüberstehen [103, S. 347].

1.5 Danksagungen

Großer Dank geht an meinen Betreuer Dr. Henning Pagnia, der meine hohen Erwartungen in jeder Art und Weise erfüllt und sogar noch übertroffen hat. Seine Bereitschaft, sich auf "mein" Thema so einzulassen, wie er es getan hat, seine Geduld, die vielen von mir fabrizierten Seiten zu lesen, und seine entscheidenden Hilfestellungen in kritischen Phasen dieser Arbeit sind besonders zu würdigen.

Weiterhin muß ich mich bei Norman Ramsey für seine hervorragende Arbeit am Programmierwerkzeug `noweb` [96, 121] bedanken, welches mir seit drei Jahren unschätzbare Dienste geleistet hat und durch das Programmieren (fast) zur Freude wurde. Dank geht auch an Friedemann Mattern dafür, daß er mich in seiner unnachahmlichen Weise mit dem überaus spannenden Gebiet der verteilten Algorithmen bekannt gemacht hat.

Die Benutzer und Systemverwalter des Instituts für Systemarchitektur und der Rechnerbetriebsgruppe haben teilweise während der in Kapitel 6 besprochenen Simulationsberechnungen unter der dadurch verursachten hohen Systemlast gelitten. Ich muß mich für ihre immer sachlichen Klagen und (unwissende) Geduld bedanken.

Thorsten Prante hat dankenswerterweise Teile dieser Arbeit gelesen und wertvolle Hinweise zu ihrer Verbesserung gegeben. Außerdem will ich Mussie Fessehatzion dankend erwähen, der teilweise parallel im angrenzenden Zimmer für sein eigenes Diplom arbeitete, dessen Computer für gelegentliche Benutzung immer zur Verfügung stand und ohne den die vielen Live-Übertragungen der Bundesliga- und Europapokalspiele nur halb so spannend gewesen wären.

Da diese Diplomarbeit auch den Abschluß meines Studiums bildet, will ich an dieser Stelle noch drei Personen nennen, die in besonders positiver Weise auf den Verlauf meiner Studien eingewirkt haben: Zum einen sind dies meine Eltern, Kurt und Traute Gärtner, die mich während meiner gesamten Studienzeit in jeder Weise unterstützt haben und mir immer ein selbstbestimmtes und unabhängiges Studium ermöglichten. Zum anderen ist das Uli Freiling, die mir als meine schärfste "nicht-informatische" Kritikerin gezeigt hat, daß zu einem erfolgreichen Studium im allgemeinen und zu einer guten Diplomarbeit im besonderen mehr gehört als bloßer technischer Sachverstand und wissenschaftliche Attitüde.

2

Modelle verteilter Algorithmen

Inhaltsübersicht

Kurzfassung

Wenn mehrere autonome Rechner kooperieren, um gemeinschaftlich ein Problem zu lösen, spricht man von einem *verteilten Algorithmus*. Naturgemäß ist ein Szenario, in dem viele Personen agieren, um einiges komplexer als ein Szenario, in dem es nur einen Akteur gibt: Beispielsweise ist es bei hinreichender Größe des Systems nur noch sehr schwer möglich, einen Überblick über alle Akteure zu einem bestimmten Zeitpunkt zu bekommen, wie er zum Beispiel bei einer Volkszählung benötigt wird. Darum ist der Entwurf verteilter Algorithmen so schwierig. Und außerdem will man natürlich auch von verteilten Algorithmen wissen, ob sie korrekt funktionieren, also auch wirklich das tun, was sie sollen.

Um diese Probleme wenigstens einigermaßen in den Griff zu bekommen, behilft man sich mit Formalismen: Um bei der Beschreibung und dem Korrektheitsnachweis verteilter Algorithmen möglichst wenig Fehler zu machen, legt man sehr genau fest (möglichst auf Mengenniveau), was man unter Begriffen wie “Algorithmus”, “Berechnung” oder “Eigenschaft eines Algorithmus” versteht. Genau dies geschieht in diesem Kapitel.

2.1 Einführung

Dieses Kapitel soll einen Überblick über die formalen Modelle geben, die zur Beschreibung verteilter Systeme und Algorithmen im Laufe dieser Arbeit verwendet werden.

Zunächst will ich kurz darstellen, wodurch sich verteilte Algorithmen von ihren nicht-verteilten Pendants unterscheiden. Dabei wird erklärt, warum gerade in diesem Bereich ein genaues formales Modell unbedingt notwendig ist. Anschließend will ich ein paar gebräuchliche Modelle skizzieren und damit die Bandbreite der Beschreibungsmöglichkeiten aufzeigen und abstecken.

Probleme der Terminologie. In der englischsprachigen Originalliteratur gibt es bezüglich der Beschreibungsmethoden verteilter Algorithmen bisher noch keine einheitliche Terminologie. Im Deutschen ist die Literatur zu diesem Thema zudem so spärlich, daß es oft unmöglich erscheint, überhaupt eine (einzige) Übersetzung eines bestimmten Begriffes zu finden. Ich werde darum bei der Benutzung der Begriffe häufig auf eigene Übersetzungen angewiesen sein und immer den Bezug zum konkreten englischsprachigen Begriff deutlich machen.

2.2 Verteilte Algorithmen

Grundprobleme verteilter Algorithmen. Verteilte Systeme sind dadurch gekennzeichnet, daß viele unabhängige Einheiten (Prozesse, Prozessoren, Computer) existieren, deren einzige Kommunikationsmöglichkeit darin besteht, Informationen mittels Nachrichten über ein zwischen ihnen bestehendes Kommunikationsnetzwerk auszutauschen. Im Gegensatz zu eng gekoppelten Parallelrechnern oder herkömmlichen Ein- oder Mehrprozessorsystemen entstehen für Algorithmen in verteilten Systemen dadurch drei wesentliche Probleme:

- Das Fehlen einer gemeinsamen Zeit.
  Eine gemeinsame "globale Zeit," wie man sie aus dem täglichen Leben kennt, ist in verteilten Systemen nicht in beliebiger Genauigkeit realisierbar. Bei einer hinreichenden Größe des Netzwerkes ist es unmöglich, die Prozesse durch eine einzelne Uhr mit einer gemeinsamen Zeit zu versorgen. Man muß davon ausgehen, daß die Uhren der einzelnen Prozesse nicht beliebig genau synchronisiert werden können und daß nur sehr ungenaue Aussagen über die Ausführungsgeschwindigkeiten verschiedener Prozesse relativ zueinander gemacht werden können.

- Das Fehlen einer globalen Sicht.
  Es ist bei hinreichender Größe des Systems unmöglich, aus einer einzigen Beobachtungsposition den Zustand des Gesamtsystems zu einem beliebigen Zeitpunkt sofort und als Ganzes zu erfassen. Dies macht Aussagen der Form "Das System befindet sich jetzt im Zustand x" unmöglich und hat weitreichende Auswirkungen auf Algorithmen, die den Zustand des Gesamtsystems betrachten müssen.

- Der inhärente Nichtdeterminismus.
  Obwohl die einzelnen Programme auf den Prozessoren alle vollkommen determiniert sein können, ist das Verhalten des Gesamtsystems trotzdem nichtdeterministisch. Dies liegt einerseits an den unbestimmten Laufzeiten der Nachrichten im zugrundeliegenden Kommunikationsnetzwerk, andererseits an der inhärenten Parallelität des Systems und dem fehlenden gemeinsamen Bezugspunkt "Zeit."

Die Beschäftigung mit verteilten Algorithmen stellt die Forscher also vor eine vollkommen neue Klasse von Problemen, die bei herkömmlichen Rechnersystemen kaum eine Bedeutung haben. Die verteilten Systeme sind geradezu eine "Verallgemeinerung" der herkömmlichen und bereits gut erforschten sequentiellen Systeme.

Die Notwendigkeit eines formalen Modells. Wegen der oben beschriebenen Probleme sind verteilte Algorithmen nicht dadurch begreifbar, daß man sich nacheinander und jeweils einzeln die Programme auf den jeweiligen Rechnerknoten anschaut. Immer wieder sind deshalb auch inkorrekte Algorithmen veröffentlicht worden (beispielsweise zur Erkennung der verteilten Terminierung [16]; vergleiche hierzu auch die Gegenbeispiele [133] und Erwiderungen [15]). Ein sehr einfacher Algorithmus, der vielleicht nur aus einem halben Dutzend Befehlen besteht, wirkt plötzlich sehr verwirrend, wenn er auf mehreren Prozessoren gleichzeitig ausgeführt wird und zu keinem Zeitpunkt klar ist, welcher Prozessor den nächsten Schritt ausführt [108, S. 3]. Auf der anderen Seite ist es auch sehr schwierig, Voraussagen über die Wirkungsweise eines Algorithmus zu machen, indem man sich anschaut, wie er in einem konkreten Fall abläuft. Bei einem erneuten Durchlauf desselben Algorithmus wird er möglicherweise vollkommen anders ablaufen, auch wenn sich die Eingaben überhaupt nicht verändert haben. All dies macht es notwendig, verteilte Algorithmen sehr genau zu beschreiben und ihren Ablauf in einem präzisen Modell zu verankern, um damit Aussagen über Eigenschaften des Algorithmus selbst und nicht eines konkreten Ablaufes zu ermöglichen.

Prozeß, Algorithmus, Berechnung. Obwohl man mit einem verteilten System oft ein Rechnernetz meint, in dem einzelne Computer räumlich verteilt zusammenarbeiten, hat es sich eingebürgert, die aktiven Elemente eines verteilten Systems mit dem Begriff "Prozeß" zu bezeichnen. Dies trägt der Tatsache Rechnung, daß auf den Einzelrechnern zumeist viele einzelne Aktivitätsträger in Form von Prozessen oder den sogenannten "Leichtgewichtsprozessen" (*threads*) auf der Ebene des Betriebssystems existieren, die alle jeweils einzeln an unterschiedlichen Aufgaben arbeiten und sich an der Kommunikation innerhalb des Gesamtsystems beteiligen.

Ein *lokaler Algorithmus* (*local algorithm*) ist eine Abarbeitungsvorschrift, die beschreibt, wie sich ein einzelner Prozeß zu verhalten hat. Wird der Algorithmus gestartet und wird er von einem Prozeß abgearbeitet, spricht man von einer *lokalen Berechnung* (*local execution*). Eine Berechnung ist demnach eine Folge von Zuständen, die der Prozeß durchläuft.

Beide Begriffe lassen sich nicht ohne weiteres auf verteilte Systeme übertragen, wie im Verlauf dieses Kapitels deutlich werden wird. Dennoch genügt vorerst ein einfach übertragenes Verständnis dieser beiden Begriffe in Bezug auf verteilte Systeme. Ziel dieses Kapitels ist es ja, diese Begriffe so genau zu fassen, daß Verfasser und Leser zu einem annähernd gleichen Verständnis der Sachverhalte kommen.

Ein *verteilter Algorithmus* ist demnach eine Menge von lokalen Algorithmen, die ihre Zustände nur mittels "Nachrichten" synchronisieren können. Eine *verteilte Berechnung* ist der Ablauf eines verteilten Algorithmus, also — grob gesprochen — eine "Folge von Zuständen des Gesamtsystems." Der Klarheit wegen möchte ich hier der Terminologie von Tel [136, S. 45] folgen und im Zweifelsfall den Begriff *Zustand* (*state*) nur auf einen einzelnen Prozeß beziehen. Der Gesamtzustand des Systems soll dann mit *Konfiguration* (*configuration*) bezeichnet werden.

2.3 Bestandteile eines formalen Modells

Modelle als Sichten der Wirklichkeit. Grundlage der Modellbildung ist die Abstraktion. Modelle sind eine abstrakte Repräsentation konkreter Systeme aus unserer täglichen Erfahrungswelt (der "Realität" oder "Wirklichkeit") und unterliegen — wie alle Formen der Abstraktion — auch einer Interpretation, welche Eigenschaften des Systems wesentlich und welche unwesentlich sind. Lamport und Lynch benennen solche Modelle auch mit unterschiedlichen Sichten (*views*) der Wirklichkeit [104, S. 1159]. Dies impliziert, daß es kein "bestes Modell" geben kann; jedes wird seine eigenen Stärken und Schwächen haben. Dennoch zeichnen sich in der Literatur gemeinsame Bezugspunkte ab, die eine Darstellung verteilter Systeme in einem einheitlichen Rahmenwerk möglich erscheinen lassen.

Die fehlende Taxonomie der Modelle. Trotz vieler Gemeinsamkeiten in den Modellen ist vermutlich wegen des rasanten Fortschrittes der Wissenschaft wenig auf dem Gebiet der zusammenfassenden Modelluntersuchungen veröffentlicht worden. Die einschlägigen Lehrbücher universitärer Curricula [108, 111, 136] benutzen alle ihre eigenen, speziellen Modelle und es steht meist dem Leser zu, die Modelle gegeneinander abzuwägen. Eine rühmliche Ausnahme bildet der Artikel von Lamport und Lynch aus dem "Handbook of Theoretical Computer Science" [104], denn die beiden Autoren versuchen dort, eine Taxonomie der Modelle zu definieren. Obwohl die dortige Einteilung aus heutiger Perspektive etwas veraltet und *ad hoc* wirkt, ist doch allein dieser Versuch zu loben.

Ich will jetzt und an dieser Stelle noch nicht näher auf alle dort beschriebenen "Achsen" der Taxonomie eingehen, sondern dies erst im abschließenden Teil (Abschnitt 2.5) dieses Kapitels tun. Zur Vorbereitung werde ich in den folgenden Abschnitten schrittweise ein immer komplexeres Gewebe von Definitionen und Beispielen flechten, angefangen bei einfachen Prozessen bis hin zu komplexen Systemen, und dabei die wesentlichen Punkte nach und nach vorstellen und diskutieren. So wird es möglich, am Ende dieses Kapitels aus einem höheren Blickwinkel das Gebiet zu überschauen, es sinnvoller einzuteilen und zusammenzufassen.

Synchrone und asynchrone Systeme. Eine Ausnahme von dieser Vorgehensweise erscheint mir angebracht und betrifft den Punkt "Zeit" in verteilten Systemen. Im einleitenden Abschnitt wurde bereits erwähnt, daß das Fehlen einer gemeinsamen Zeit zu den Hauptproblemen verteilter Systeme gehört. Nun ist "Zeit" ein Gegenstand mit vielen Facetten und das Fehlen einer gemeinsamen Zeit kann somit ganz unterschiedlich verstanden werden. Ein fundamentaler Unterschied in den Modellen besteht darin, ob das System eine gewisse Vorstellung von Zeit hat oder überhaupt gar keine. Erstere Modelle werden im Englischen *synchronous*, letztere *asynchronous systems* (oder *(a)synchronous network model*) genannt.[1] In den so bezeichneten asynchronen Systemen kommen versandte Nachrichten irgendwann an, arbeiten Prozesse irgendwie weiter, aber man weiß nie, wie lange das dauert [104, S. 1166]. In den synchronen Systemen auf der anderen Seite sind die Unterschiede in den Geschwindigkeiten der einzelnen Prozesse und auch die maximal mögliche Nachichtenlaufzeit durch einen festen Zeitwert beschränkt [126, S. 20]. Die Unabhängigkeit zwischen den Ablaufgeschwindigkeiten der einzelnen Prozesse bewegt sich so in einem festen, vorher bestimmten Zeitrahmen, der beispielsweise durch Standardalgorithmen zur Uhrensynchronisation [136, S. 448ff] erreicht werden kann. Bei einer sehr geringen Zeitschranke erhält man plötzlich Systeme, deren Berechnungen in "synchronen Runden" (*lock step mode*) ablaufen und in denen sich die Prozesse (wie beim Walzer) alle im gleichen Takt fortbewegen [108, S. 8]. Obwohl das andere (asynchrone) Modell das allgemeinere ist, liegen doch in der Realität zumeist synchrone Systeme in diesem Sinne vor.

[1] Man muß diese Begriffe unbedingt abgrenzen von *synchronous* und *asynchronous communication*; dazu später mehr in Abschnitt 2.3.5.

Petri-Netze. Bevor ich mich gleich in konkrete Modellierungsarbeit stürze, ist eine Anmerkung zu Petri-Netzen [23] angebracht. Petri-Netze sind ein weiteres Modell zur Beschreibung und Analyse von Abläufen mit nebenläufigen und nichtdeterministischen Vorgängen, wie sie auch in verteilten Systemen auftreten. Doch wie Lamport und Lynch in ihrem Überblicksartikel über verteilte Systeme [104] anmerken, werden Petri-Netze in diesem Zusammenhang normalerweise nicht zur Modellierung eingesetzt[2], weshalb ich sie auch hier nicht weiter in Betracht ziehen will.

2.3.1 Prozesse als einfache Automaten

Formale Modelle verteilter Algorithmen stützen sich zumeist auf die Vorstellung, der einzelne Prozeß sei ein einfacher Automat bestehend aus einer Menge Z von Zuständen, einer Menge $I \subseteq Z$ von Startzuständen und einer Zustandsübergangsfunktion $\delta : Z \mapsto Z$, die den eigentlichen Algorithmus beschreibt. Im Gegensatz zu den verbreiteten Modellen der Automatentheorie haben diese Automaten weder eine feste Menge von Endzuständen, noch eine Vorstellung von Ein- und Ausgabe, wie sie etwa Turing-Maschinen besitzen. Auch haben herkömmliche Automaten lediglich einen einzigen Startzustand. Dennoch ist das einfache Automatenmodell von der Ausdruckskraft her nicht direkt eingeschränkt. Die Formulierung einer Eingabe des Algorithmus wird durch die Erweiterung von einem auf mehrere Startzustände ermöglicht. Dies wird durch die Vorstellung gestützt, die einzelnen Prozesse enthielten ihre Startwerte (beispielsweise ihre Knotenidentität) in Form lokaler Variablen. Endzustände werden definiert als Zustände $z \in Z$, "in denen es nicht mehr weitergeht", die also keinen Folgezustand besitzen. Häufig wird die Zustandsübergangsfunktion in der Art und Weise erweitert, daß solche Zustände als Endlosschleifen der Form $\delta(z) = z$ definiert werden [4, S. 181][94, S. 18]. Dies modelliert Prozesse, die nichts mehr zu tun haben. Der Schwerpunkt dieser Vorstellung eines Prozesses als Automat liegt also weniger darin, daß der Prozeß eine "informationsverarbeitende Maschine" ist, sondern eher auf der Sichtweise des Prozesses als autonomer Maschine, "die vor sich hin rechnet."

Die Zustandsübergänge innerhalb des Prozesses werden gewöhnlich "Ereignisse" (engl. *events* oder *transitions*) genannt und können mit "Aktionen" (*actions*) eines Prozesses assoziiert werden. Üblicherweise unterscheidet man hierbei Eingabe- und Ausgabeaktionen (*input* und *output actions*) sowie interne Aktionen (*internal events*).

2.1 Beispiel (einfacher Automat) Ein einfaches Beispiel für einen "Prozeßautomaten" ist ein Nachrichtenzähler, also ein Prozeß, der Nachrichten

[2] "Other models, such as Petri nets [...], are usually not studied under the title of distributed computing, even though they may be used to model spatially distributed systems." [104, S. 1159]

empfängt, anschließend einen lokalen Zähler hochzählt, dann eine Quittung zurückschickt und wieder auf eine neue Nachricht wartet. Der interne Zustand wird gebildet durch einen einfachen Zähler z, der natürliche Zahlen annehmen kann und anfangs auf 0 steht, sowie durch eine boolesche Variable ack, die nur die Werte $true$ und $false$ annehmen kann. Die Menge der Startzustände besteht aus einem einzigen Zustand, nämlich dem, bei dem $z = 0$ und $ack = false$ ist.

Die Zustandsübergangsfunktion wird durch atomare Aktionen mit Vorbedingungen formuliert. Eine Empfangsaktion hat die Vorbedingung $ack = false$ und bewirkt folgende Zustandsänderungen:

$$ack := true \quad \text{sowie} \quad z := z + 1$$

Eine Sendeaktion hat die Vorbedingung $ack = true$ und führt dazu, daß die Variable ack wieder in ihren Ursprungszustand $false$ zurückversetzt wird.

Die Existenz der Variable ack stellt sicher, daß zu jeder Empfangsaktion genau eine Sendeaktion ausgeführt wird. Bei schwachem Nachrichtenverkehr wird der Automat vermutlich die meiste Zeit in einem Zustand mit $ack = false$ "herumdümpeln" und auf eine Eingabeaktion warten.

Ein-/Ausgabe-Automaten. Zur Modellierung einzelner Prozesse wird neuerdings oft [108, 140] das formale Rahmenwerk der sogenannten Ein-/Ausgabe-Automaten (*input/output automaton*) verwendet [108, S. 199ff]. Dieses ursprünglich von Nancy Lynch und Mark Tuttle [110] eingeführte Modell ist ein sehr mächtiges und genau ausgearbeitetes Beschreibungsmittel für viele Aspekte verteilter Algorithmen. Die Vorteile dieses Modells (vor allem seine Genauigkeit) werden in vielen Situationen jedoch durch die hohe Komplexität der mit ihm verbundenen Formalismen aufgewogen (obwohl auch hier die Möglichkeit zur Komposition und Abstraktion besteht). Darum will ich in dieser Diplomarbeit lieber andere, schneller verstehbare Beschreibungsmittel benutzen und dem Leser nicht auch noch diese zusätzliche Verständnislast aufbürden.

2.3.2 Bewachte Anweisungen

Im Beispiel 2.1 wurde der Algorithmus, der auf dem Knoten lief, größtenteils umgangssprachlich formuliert. Um auch in diesem Punkt den Ansprüchen an die Genauigkeit zu genügen, aber auf der anderen Seite keine Formeln auf Mengenniveau hinschreiben zu müssen, werde ich Algorithmen im weiteren Verlauf dieser Arbeit in einer formalen Programmiersprache formulieren, die auf das Konzept der "bewachten Anweisungen" (*guarded commands*) [47] aufbaut und deren Notation an die Sprache CSP [87] angelehnt ist. Grund für diese Wahl ist die Tatsache, daß diese Form der Präsentation in den meisten Veröffentlichungen über Selbststabilisierung (Gegenstand von Kapitel 4) [31, 49, 79] bevorzugt wird. Nancy

Lynch nennt die Notation "*precondition-effect style*" [108, S. 203]. (Entsprechende Konstrukte namens `ALT` und **select** gibt es in den Programmiersprachen Occam/Occam2 [114] beziehungsweise Ada [22].)

$$
\begin{array}{lccc}
\textbf{begin} & & & \\
 & \langle guard_1 \rangle & \longrightarrow & \langle commands_1 \rangle \\
[\!] & \langle guard_2 \rangle & \longrightarrow & \langle commands_2 \rangle \\
[\!] & \vdots & \longrightarrow & \vdots \\
[\!] & \langle guard_n \rangle & \longrightarrow & \langle commands_n \rangle \\
\textbf{end} & & &
\end{array}
$$

Abbildung 2.1: Allgemeine Form der *guarded commands*.

Bei dieser speziellen Form der Modellierung des Prozeßverhaltens geht man davon aus, daß die einzelnen Zustandsänderungen innerhalb eines Prozesses von außen als nicht unterteilbar, also *atomar* beobachtet werden. Auch Nachrichten, die von außen an einen Prozeß geschickt werden, unterbrechen die lokale Ausführung nicht augenblicklich sondern nur zu genau festgelegten Zeitpunkten, deren Frequenz implizit vom Programmierer durch die Formulierung des Algorithmus festgelegt wird. Demnach ist ein Programm eine Menge bewachter Anweisungen, welche selbst im Prinzip atomare Aktionen mit Vorbedingungen sind (siehe Abbildung 2.1; das Kästchen "$[\!]$" trennt unterschiedliche *guarded commands* untereinander). Die Vorbedingung (die "Wache", engl. *guard*) ist ein logisches Prädikat auf dem lokalen Zustand und ist gefolgt von einer Liste von Befehlen (den *commands*). Die Befehle werden nur ausgeführt, wenn die Wache zu einem "wahren" booleschen Wert auswertet. In diesem Fall spricht man davon, daß die Anweisungen *aktiv* (*active*) oder *ermöglicht* (*enabled*) sind. Sind mehrere Anweisungen aktiv, so erfolgt die Auswahl der auszuführenden Befehlsfolge nichtdeterministisch. In der Notation sind die Wache und die Befehle durch einen Pfeil voneinander getrennt. Ein Ausführungszyklus eines solchen Programmes besteht aus der Evaluation aller Wachen und der Ausführung einer beliebigen ermöglichten Anweisungsfolge. Anschließend beginnt dieser Ablauf von neuem.

2.2 Beispiel (Beispiel 2.1 in formaler Notation) Den Algorithmus aus Beispiel 2.1 würde man in der Notation der *guarded commands* wie folgt beschreiben:

$$
\begin{array}{llcl}
\textbf{begin} & & & \\
 & \neg ack & \longrightarrow & ack := true;\ z := z + 1 \\
[\!] & ack & \longrightarrow & ack := false \\
\textbf{end} & & &
\end{array}
$$

Die erste Zeile entspricht der Empfangs- und die letzte Zeile entspricht der Sendeaktion.

Aus dem eben gezeigten Beispiel wird noch nicht ganz klar, wie der eigentliche Nachrichtenaustausch zustande kommt. Man kann sich jedoch vorerst vorstellen, daß Eingabeaktionen spontan von außen angestoßen werden können (nämlich wenn eine entsprechende Nachricht vorliegt) und daß sowohl Eingabe- als auch Ausgabeaktionen den Zustand der Umgebung verändern.

Das obige Beispiel zeigt deutlich, wie der Nichtdeterminismus des Sprachkonstruktes durch die Zuhilfenahme von Variablen in Vorbedingungen eingeschränkt werden kann. Und tatsächlich ist das Programm eine normale sequentielle Schleife.

Die Umsetzung eines Programmes aus bewachten Anweisungen in eine Zustandsübergangsfunktion ist einfach: Die Zustandsübergangsfunktion ist eine Teilmenge aller Tupel aus $Z \times Z$. Man muß also nur die Zustände vor und nach der Ausführung einer Aktion formal beschreiben und die entsprechenden Tupel in die Funktion mitaufnehmen [77, S. 8].

2.3 Beispiel (Umsetzung von Beispiel 2.2 in Mengennotation) In Beispiel 2.2 (bzw. Beispiel 2.1) besteht der Zustand des Prozesses aus den Zuständen der Variablen *ack* und z, der hier als Tupel $\langle x, y \rangle$ dargestellt werden soll, wobei x ein Wert von *ack* ist und y ein Wert von z (zur Abkürzung benutze ich die Bezeichnung $\tilde{z}$ als "*don't care*"-Symbol). Die enstehende Funktion δ besteht dann aus den folgenden Tupeln:

$$\delta = \{(\langle \mathit{false}, \tilde{z} \rangle, \langle \mathit{true}, \tilde{z}+1 \rangle), (\langle \mathit{true}, \tilde{z} \rangle, \langle \mathit{false}, \tilde{z} \rangle)\}$$

2.3.3 Kommunikation über Nachrichten

Eine einfache Automatendefinition läßt noch nicht deutlich erkennen, wie Kommunikation durch den Austausch von Nachrichten modelliert werden kann. An diesem Punkt unterscheiden sich die in der Literatur vorkommenden Modelle am deutlichsten.

Alle gehen davon aus, daß es eine globale Menge M von Nachrichten gibt. Folgende Definition soll als erstes Beispiel dienen.

2.4 Definition (lokaler Algorithmus nach Tel [136, Def. 2.4]) Ein lokaler Algorithmus eines Prozesses ist ein Quintupel $(Z, I, \delta_i, \delta_s, \delta_r)$, wobei Z eine Menge von Zuständen und $I \subseteq Z$ eine Menge von Startzuständen sind, δ_i eine Relation auf $Z \times Z$ und δ_s, sowie δ_r Relationen auf $Z \times M \times Z$.

Die drei Relationen sind Ausprägungen der Zustandsübergangsfunktion, die der Tatsache Rechnung tragen, daß ein- und ausgehende Nachrichten ebenfalls einen Zustandswechsel hervorrufen. Die Relationen δ_s und δ_r stehen für die Sende- und Empfangsereignisse des Prozesses, δ_i steht für die internen Ereignisse. Die Relationen lassen sich zu einer einzigen binären Relation $\delta \subseteq Z \times Z$ zusammenfassen durch die Definition:

$$(c, d) \in \delta \iff (c, d) \in \delta_i \vee \exists m \in M : (c, m, d) \in (\delta_s \cup \delta_r)$$

2.5 Beispiel Der Erhalt einer bestimmten Nachricht m im Zustand c_1 und die Reaktion des Algorithmus durch einen Übergang in den Zustand d_1 kann somit durch Hinzufügen eines Tripels (c_1, m, d_1) in δ_r modelliert werden. Die Produktion (das Versenden) einer Nachricht a im Zustand c_2 (bei gleichzeitigem Übergang in Zustand d_2) wird durch Hinzunahme von (c_2, a, d_2) in δ_s erreicht.

2.6 Beispiel (Beispiel 2.1 in der Notation von Tel) In Anlehnung an Beispiel 2.3 will ich nochmals den Algorithmus des Nachrichtenzählers aus Beispiel 2.1 in der formalen Notation von Tel beschreiben:

Wie bei dem Ein-/Ausgabe-Automaten sind Zustände Elemente des Kreuzproduktes der Variablen *ack* und z und sollen als $\langle x, y\rangle$ dargestellt werden. Die Menge der Nachrichten M besteht aus den beiden Elementen m und a. Wenn man auf den Kunstgriff eines "*don't care*"-Symbols verzichtet, kommt man auf folgende Formulierungen der Komponenten $(Z, I, \delta_i, \delta_s, \delta_r)$:

$$\begin{aligned}
Z &= \{\langle \mathit{false}, 0\rangle, \langle \mathit{true}, 0\rangle, \langle \mathit{false}, 1\rangle, \langle \mathit{true}, 1\rangle, \dots\} \\
I &= \{\langle \mathit{false}, 0\rangle\} \\
\delta_i &= \emptyset \\
\delta_s &= \{(\langle \mathit{true}, 1\rangle, a, \langle \mathit{false}, 1\rangle), \dots\} \\
\delta_r &= \{(\langle \mathit{false}, 0\rangle, m, \langle \mathit{true}, 1\rangle), (\langle \mathit{false}, 1\rangle, m, \langle \mathit{true}, 2\rangle), \\
&\phantom{= \{} (\langle \mathit{false}, 2\rangle, m, \langle \mathit{true}, 3\rangle), \dots\}
\end{aligned}$$

Die Punkte am Ende der Mengenaufzählung deuten an, daß es sich hier um eine unendliche Zustandsmenge und unendliche Relationen handelt. Konnte dies in Beispiel 2.3 durch Einführung eines *don't care*-Symbols noch erfolgreich verschleiert werden, tritt diese Tatsache hier deutlich zutage. In der Realität wird man derartige Automaten im allgemeinen nie bauen können, da die Ressourcen an Speicher und Zeit immer endlich sind. Dennoch ist es praktisch, diese Möglichkeit im Modell nicht von vornherein einzuschränken, da sich bestimmte Probleme [79] damit viel eleganter lösen lassen oder überhaupt erst lösbar werden. (Beispielsweise werden unbeschränkte Sequenznummern durch diese Art von Automaten modellierbar.)

Woher weiß aber der Prozeß beim Erhalt der Nachricht m, von welchem Prozeß die Nachricht versendet wurde? Man kann es sich so vorstellen, daß Sender und Adressat in die Nachricht selbst hineincodiert sind; eigentlich müßte man dann $m_{i,j}$ schreiben, aber es ist notationell einfacher (und auch gebräuchlicher), die Adressen an die Sende- und Empfangsoperationen anzuhängen (etwa in der Form "**send**(m) **to** j").

2.7 Bemerkung (Zustandsübergangsrelation und Nichtdeterminismus) Es ist durchaus relevant, daß in der vorangehenden Definition nicht mehr von einer Zustandsübergangsfunktion gesprochen wurde, sondern von einer Relation auf Zuständen. Eine Funktion ist eine eindeutige Zuordnung von Elementen aus einer Menge A zu Elementen einer Menge B. Zwar dürfen mehrere Elemente aus B dasselbe Urbild aus A besitzen, doch ist die Abbildung in der anderen Richtung immer eindeutig. Dies entspricht im Falle des Automaten einem strengen Determinismus. Die Benutzung einer Relation zur Beschreibung der Zustandsübergänge ermöglicht es, Nichtdeterminismus zu modellieren.

Der Nichtdeterminismus wird programmiersprachlich mittels bewachter Anweisungen ausgedrückt und ist ein mächtiges Hilfsmittel bei der Programmierung verteilter Systeme, wie später noch deutlich werden wird.

2.3.4 Kombination mehrerer Prozesse

Die Definition eines lokalen Algorithmus mittels eines Automaten ermöglicht es nun, den Automatenbegriff auf ein gesamtes verteiltes System auszudehnen und den Gesamtautomaten mittels der Komponenten seiner Teile zu definieren.

Informal gesprochen ist ein verteiltes System demnach ein großer Automat, der aus einer Ansammlung von Einzelautomaten (Prozessen) besteht, die Nachrichten versenden und erhalten können und sich so synchronisieren. Der Zustand des Gesamtsystems besteht aus den Zuständen der einzelnen Komponenten zuzüglich der Zustände der Kanäle. Die Kanäle selbst können wieder als Automaten [108], als gemeinsame Variablen von jeweils zwei Prozessen [79, S. 449] oder — wie in Definition 2.9 — einfach als Multimengen [136, S. 46] definiert werden (also als die Menge der Nachrichten, die noch unterwegs sind). Multimengen verhalten sich wie Mengen mit der Ausnahme, daß Elemente mehrfach auftreten können. Dies ist notwendig, um das Unterwegssein mehrerer identischer Nachrichten modellieren zu können. Ein Zustandsübergang des Systems findet immer dann statt, wenn eine einzelne Komponente einen Zustandswechsel vornimmt. Das Modell muß zudem gewährleisten, daß versandte Nachrichten auch ankommen und nicht dupliziert werden oder verloren gehen (Fehler müssen ggf. explizit modelliert werden). In Fortführung von Definition 2.4 will ich dies verdeutlichen.

2.8 Definition (verteilter Algorithmus nach Tel [136, Def. 2.5]) Ein verteilter Algorithmus für eine endliche Menge $P = \{p_1, \dots, p_n\}$ von Prozessen ist die Menge der lokalen Algorithmen von allen Prozessen p_i.

Das Verhalten eines verteilten Algorithmus wird durch einen Gesamtautomaten beschrieben, der aus der Kombination der Einzelautomaten entsteht. Dabei ist die Art der Kommunikation zu berücksichtigen. Ich will zunächst von der gebräuchlicheren Variante der *asynchronen Kommunikation* ausgehen und auf andere Arten später eingehen.

2.9 Definition (Automat eines verteilten Algorithmus mit asynchroner Kommunikation nach Tel [136, Def. 2.6]) Der Automat eines verteilten Algorithmus mit asynchroner Kommunikation ist ein Tripel (C, I, δ) und ergibt sich aus den lokalen Algorithmen einer Menge $P = \{p_1, \ldots, p_n\}$ von Prozessen wie folgt:

Bezeichne $(Z^k, I^k, \delta_i^k, \delta_s^k, \delta_r^k)$ die Komponenten des lokalen Algorithmus von Prozess p_k und bezeichne $\mathcal{M}(M)$ die Menge aller Multimengen der Nachrichtenmenge M.

- Die Zustandsmenge C ist die Menge der Vektoren aus allen möglichen lokalen Zuständen der einzelnen Prozeße p_k, sowie einer Multimenge von Nachrichten, also:

$$C = \{(z_1, \ldots, z_n, \mu) | (\forall k \in \{1, \ldots, n\} : z_k \in Z^k) \wedge \mu \in \mathcal{M}(M)\}$$

- Die Startzustände sind alle Vektoren aus C, die nur aus Startzuständen der Einzelalgorithmen bestehen und eine leere Nachrichtenmenge besitzen, also:

$$I = \{(z_1, \ldots, z_n, \mu) | (\forall k \in \{1, \ldots, n\} : z_k \in I^k) \wedge \mu = \emptyset\}$$

- Die Zustandsübergangsrelation $\delta \subseteq C \times C$ entsteht aus den Relationen δ_i^k, δ_s^k und δ_r^k der Einzelprozesse wie folgt:

  Das Paar

$$(z_1, \ldots, z_k, \ldots, z_n, \mu_1), (z_1, \ldots, z_k', \ldots, z_n, \mu_2)$$

  ist genau dann in δ wenn eine der drei folgenden Bedingungen zutrifft:

  1. $(z_k, z_k') \in \delta_i^k \wedge \mu_1 = \mu_2$
  2. $(\exists m \in M : (z_k, m, z_k') \in \delta_s^k) \wedge \mu_2 = \mu_1 \cup \{m\}$
  3. $(\exists m \in M : (z_k, m, z_k') \in \delta_r^k) \wedge \mu_1 = \mu_2 \cup \{m\}$

Erläuterungen der Definition. Die globalen Zustände (also die Konfigurationen) werden wie oben beschrieben aus den Zuständen aller Prozesse gebildet. Die Multimengen modellieren die Nachrichten, die noch unterwegs sind (also den Zustand der Kanäle). Diese Sicht spiegelt die Auffassung wieder, das Netzwerk sei ein Kommunikations-Dienst, der Punkt-zu-Punkt-Kommunikation von Prozeß zu Prozeß zur Verfügung stellt. Der Vorteil davon ist, daß die Topologie des Netzes nicht eingeschränkt wird. Werden Kanäle als gemeinsame Variable oder selbst als Automaten modelliert, so müssen deren Zustände direkt in den Zustandsvektor eingehen und man kann durch das Hinzunehmen oder Weglassen einzelner Komponenten eine konkrete Netz-Topologie nachbilden.

Die drei Fälle der Zustandsübergangsrelation spiegeln wieder, welche Art von Ereignis tatsächlich passiert ist: Bei einem internen Ereignis ändert sich der Zustand des Kommunikationsnetzwerkes nicht. Bei Sende- und Empfangsereignissen wird die gesendete Nachricht “auf das Netz gegeben,” bzw. die empfangene Nachricht aus dem Netz entfernt.

Sprachliche Einbindung von Kommunikation. Die Kommunikationsmechanismen müssen auch noch programmiersprachlich in die Notation der bewachten Anweisungen eingebaut werden. Dies geschieht durch die Hinzunahme zweier besonderer Aktionen **out** (entspricht dem Senden, oft auch geschrieben als **send**) und **in** (entspricht dem Empfangen, auch **receive** oder **rcv** geschrieben).[3] Beide Aktionen dürfen neben Zuweisungen im *command*-Teil der bewachten Anweisungen gebraucht werden, die **in**-Aktion allerdings auch innerhalb einer Wache. Die Aktion gilt als "wahr", wenn eine entsprechende Nachricht für sie anliegt. In Fortführung von Beispiel 2.2 soll dies nachfolgend deutlicher gemacht werden.

process *Message Counter*
  var $z : \mathbb{N}$ **init** 0
   ack : **boolean init** *false*
  begin
    $\neg ack \wedge \textbf{in}(m) \longrightarrow ack := true;\ z := z + 1$
  $[\!]$ $ack \longrightarrow \textbf{out}(a);\ ack := false$
  end

Abbildung 2.2: Der lokale Algorithmus des Nachrichtenzählers aus Beispiel 2.10.

2.10 Beispiel (vollständiger Nachrichtenzähler mit Sendeprozeß) Im Algorithmus des Nachrichtenzählers aus Beispiel 2.2 werden nun die Sende- und Empfangsaktionen explizit eingebaut. Der Zähler soll tatsächlich nur beim Vorliegen einer Nachricht hochgezählt werden. Darum wird die **in**-Anweisung mit in die Wache aufgenommen. Sobald das Hochzählen passiert ist, wird eine Bestätigung mittels **out** zurückgesandt. Der Algorithmus ist in Abbildung 2.2 dargestellt.

Der "Partnerprozeß," dessen Nachrichten gezählt werden und dessen lokaler Algorithmus in Abbildung 2.3 dargestellt ist, schläft die meiste Zeit (modelliert durch eine boolesche Variable *sleep*) und wacht ab und zu einmal auf, um eine Nachricht zu versenden. Anschließend wartet er auf eine Bestätigung und schläft wieder ein.

Die Adressierung der Empfänger erfolgt hier nicht innerhalb des Sprachkonstruktes sondern durch die besondere Art der Nachricht, die versendet wird (dabei sollen a und m unterschiedliche Nachrichten der Nachrichtenmenge M sein). Durch die Einbindung der **in**-Kommandos innerhalb der Wachen wird nun selektives Empfangen möglich, eine Fähigkeit, ohne die man bei der Programmierung auf Probleme stößt (siehe hierzu später Beispiel 2.12).

[3] Ähnlichkeiten zu den Primitiven der Programmiersprache *Linda* [68] sind unbeabsichtigt.

process *Partner*
  var *sleep* : **boolean init** *true*
   wait : **boolean init** *false*
  begin

$$\begin{array}{lll} & sleep & \longrightarrow & sleep := true \\ [\!] & sleep & \longrightarrow & sleep := false \\ [\!] & \neg sleep \wedge \neg wait & \longrightarrow & \mathbf{out}(m);\ wait := true \\ [\!] & wait \wedge \mathbf{in}(a) & \longrightarrow & wait := false;\ sleep := true \end{array}$$

  end

Abbildung 2.3: Der lokale Algorithmus des Partnerprozesses aus Beispiel 2.10.

Reaktivität von Prozessen. Was ist, wenn ein Prozeß A eine Nachricht an Prozeß B schickt, der aber gerade keine Nachricht von A erwartet? Im allgemeinen Fall würde man hier von "Pech" spechen: A bleibt so lange blockiert, bis B bereit ist. Dies kann lange dauern, möglicherweise unendlich lange, wenn B gar nichts von A hören will, und kann ein Indiz für einen Protokollfehler sein oder zu einer Verklemmung führen.

In der Literatur wird dieses Problem wegdefiniert. (Tel schreibt beispielsweise dazu: "It is [always] assumed that for each message there is a unique process that can receive the message." [136, S. 47]) Die Forderung dort lautet: Jede Eingabeaktion muß immer und zu jedem Zeitpunkt möglich sein. Das heißt, daß jeder Prozeß prinzipiell mit jeder Nachricht zurecht kommen muß, die ihm zugeschickt wird. Nancy Lynch nennt dies *input-enabling property* [108, S. 202f], was ich als *Reaktivität* übersetzen möchte.

Natürlich ist die Forderung nach Reaktivität von Prozessen sehr stark, vielleicht zu stark, um nützlich zu sein. Aber die Erfahrung zeigt, daß eingehende Nachrichten, an die man nicht gedacht hat, eine ernstzunehmende Fehlerquelle beim Entwurf verteilter Algorithmen sind. Wenn man hingegen Reaktivität fordert, schließt man so von vornherein diese Art von Fehlern aus [108, S. 203]. Die bis eben genannten Beispiele sind zwar nicht reaktiv, könnten aber in einfacher Weise erweitert werden, um diese Forderung zu erfüllen, sei es auch dadurch, daß unerwartete Nachrichten einfach ignoriert werden.

2.3.5 Synchrone und asynchrone Kommunikation

In der Literatur exisitert kein einheitliches Verständnis der Begriffe *synchron* und *asynchron* bezogen auf Kommunikation, insbesondere nicht in ihrer Abgrenzung zu den Begriffen *blockierend* und *nicht-blockierend.* Informal wird asynchrone Kommunikation immer verglichen mit dem Versenden einer Postkarte, synchrone Kommunikation mit dem Nachrichtenaustausch per Telefon.[4]

Das Adjektiv "synchron" bedeutet im normalen Sprachgebrauch "gleichzeitig" (von griech. *syn* = zusammen, zugleich und *chronos* = Zeit). Bezogen auf die Kommunikation in verteilten Systemen entspricht dies dem gleichzeitigen Stattfinden von Sende- und Empfangsereignis. Da Nachrichten in der Realität (selbst per Telefon) immer eine endliche Ausbreitungsgeschwindigkeit haben, genügt in diesem Zusammenhang ein Verständnis von "virtueller Gleichzeitigkeit", das heißt es existiert ein gemeinsamer Zeitpunkt der (gedachten) Globalzeit, an dem sich beide Prozesse synchronisieren. Falls also beispielsweise ein Prozeß P eine Nachricht synchron an Prozeß Q senden (und umgekehrt Q eine Nachricht von P erhalten) will, können beide Prozesse nichts tun, bis die Nachricht untereinander ausgetauscht ist [38, S. 184]. Aus Sicht der Prozesse erfolgt die Kommunikation zur "gleichen Zeit" (d.h. beide arbeiten erst nach erfolgter Kommunikation weiter).

Im formalen Modell des verteilten Systems bedeutet dies, daß Sende- und Empfangsereignis *ein* Zustandsübergang im System sind. Definition 2.9 muß dann in folgender Art und Weise abgeändert werden:

2.11 Definition (Automat eines verteilten Algorithmus (mit synchroner Kommunikation) nach Tel [136, Def. 2.7]) Der Automat eines verteilten Algorithmus (mit synchroner Kommunikation) ist ein Tripel (C, I, δ) und ergibt sich aus den lokalen Algorithmen einer Menge $P = \{p_1, \ldots, p_n\}$ von Prozessen wie folgt:

Bezeichne $(Z^k, I^k, \delta_i^k, \delta_s^k, \delta_r^k)$ die Komponenten des lokalen Algorithmus von Prozeß p_k und bezeichne M die Menge aller möglichen Nachrichten.

- Die Zustandsmenge C ist die Menge der Vektoren aus allen möglichen lokalen Zuständen der einzelnen Prozesse p_k, also:

$$C = \{(z_1, \ldots, z_n) | \forall k \in \{1, \ldots, n\} : z_k \in Z^k\}$$

- Die Startzustände sind alle Vektoren aus C, die nur aus Startzuständen der Einzelalgorithmen bestehen, also:

$$I = \{(z_1, \ldots, z_n) | \forall k \in \{1, \ldots, n\} : z_k \in I^k\}$$

- Die Zustandsübergangsrelation $\delta \subseteq C \times C$ entsteht aus den Relationen δ_i^k, δ_s^k und δ_r^k der Einzelprozesse wie folgt:

  Das Paar

$$(z_1, \ldots, z_k, \ldots, z_l, \ldots, z_n), (z_1, \ldots, z_k', \ldots, z_l', \ldots, z_n)$$

[4] Wie bereits oben erwähnt, muß man die englischen Begriffe *synchronous* und *asynchronous communication* unbedingt abgrenzen von den Begriffen *(a-)synchronous network model* oder *(a-)synchronous system*.

ist genau dann in δ wenn eine der zwei folgenden Bedingungen zutrifft:

1. $(z_k, z'_k) \in \delta_i^k \wedge z_l = z'_l$
2. $\exists m \in M : (z_k, m, z'_k) \in \delta_s^k \wedge (z_l, m, z'_l) \in \delta_r^l$

Die Tatsache, daß Kommunikation als ein einzelnes atomares Ereignis betrachtet wird, läßt die Kommunikationskanäle immer leer erscheinen. Dies ist auch der Grund dafür, warum der Zustand der Kanäle nicht in die Konfiguration des Gesamtsystems eingeht.

Es ist bekannt, daß keine der beiden Kommunikationsarten der andere überlegen ist. In einer aktuellen Untersuchung [38] analysieren Charron-Bost, Mattern und Tel die Eigenschaften verteilter Berechnungen bei unterschiedlichen Annahmen über die zugrundeliegenden Kommunikationsmechanismen. Dazu schreiben sie:

> "While asynchronous communication is less prone to deadlocks and often allows a higher degree of parallelism [...], its implementation requires complex buffer management and flow control mechanisms. Furthermore, algorithms making use of asynchronous communication are often more difficult to develop and to verify than algorithms working in a synchronous environment. It is a well known fact, however, that it is possible to simulate one mode with the other." [38, S. 173f]

Da der asynchrone Fall oft als "allgemeinerer" Fall der Kommunikation betrachtet wird, gehen die Autoren auch der Frage nach, welche Algorithmen für asynchrone Kommunikation auch mit synchroner Kommunikation korrekt laufen und umgekehrt. Dabei zeigt sich, daß Transformationen in beide Richtungen nicht-trivial sind, daß es aber in der Regel einfacher ist, Algorithmen mit asynchroner Kommunikation auf den synchronen Fall zurückzuführen. In dieser Richtung können Korrektheitsbeweise nämlich größtenteils beibehalten werden, nur besteht die Gefahr von Verklemmungen (*deadlocks*).

2.12 Beispiel (Das Potential von Nichtdeterminismus [113, S. 97ff]) Mittels synchroner Kommunikation und vollkommen deterministischer Programmkonstrukte soll ein beschränkter Puffer als Prozeß realisiert werden. Der Puffer soll zwischen einem Sende- und einem Empfangsprozeß plaziert werden und ihre Kommunikation entkoppeln (siehe Abbildung 2.4). Dabei soll der Puffer so realisiert werden, daß erstens der Sender neue Puffereinträge an den Puffer schicken kann bis er voll ist, und zweitens der Empfänger durch bestimmte *request*-Nachrichten den nächsten Puffereintrag anfragen kann.

Eine programmiersprachliche Realisierung ohne *guarded commands* (also ohne Nichtdeterminismus) könnte etwa wie folgt aussehen: In einer Endlosschleife wartet der Puffer auf eine neue Nachricht. Neue Puffereinträge werden einfach in den Puffer geschrieben (solange noch Platz ist);

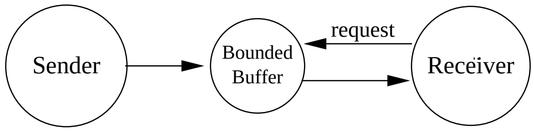

Abbildung 2.4: Architektur des beschränkten Puffers aus Beispiel 2.12.

Anfragen nach Puffereinträgen werden (solange noch etwas im Puffer ist) durch die Lieferung des nächsten Pufferelementes beantwortet.

```
process Bounded Buffer
  var ⟨lokaler Pufferspeicher⟩
  while true begin
    if ⟨Puffer ≠ voll⟩ then begin
      receive p from Sender
      ⟨füge p in lokalen Pufferspeicher ein⟩
    end
    if ⟨Puffer ≠ leer⟩ then begin
      receive request from Receiver
      ⟨entferne q aus lokalem Pufferspeicher⟩
      send q to Receiver
    end
  end
```

Abbildung 2.5: Lokaler Algorithmus des Puffers aus Beispiel 2.12.

Das Problem mit dieser Form der Realisierung liegt aber darin, daß der Puffer für den Sender blockiert, obwohl nur ein einzelnes Element gespeichert ist. Betrachtet man beispielsweise den Fall, daß der Puffer leer ist und eine Nachricht p vom Sender empfangen wird. Nachdem diese in den lokalen Pufferspeicher eingefügt wurde, geht der Puffer in die zweite **if**-Anweisung hinein (der Puffer ist ja nicht leer) und wartet auf eine Anfrage des Empfängers. Dies kann unter Umständen sehr lange dauern und blockiert den Puffer, dessen Kapazität ja noch längst nicht erreicht sein muß. Eine weitere **if**-Abfrage an dieser Stelle nach einer weiteren Nachricht des Senders würde den Empfänger blockieren, falls der Sender gar keine weitere Nachricht mehr zu schicken hat. Das Problem besteht also darin, daß man nie weiß, welche Nachricht als nächstes eintrifft, also welches *receive*-Kommando man als nächstes benötigt. Auch eine Lösung, in der man erst einmal eine Nachricht (egal von wem) empfängt und anschließend durch Inspektion der Nachricht entscheidet, ob sie ein neues Pufferelement oder eine Anfrage ist, würde nicht weiterhelfen, da man beispielsweise im Falle *Puffer* = *voll* gar kein neues Pufferelement empfangen kann und will. Man müßte ein komplexes Protokoll von Nachrichten vereinbaren (quasi ein "aufgesetztes selektives Empfangen"), um diesem Problem beizukommen, welches das Programm wesentlich komplizierter machen würde als nötig.

Die Lösung mit *guarded commands* dagegen ist bestechend einfach (siehe Abbildung 2.6) und besteht im wesentlichen nur aus zwei Zeilen. So ungewöhnlich Nichtdeterminismus in sequentiellen Programmen ist, so mächtig ist doch seine sprachliche Ausdruckskraft in verteilten Systemen.

process *Nondeterministic Bounded Buffer*
  var $\langle$*lokaler Pufferspeicher*$\rangle$
  begin
    $\langle \textit{Puffer} \neq \textit{voll} \rangle \wedge \mathbf{in}(q) \longrightarrow \langle \textit{füge } p \textit{ in lokalen Pufferspeicher ein} \rangle$
  $[\!]$ $\langle \textit{Puffer} \neq \textit{leer} \rangle \wedge \mathbf{in}(\textit{request}) \longrightarrow \langle \textit{entferne } q \textit{ aus lokalem Pufferspeicher} \rangle;\ \mathbf{out}(q)$
  end

Abbildung 2.6: Lokaler Algorithmus des Puffers aus Beispiel 2.12 mit *guarded commands*.

2.3.6 Berechnungen und Kausalität

In diesem Abschnitt möchte ich nochmals auf den Begriff der *verteilten Berechnung* eingehen, der schon in der Einleitung (Abschnitt 2.2) intuitiv eingeführt wurde. Dort wurde eine verteilte Berechnung als die Ausführung eines verteilten Algorithmus beschrieben, also als eine Folge von Konfigurationen, die das Gesamtsystem durchläuft. Gewöhnlich stellt man eine verteilte Berechnung als Raum/Zeit-Diagramm dar. Abbildung 2.7 zeigt ein Beispiel für ein solches Diagramm. Darin sind die einzelnen Prozesse durch horizontale Zeitachsen dargestellt (die Zeit vergeht "von links nach rechts") und Ereignisse durch Punkte auf den Achsen. Zusammengehörige Sende- und Empfangsereignisse werden durch Pfeile verbunden, die den Weg der Kommunikation anzeigen. (Beispielsweise kommt die Nachricht, die in Ereignis a losgeschickt wurde, in Ereignis f an, und Ereignis k ohne Pfeil steht für ein internes Ereignis.)

In Raum/Zeit-Diagrammen ist sehr schön zu sehen, wie einzelne Ereignisse auf den Prozessen die globale Konfiguration beeinflussen. Eine Konfiguration entspricht einem senkrechten Schnitt durch das Raum/Zeit-Diagramm, der keine Ereignisse schneidet (in Abbildung 2.7 zum Beispiel γ). Die einzelnen Ereignisse wirken sich immer nur auf einen kleinen Teil des Gesamtsystems aus. Ereignisse, die unterschiedliche Teile der Konfiguration beeinflussen, nennt man *unabhängig*.

Ein Problem (oder eine Stärke) von Raum/Zeit-Diagrammen ist, daß eine gedachte Globalzeit (also eine Zeit, die für alle Prozesse gleichermaßen gilt) implizit vorhanden ist. Ereignisse, die "weiter rechts" als andere

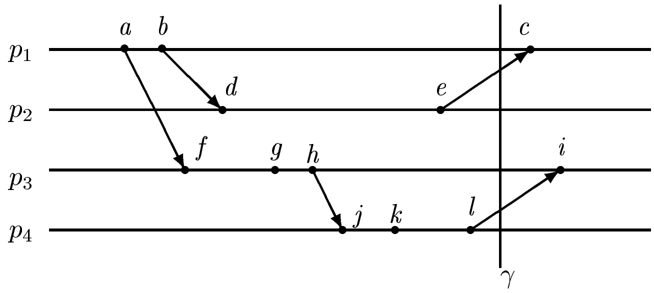

Abbildung 2.7: Ein Raum/Zeit-Diagramm [136, S. 54].

stehen, passieren in dieser Zeit "später" (zum Beispiel Ereignis g nach Ereignis d). Die eingangs geschilderten Probleme verteilter Systeme besagen aber, daß diese Globalzeit nicht allen Prozessen gleichzeitig zur Verfügung stehen kann. Es gibt also für die Prozesse keine Möglichkeit, zwei beliebige Ereignisse nach dem Globalzeitpunkt ihres Eintretens zu sortieren. Gerade die unabhängigen Ereignisse machen hier die Probleme: Will Prozeß p_3 beispielsweise feststellen, ob sein Ereignis g vor oder nach Ereignis d (auf Prozeß p_2) stattgefunden hat, muß er mit p_2 kommunizieren und wird von ihm den Zeitpunkt genannt bekommen, zu dem d auf p_2 passierte. Doch mit diesem Zeitpunkt kann p_3 nichts anfangen, da ungewiß ist, um wieviel die Uhr von p_2 gegenüber der von p_3 vor- oder nachgeht. Zwar kann man heutzutage die Uhren einzelner Prozesse schon recht genau (im Bereich von Millisekunden) synchronisieren, doch laufen die Operationen zum Teil im Mikrosekundenbereich ab, und wie eingangs dieses Kapitels gesagt wurde, ist eine beliebig genaue Uhrensynchronisation in verteilten Systemen nicht möglich.

Wie gesehen machen unabhängige Ereignisse Probleme, doch sie haben auch die interessante Eigenschaft, daß ihre relative Ablaufreihenfolge für den Ausgang der Berechnung keine Rolle spielt. In Abbildung 2.7 ist es zum Beispiel egal, ob Ereignis b vor oder nach Ereignis f stattfindet; in beiden Fällen wird die Berechnung zum gleichen Ergebnis kommen, da beide Ereignisse nicht die Ursache füreinander sind. Der Begriff, der diese Tatsache genauer faßt, wird durch die Definition der Kausalrelation festgelegt.

2.13 Definition (Kausalrelation) Die Kausalrelation $<$ auf Ereignissen ist die kleinste Relation, die folgende drei Bedingungen erfüllt:

1. Falls e und f Ereignisse auf demselben Prozeß sind und e vor f passiert, dann gilt $e < f$.

2. Falls s ein Sendeereignis und r das zugehörige Empfangsereignis ist, dann gilt $s < r$.

3. Die Relation $<$ ist transitiv.

Die Kausalrelation formalisiert den Begriff der *kausalen Abhängigkeit*, also der potentiellen Beeinflussung. Ereignisse auf ein und demselben Prozeß sind in natürlicher Weise voneinander abhängig: Ereignis a in Abbildung 2.7 kann den Zustand des Prozesses p_1 derart ändern, daß es zum Ereignis b kommt (Bedingung 1 der Definition). Genauso beeinflußt a auch den Zustand von p_3 durch das Senden einer Nachricht. Ereignis f ist also direkt abhängig von a (Bedingung 2 der Definition). Die Transitivität (Bedingung 3) gewährleistet, daß auch die mögliche Abhängigkeit von Ereignis j beispielsweise von Ereignis a formal gefaßt wird: Der Empfang der Nachricht von p_1 in f kann durchaus den Prozeß p_3 zum Senden der Nachricht in h veranlaßt haben. Es kann natürlich sein, daß in einem konkreten Algorithmus zwischen j und a keinerlei kausaler Zusammenhang besteht, jedoch kann er bei der alleinigen Betrachtung der Struktur der Berechnung nicht ausgeschlossen werden. Deswegen spricht man ja auch von der "potentiellen Beeinflussung." Man beachte, daß die Kausalrelation nur eine partielle Ordnung auf der Menge der Ereignisse ist; diejenigen Ereignisse, die nicht durch die Kausalrelation geordnet werden, sind gerade die unabhängigen Ereignisse.

Projiziert man die Ereignisse der Ausführung eines verteilten Algorithmus senkrecht auf eine gemeinsame horizontale Zeitachse, bekommt man eine totale Ordnung auf der Ereignismenge. Diese Ordnung entspricht der Abfolge der einzelnen Ereignisse in einer gedachten Globalzeit. Man kann zeigen, daß jede mögliche Permutation von Ereignissen innerhalb einer solchen Linearisierung, die mit der Kausalrelation verträglich ist, den Ausgang der Berechnung nicht verändert [136, Theorem 2.21]. Eine verteilte Berechnung enthält also viele mögliche Ausführungen. Abbildung 2.8 stellt dieselbe Berechnung dar wie die aus Abbildung 2.7, nur sind die Ereignisse anders angeordnet. Die Schlußkonfiguration der Berechnung ist bei beiden Abbildungen dieselbe, doch ist die Folge von Konfigurationen, durch die beide Berechnungen laufen, durchaus verschieden (beispielsweise taucht die Konfiguration γ gar nicht mehr in der Ausführung auf).

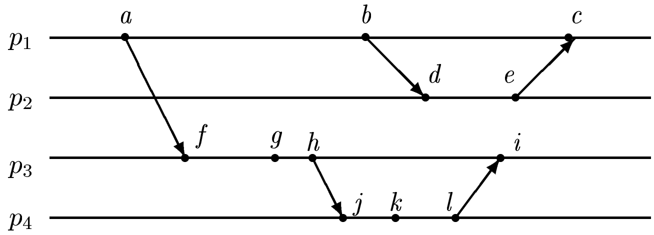

Abbildung 2.8: Ein anderer möglicher Ablauf der Berechung aus Abbildung 2.7 [136, S. 59].

Festzuhalten ist, daß eine Ausführung (oder ein Ablauf) eines verteilten Algorithmus einer Folge von Konfigurationen entspricht, daß aber eine verteilte Berechnung einer Menge von Ausführungen entspricht, nämlich allen, die die Kausalrelation der Ereignisse respektieren. Während es also durchaus Sinn macht, nach den Konfigurationen eines Ablaufes zu fragen, ist es unsinnig, nach den einzelnen Konfigurationen einer Berechnung zu suchen.

2.3.7 Fehler

Rechnernetze und verteilte Systeme bestehen aus einer Vielzahl von Hardware- und Software-Komponenten, die irgendwann auch mal ihren Dienst versagen [45, S. 57]. Die Fehlerproblematik in verteilten Systemen hat zwei Seiten: Je größer das Rechnernetz ist, desto größer ist auf der einen Seite die Wahrscheinlichkeit, daß zu einer bestimmten Zeit ein beliebiger Bestandteil des Systems ausfällt, doch auf der anderen Seite wird es immer unwahrscheinlicher, daß das Gesamtsystem insgesamt auf einen Schlag den Geist aufgibt. Um das Verhalten solcher Systeme auch in solchen Fällen modellieren zu können, muß das Modell um die Möglichkeit von Fehlern erweitert werden. Erst so wird es möglich, Aussagen über das Verhalten von Algorithmen im Fehlerfall zu treffen.

Von einem Fehler spricht man immer dann, wenn sich ein System nicht gemäß seiner Spezifikation verhält [45, S. 58].[5] Bisher wurden Fehler danach klassifiziert, wie sie sich äußerten: gebräuchliche Klassen sind beispielsweise *crash failure* (ein Prozeß reagiert nicht mehr), *send/receive omission* (nur eine Teilmenge der Nachrichten wird tatsächlich versandt oder empfangen) oder *byzantine failures* (beliebiges Fehlverhalten eines Prozesses). Hard- und Softwaresysteme wurden daraufhin untersucht, welche Fehlerklassen in ihnen auftreten konnten, und anschließend mit einer Fehlersemantik (*failure semantics* [45, S. 59]) ausgestattet, die das Systemverhalten im Fehlerfall beschreibt. Obwohl die vielen so entstandenen Fehlerklassen schon oft schematisiert und hierarchisiert worden sind [125], stehen sie trotzdem oft in einem ungewissem Verhältnis zueinander. Auch bestanden erhebliche Unsicherheiten, wie man das sporadische und unkontrollierte Auftreten von Fehlern modellieren sollte. Oft wird interessanterweise von einem “bösen Gegenspieler” (*malicious adversary* [135, S. 13]) ausgegangen, der andauernd auf seine Chance wartet, das System zu sabotieren. Tel hat gegen diese Sichtweise sowohl moralische als auch mathematische Bedenken [135, S. 13]. Und es gibt tatsächlich auch elegantere Möglichkeiten, “Fehler” in das Modell integrieren.

Den ersten Vorschlag dazu veröffentlichte Flaviu Cristian im Jahre 1985. In seinem Artikel “A rigorous approach to fault-tolerant programming” [44] betrachtet er Fehler als Operationen des Prozesses auf einem Zustandsraum, der durch genau definierte Fehlerzustände erweitert ist. Durch diesen “Trick” stehen die herkömmlichen Methoden des Schlie-

[5] Im Englischen existieren die Begriffe *failure*, *fault* und *error* [8, S. 396], die unterschiedlich verwendet werden. Cristian schreibt dazu: “[...]what one person calls a failure, a second person calls a fault, and a third person might call an error.” [45, S. 57] Im Deutschen gibt es ebenfalls verschiedene Terminologien [55].

ßens über die Korrektheit von Algorithmen auch zur Betrachtung von fehlerbehafteten Systemen zur Verfügung. Anish Arora hat diese Idee im Rahmen seiner Dissertation [14] in ein allgemeines mathematisches Modell der Fehlertoleranz eingebettet und bringt zur Verdeutlichung das folgende Beispiel.

2.14 Beispiel (Modellierung eines fehlerhaften Übertragungskanals [8, S. 397]) Ein einfacher Übertragungskanal wird durch den folgenden Automaten repräsentiert:

Die momentanen Ein- und Ausgabewerte des Übertragungskanals werden durch zwei Variablen *in* und *out* repräsentiert, die beide Werte aus der Menge $\{0, 1\}$ annehmen können. Zusätzlich existiert noch eine interne boolesche Variable *broken*, die einen fehlerhaften Zustand des Drahtes anzeigt. Das normale Verhalten des Kanals besteht darin, den Wert *out* auf den Wert *in* zu setzen, immer wenn $out \neq in$ und der Kanal $\neg broken$ ist, also:

process *data channel*
  var *in* : **input** $\in \{0, 1\}$
   out : **output** $\in \{0, 1\}$
  begin
    $out \neq in \wedge \neg broken \quad \longrightarrow \quad out := in$
  end

Nehmen wir an, im Fehlerfall ist der Kanal tot und überträgt nichts mehr, sendet auf *out* nur noch den Wert 0. Dies wird durch folgende hinzuzufügende Anweisung ausgedrückt:

$$broken \quad \longrightarrow \quad out := 0$$

Das plötzliche Eintreten eines Fehlers wird durch eine zusätzliche Anweisung modelliert:

$$\neg broken \quad \longrightarrow \quad broken := true$$

Falls der Kanal ab und zu wieder normales Verhalten an den Tag legt (Wackelkontakt), könnte man noch die folgende Zeile hinzunehmen:

$$broken \quad \longrightarrow \quad broken := false$$

Schließlich möchte man auch manchmal ein vollkommen zufälliges Verhalten eines Kanals modellieren (Ausgabewert springt ohne Grund zwischen 0 und 1 hin und her). Dies wird durch die folgende Anweisung erreicht:

$$broken \quad \longrightarrow \quad out := 1$$

Man beachte, daß das zufällige Wechseln der Zustände durch den Nichtdeterminismus der bewachten Anweisungen erreicht wird.

In dem vorangehenden Beispiel ist der Zustandsraum des Automaten um die Variable *broken* erweitert worden. Fehler sind nun immer Fehleroperationen auf diesem erweiterten Zustandsraum. Das “normale” Programm besteht nur aus den ersten beiden Anweisungen. Nimmt man die dritte Anweisung hinzu, bekommt man eine Art *crash failure* des Kanals, da er nach einem Fehler nie wieder sein normales Verhalten aufnimmt. Zusammen mit der vierten Anweisung hat man für die angeschlossenen Prozesse eine Art *send omission* realisiert, und nimmt man noch die letzte Anweisung hinzu, realisiert man eine Art *byzantine failure*.

2.3.8 Weitergehende Fragen

Die in den vorhergehenden Abschnitten angesprochenen Fragestellungen sind die für den weiteren Verlauf der Arbeit wichtigsten Gesichtspunkte, die bei der formalen Behandlung des Themas von Bedeutung sind. Bei der Behandlung der einzelnen Modelle sind aber noch ein paar weitergehende Fragen offengeblieben, deren Auswirkungen auf das Modell ich jetzt kurz ansprechen will.

Unendlich viele Prozesse? In den Modellen verteilter Algorithmen von Tel [136, Def. 2.5 und 2.6] (hier Definitionen 2.8 und 2.9), sowie den Formalisierungen verteilter Berechnungen von Schwarz und Mattern [127] und Charron-Bost, Mattern und Tel [38] sind die Zustandsmengen sowie Anzahl der Prozesse immer endlich. Im Gegensatz dazu geht Lynch in ihren Modellen [108, S. 201] immer auch von einer möglichen unendlichen Menge von Prozeßzuständen [108, S. 202] und von einer abzählbar unendlichen Menge von Teilautomaten aus [108, S. 208]. Während letzteres gut dazu geeignet ist, unbeschränkte Datenstrukturen (wie Zähler) oder unendlich laufende Prozesse zu modellieren, ist es doch eher unrealistisch und bringt beweistechnische Probleme mit sich [108, S. 207]. Praktische Systeme und Berechnungen sind zum Untersuchungszeitpunkt immer endlich und das dynamische Erzeugen von Prozessen kann notfalls dadurch simuliert werden, daß die benötigten Prozesse von vornherein “schlafend” im System anwesend sind und erst durch eine “Aktivierungsnachricht” in die aktuelle Berechnung eingebunden werden [127, Fußnote 1]. Ich sehe deshalb in der Beschränkung des Modells auf die Endlichkeit keine wesentliche Einschränkung der praktischen Ausdruckskraft.

Fairneß? In Definition 2.9 wurde das Verhalten eines verteilten Algorithmus mittels eines “zusammengesetzten” Automaten beschrieben: Zustandsübergänge auf den Einzelprozessen wurden zu Zustandsübergängen des Gesamtsystems. Diese Betrachtungsweise ist nur sinnvoll, wenn man davon ausgeht, daß jeder Prozeß eine faire Chance erhält, selbst in seinem Ablauf fortzuschreiten. Es macht keinen Sinn, Ausführungen zu betrachten, in denen eine kleine Gruppe von Prozessen andauernd ihre Zustände ändert und andere Prozesse, die ebenfalls bereit sind, nie zum Zuge kommen. Dieser Umstand wird mit dem Begriff der *Fairneß* bezeichnet [136, S. 49]. Fairneß bezieht sich demnach in gewisser Weise auf die relativen Ausführungsgeschwindigkeiten der Prozesse. Tel unterscheidet dabei zwischen schwacher und starker Fairneß.

2.15 Definition (Fairneß nach Tel [136, Def. 2.8]) Die Ausführung eines verteilten Algorithmus ist *schwach fair* (*weakly fair*), falls jedes Ereignis, welches in unendlich vielen aufeinanderfolgenden Konfigurationen möglich ist, immer auch in der Ausführung eintritt.

Die Ausführung ist *stark fair* (*strongly fair*), falls jedes Ereignis, welches in unendlich vielen Konfigurationen möglich ist, immer auch in der Ausführung eintritt.

Der Unterschied zwischen schwacher und starker Fairneß liegt in der Differenzierung von "immer" und "unendlich oft." Eine schwach faire Ausführung muß nur diejenigen Ereignisse beachten, die *immer* eintreten können. Eine stark faire Ausführung hingegen wird auch den Ereignissen Gehör schenken, die zwar nicht immer, aber trotzdem unendlich oft möglich sind.

Ich will in dieser Arbeit immer von fairen Ausführungen im letzteren Sinne ausgehen und den Begriff der Fairneß deshalb nicht explizit in das Modell aufnehmen. Wo es nötig ist, werde ich diesen Sachverhalt ansprechen.

Afek und Brown spechen in diesem Zusammenhang auch von einer Fairneß der Kommunikationskanäle (*fairness assumption on communication links* [2, S. 30]). Sie meinen damit die Tatsache, daß eine Nachricht, die über eine Kommunikationsleitung geschickt wird, entweder ankommen oder verlorengehen kann, daß aber, wenn man dieselbe Nachricht nur oft genug wiederholt, irgendwann einmal eine Kopie davon auf der anderen Seite ankommt. In den Modellen, in denen der Zustand des Kommunikationsnetzwerkes durch Multimengen modelliert wird, kommen Nachrichten immer an. Werden die Kanäle allerdings durch eigene Prozesse realisiert, die auch fehlerhaft sein können, so entspricht die von Afek und Brown geforderte Variante von Fairneß genau der Fairneß auf Prozessen, wie sie durch die Definition 2.15 gefaßt wird.

2.4 Verifikation verteilter Algorithmen

Ein formales Modell verteilter Systeme allein ist nicht wirklich nützlich. Es wird erst richtig "fruchtbringend," wenn man zeigen will, daß bestimmte Algorithmen in gewissen Modellen genau bezeichnete Eigenschaften besitzen. In diesem Abschnitt geht es um allgemeine Techniken der Verifikation, also des Nachweises bestimmter Eigenschaften verteilter Algorithmen [104, S. 1167].

2.4.1 Eigenschaften verteilter Algorithmen

Wenn man nachweisen will, daß ein Algorithmus eine gewisse Eigenschaft besitzt, muß man zuerst sagen, was eine "Eigenschaft" an sich eigentlich ist. Eigenschaften (*properties*) verteilter Systeme werden in der Regel durch Mengen von Ausführungen definiert. Mit Ausführungen eines verteilten Algorithmus sind hier unendliche Folgen von Konfigurationen gemeint. Endliche Ausführungen werden durch unendliche Wiederholungen ihres Terminierungszustandes ins Unendliche fortgesetzt. Mit einer teilweisen oder partiellen Ausführung (*partial execution*) meint man eine endliche Folge von Konfigurationen. Jede Konfiguration entsteht aus ihrem Vorgänger durch die Ausführung einer einzelnen aktiven bewachten Anweisung.

2.16 Definition (Eigenschaften als Menge von Ausführungen [4, S. 181]) Eine Eigenschaft (*property*) eines verteilten Algorithmus ist eine Menge $p = \{\alpha_0, \alpha_1, \dots\}$ von Ausführungen.

Ein verteilter Algorithmus beschreibt in natürlicher Weise selbst eine Menge von Ausführungen, nämlich genau die, die von seinen Startzuständen aus erreichbar sind. Ein Algorithmus hat (oder erfüllt) eine Eigenschaft p, gdw. die Menge von Ausführungen p', die er selbst beschreibt, in der Menge p enthalten ist.

Es gibt im wesentlichen zwei große Klassen von Eigenschaften, die sich durch die Techniken unterscheiden, wie sie nachgewiesen werden. Diese beiden Klassen werden mit den schwer übersetzbaren englischen Begriffen *safety* und *liveness* bezeichnet und wurden zuerst durch Leslie Lamport beschrieben [100].

2.4.2 Safety

Eine *safety*-Eigenschaft besagt salopp, daß irgendetwas Unerwünschtes nie eintreten wird ("Something bad will never happen."). Beispiele für Eigenschaften dieser Klasse sind der wechselseitige Ausschluß oder die herkömmliche partielle Korrektheit von sequentiellen Programmen.

2.17 Definition (Safety) Eine Eigenschaft $p = \{\alpha_0, \alpha_1, \dots\}$ ist eine *safety*-Eigenschaft, gdw. es ein nichttriviales ("wahr" und "falsch" sind in diesem Sinne trivial) Prädikat P auf Konfigurationen gibt, welches für jede Konfiguration α_i gilt.

Eine *safety*-Eigenschaft besagt also, daß ein gewisses Prädikat in jeder Konfiguration des Gesamtsystems (also immer) gilt [4, S. 182]. Nachgewiesen werden solche Eigenschaften in der Regel durch das Schließen mit Zusicherungen (*assertional reasoning*). Eine Zusicherung ist in diesem Zusammenhang ein Prädikat P auf Systemzuständen und man muß zeigen, daß dieses Prädikat durch die Aktionen des Algorithmus nicht verletzt wird. Für P benutzt man auch häufig den Ausdruck "Invariante."

Der Kern einer *safety*-Eigenschaft ist die Invariante P. Man könnte denken, daß sich alle Eigenschaften durch derartige Prädikate ausdrücken lassen und daraufhin die Definition 2.16 ändern. Doch dem ist nicht so.

2.4.3 Liveness

Eine *liveness*-Eigenschaft drückt aus, daß irgendwann etwas Erwünschtes eintreten wird ("Something good will eventually happen."). Das Paradebeispiel für eine solche Eigenschaft ist die Terminierung eines Algorithmus.

2.18 Definition (Liveness) Eine partielle Ausführung α eines Algorithmus ist *live* für eine Eigenschaft p, gdw. α in p unendlich fortgesetzt werden kann. Eine Eigenschaft $p = \{\alpha_0, \alpha_1, \dots\}$ ist eine *liveness*-Eigenschaft für einen Algorithmus A, gdw. jede partielle Ausführung von A *live* für p ist.

Informal bedeutet die Definition, daß keine partielle Ausführung "verloren" ist; es muß immer möglich sein, daß das besagte erwünschte Ereignis in Zukunft noch passieren wird.[6] Falls dies nicht mehr möglich ist, könnte man dies an einem gewissen Punkt festmachen und hätte so etwas Unerwünschtes beschrieben. Deshalb können *liveness*-Eigenschaften nie etwas Unerwünschtes beschreiben und sind in gewisser Weise also komplementär zu den *safety*-Eigenschaften.

Zwei Punkte sind hierbei bemerkenswert und grenzen *safety* und *liveness* voneinander ab [4, S. 182]: Zum einen legt die Definition nicht fest, was dieses erwünschte Ereignis eigentlich ist (es muß noch nicht einmal ein diskretes Ereignis sein). Zum anderen kann man mittels *liveness*-Eigenschaften nicht ausdrücken, daß "etwas Erwünschtes immer passiert."

Der Beweis von *liveness*-Eigenschaften erfolgt in der Regel analog zur Terminierung bei der sequentiellen Programmierung durch ein Wohlfundiertheitsargument. Dafür konstruiert man eine Funktion (die Terminierungsfunktion oder *norm function* [136, S. 53]) aus der Menge der Konfigurationen in eine wohlfundierte Menge (beispielsweise die natürlichen Zahlen) und zeigt, daß in jeder Ausführung die Funktion streng monoton fällt oder die geforderte Bedingung (zum Beispiel Terminierung) bereits gilt.

Zusammenhang zwischen safety und liveness. In ihrem Artikel "Defining Liveness" [4] zeigen Alpern und Schneider, daß jede nicht-triviale Eigenschaft eines Algorithmus als die Schnittmenge einer *safety*- und einer *liveness*-Eigenschaft ausgedrückt werden kann. Das bedeutet, daß zum

[6] Alpern und Schneider bemühen hier Cicero: "While there's life there's hope" [4, S. 182]

Nachweis der Korrektheit verteilter Algorithmen in der Regel zwei Teilbeweise notwendig sein werden.

2.5 Versuch einer Taxonomie der Modelle

In diesem Abschnitt möchte ich "aus der Vogelperspektive" auf das Kapitel zurückblicken und eine Taxonomie der Modelle skizzieren, wie sie eingangs von Abschnitt 2.3 bereits angesprochen wurde.

Prozesse als Automaten. Verteilte Systeme werden in der Regel als Mengen von parallel ablaufenden sequentiellen Prozessen dargestellt, die selbst wiederum als einfache Automaten modelliert werden. Das Gesamtsystem ist ein aus den einzelnen Prozeß-Automaten zusammengesetzter Automat, der durch das Voranschreiten seiner Komponenten seinen (Gesamt-) Zustand wechselt. Die einzelnen Zustandsänderungen der Prozesse erfolgen atomar und nicht gleichzeitig (bezogen auf eine gedachte Globalzeit). Die Prozesse werden durch bewachte Anweisungen sprachlich beschrieben.

Die Unterscheidungsmerkmale der einzelnen Modelle lassen sich meines Erachtens unter vier Stichworten subsummieren, von denen jedes im folgenden kurz besprochen werden soll: Zeit, Kommunikationsmechanismus, Fehlermodell und Problemstellung.

Zeit. Eines der beiden wesentlichen Unterscheidungsmerkmale der Modelle wurde bereits in Abschnitt 2.3 zu Beginn diese Kapitels angesprochen und betrifft die Vorstellung von Zeit, die in den Modellen vorhanden ist. Synchrone Systeme haben (durch ihr Wissen über die maximalen Zeitunterschied zwischen ihrer Uhr und der Uhr eines Nachbarn) eine Vorstellung von realer Zeit, asynchrone Systeme nicht. Oft findet man eine Dreiteilung dieses Spektrums, in denen die Systeme, die in synchronen Runden ablaufen, als "synchron" bezeichnet werden, und andere als *teilweise synchron* (*partially synchronous*) [108, S. 5]. Während der Entwurf von Algorithmen im so bezeichneten synchronen Modell einfacher ist als bei den anderen, ist das Modell auch am unrealistischsten. Im asynchronen Modell hingegen bringt das generelle Fehlen einer Vorstellung von Realzeit einige Probleme mit sich, dafür sind die darin entworfenen Algorithmen eher in realen Umgebungen einsetzbar (aber oft auch ineffizienter als nötig). Mit dem teilweise synchronen Modell kommt man der Realität am nächsten, doch sind Algorithmen darin noch schwieriger zu entwerfen: Man kann zwar das Wissen um die relativen Ablaufgeschwindigkeiten nutzen, um effizientere Verfahren zu konstruieren, doch reagieren diese Algorithmen unter Umständen sehr empfindlich auf die Verletzung auch nur einer ihrer zeitlichen Voraussetzungen [125, S. 20].

Kommunikationsmechanismen. Das zweite wesentliche Unterscheidungsmerkmal ist die Wahl des Kommunikationsmechanismus, welches auch das komplexeste ist. In diesem Kapitel wurden die zwei grundsätzlichen Arten, synchrone und asynchrone Kommunikation, bereits vorgestellt. Bei synchroner Kommunikation stellen Sende- und Empfangsereignis einen einzigen Zustandsübergang dar, während bei asynchroner Kommunikation beide Ereignisse getrennt stattfinden. Im letzteren Fall muß man oft noch zusätzliche Eigenschaften des Kommunikationskanals festlegen, beispielsweise ob die Nachrichten in der Reihenfolge ankommen, in der sie abgeschickt wurden (*first in first out* oder *FIFO property*) oder wieviele Nachrichten durch das Kommunikationsnetz gepuffert werden können (*message buffering*) [104, S. 1162]. Wie das vergangene Kapitel gezeigt hat, gibt es auch zahlreiche Alternativen, das Kommunikationsnetzwerk zu modellieren: als eigenständige Prozesse, also Ein/Ausgabe-Automaten oder als Punkt-zu-Punkt-Netzwerk durch Multimengen.

In der Literatur wird oft eine dritte Art der Kommunikation aufgeführt, nämlich die Kommunikation über Variablen, die einer Teilmenge von Prozessen gemeinsam ist (*local shared variables* [104, S. 1164f]). Dies ist eine Spezialisierung der bekannten Kommunikation über global gemeinsamen Speicher, die aber nicht im eigentlichen Sinne "verteilt" ist. Lamport und Lynch weisen darauf hin, daß man in diesem Zusammenhang aufpassen muß, konzeptuelle Aspekte ("what one says") und Sprachaspekte ("how one says it") nicht durcheinanderzubringen [104, S. 1164]. Die Kommunikation über gemeinsame Variable erfolgt durch Lesen oder Schreiben einer Speicherzelle, auf die mehrere Prozesse Zugriff haben. Konzeptuell ist dies eine einzelne Zustandsänderung, die jedoch auf der Empfängerseite keine sprachliche Realisierung (etwa in Form eines **receive**-Kommandos) hat. Diese Kommunikationsart müßte darum zur synchronen Kommunikation gezählt werden. Jedoch birgt die Form der sprachlichen Realisierung, also das Fehlen einer expliziten Empfangsanweisung, Probleme in sich.

Zum einen muß der Zugriff auf diese Variablen unter wechselseitigem Ausschluß geschehen. Der gegenseitige Ausschluß wird formal dadurch garantiert, daß die einzelnen Prozesse ihre atomaren Aktionen nie gleichzeitig ausführen. Diese Voraussetzung macht deutlich, wie unrealistisch selbst diese einfache Annahme aus den Voraussetzungen ist, die das automatentheoretische Modellieren so einfach gemacht hat. In der Realität verlaufen Zugriffe auf Variablen nie atomar und man muß den wechselseitigen Ausschluß in anderer Form realisieren. Die Anzahl von Prozessen, die parallel eine Aktion ausführen dürfen, wird deshalb auch oft gerne als Unterscheidungsmerkmal für Modelle herangezogen [90]. Wenn zwei bewachte Anweisungen nicht parallel ausgeführt werden dürfen, spricht man davon, daß sie sich *gegenseitig behindern* oder *beeinflussen* (*interfering guarded commands* [30, S. 845]). Statt wechselseitigen Ausschluß zu garantieren, kann man aber auch einfach dafür sorgen, daß sich die einzelnen Anweisung nicht gegenseitig beeinflussen, in dem Sinne, daß eine einmal ermöglichte bewachte Anweisung weiter möglich bleibt, egal welche Zustandsänderungen andere Prozesse durchführen (*noninterfering property*) [30, S. 845].

Eine andere Möglichkeit, wechselseitigen Ausschluß zu garantieren, besteht darin, die Kommunikation durch *Kommunikationsregister* (*communication registers*) zu realisieren [53, S. 4]. Kommunikationsregister sind lokale Variable eines Prozesses, auf die nur er selbst schreibend zugreifen darf und seine Kommunikationsnachbarn nur lesend. Hierdurch realisiert man wieder eine Art asynchrone Kommunikation, ohne Nachrichten modellieren zu müssen, und erlaubt wieder die gebräuchliche automatentheoretische Modellierung.

Kommunikation über gemeinsame Variable ist die Form von Kommunikation, die in den frühen Jahren der Informatik zur Synchronisierung paralleler Prozesse eingesetzt wurde und die relativ einfache Lösungen zu vielen Problemen erlaubt. Heutzutage ist es üblicher, eines der *message passing*-Modelle zu benutzen.

Fehlermodell. Ein weiteres, aber weniger oft genanntes Unterscheidungsmerkmal betrifft die Art von Fehlern, die durch das System modelliert werden. Hier gibt es eine ganze Reihe von Fehlerklassen, die ich in diesem Kapitel bereits kurz angerissen habe und die im Kapitel 4 nochmals aufgegriffen werden.

Problemstellung. Die eigentliche Problemstellung des Algorithmus hat auch Einfluß auf das Modell. Lamport und Lynch zählen beispielsweise die Topologie des Kommunikationsnetzwerkes (*network topology*) mit zu den Unterscheidungskriterien von *message passing*-Modellen [104, S. 1160]. Andere hiermit verbundene Aspekte sind beispielsweise:

- Die Anzahl der Prozesse im Netzwerk.
- Die Tatsache, daß Prozesse gleich aufgebaut sind und sie exakt denselben Algorithmus ausführen oder nicht.
- Die Möglichkeit, Knoten aufgrund eindeutiger Knotenidentitäten untereinander zu unterscheiden.

Nimmt man beispielsweise die Annahme ins Modell mit auf, daß Knoten keine eindeutigen Identitäten besitzen, sind viele Probleme nur mittels probabilistischer Algorithmen lösbar [136, S. 303ff]. Ein anderes Beispiel sind Traversierungsalgorithmen, die für bestimmte Topologien (wie einen n-dimensionalen Würfel) konstruiert worden sind und auch nur auf diesem funktionieren [113, S. 48f]. Das Modell hat also Einfluß auf die Problemstellung und umgekehrt.

Fazit. Wie man sieht, gibt es eine Vielfalt von Modellen, deren Taxonomie zwar noch nicht genau genug untersucht worden ist, unter denen man sich aber für seine Aufgabe gerade das richtige heraussuchen kann. Im bereits erwähnten Beitrag zum "Handbook of Theoretical Computer Science" schreiben Lamport und Lynch 1990 über die Möglichkeit, die verschiedenen Modelle gegeneinander abzugrenzen: "[...] we know of no simple alternative for classifying the standard models." [104, S. 1164] Darum will ich es bei diesen zusammenfassenden Betrachtungen belassen.

2.6 Ausblick

Ich habe in diesem Kapitel einen Überblick über die gesamte Bandbreite der Modelle verteilter Algorithmen gegeben und die wesentlichen Begriffe (lokaler/verteilter Algorithmus, synchrone/asynchrone Kommunikation, Ausführung eines Algorithmus, Berechnung, Kausalität, Eigenschaften) in ihrem Zusammenhang definiert. Ich habe erläutert, wie man Algorithmen formal beschreiben kann (durch bewachte Anweisungen) und wie sich zentrale Eigenschaften (*safety* und *liveness*) nachweisen lassen. Die beobachtete Vielfalt spiegelt die vielen verschiedenen Möglichkeiten wieder, von der Realität zu abstrahieren: Mit der Wahl eines Modells entscheidet man sich auch dafür, was im Zusammenhang mit den eigenen Untersuchungen relevant ist und was nicht. Wichtig ist nur, daß man sein Modell und die Algorithmen, die darin ablaufen, genau genug beschreibt und sich der inhärenten Probleme, die mit einer Darstellungsweise zusammenhängen, bewußt ist. Genau dies sollte dieses Kapitel nun ermöglichen. Nachdem also die formalen Grundlagen geklärt sind, ist die "Saat gesät" für eine fruchtbare Behandlung konkreter Problemstellungen in verteilten Umgebungen.

3

Replikation

Inhaltsübersicht

Kurzfassung

Seitdem es möglich ist, identische Kopien von gedruckten Texten, elektronischen Verzeichnissen oder handlichen Gebrauchsgegenständen herzustellen, hat man stark frequentierte Dienstleistungen (wie etwa die Telefonauskunft) mehrfach zur Verfügung gestellt (beispielsweise als Telefonbuch), um den großen Ansturm der Anfragen zu meistern. In der Fachsprache spricht man hier von "Replikation." Replikation wird insbesondere auch in Computersystemen eingesetzt, um eine Vielzahl von Auskunfts- und Informationsdiensten zu realisieren. Beispiele hierfür sind Telefonverzeichnisse oder Kataloge auf CD-Rom, elektronische Tageszeitungen und Software-Archive. In diesem Kapitel wird Replikation als Methode der Informatik vorgestellt und ihre inhärenten Vor- und Nachteile beschrieben. Dabei wird deutlich, daß manche Anwendungen ohne Replikation nicht realisierbar wären und daß gerade in diesen Fällen Aspekte der Fehlertoleranz eine wichtige Rolle spielen.

3.1 Einführung

Das Verbum "replizieren" hat im allgemeinen Sprachgebrauch eine diffuse Bedeutung, die im wesentlichen durch die Inhalte verschiedener populärer Science Fiction-Filme geprägt ist. In ihnen tauchen beispielsweise extraterrestrische Kampfroboter auf, die sich kurz vor ihrer Zerstörung plötzlich wie bei einer Zellteilung verdoppeln ("er hat sich repliziert!") und mit unverminderter Kraft weiterkämpfen; andere außerirdische Lebensformen können durch eine künstlich herbeigeführte ungeschlechtliche Vermehrung identische Kopien ihrer selbst (oder anderer Mitlebewesen) herstellen ("sie stellen Replikate ihrer selbst her!"), was man neudeutsch auch als "klonen" bezeichnet. Ursprünglich stammt der Begriff "Replikation" aus dem Lateinischen. Dort bedeutet *replicare* soviel wie "wieder aufrollen" und *replica(tio)* "Wiederholung." Replikation, als das Bereitstellen mehrerer identischer Objekte, ist eine recht gebräuchliche Problemlösungsstrategie, wie das folgende Beispiel verdeutlicht.

3.1 Beispiel Ein einfaches Beispiel für "Replikation" ist ein Telefonbuch, in dem alle Fernsprechteilnehmer einer bestimmten Region zusammen mit ihrer Rufnummer aufgelistet sind. In der Sprache der Informatik ist dieses Verzeichnis ist ein "Auskunftsdienst" der Telefongesellschaft in schriftlicher Form. Durch ihn wird ermöglicht, die Telefonnummer eines anderen Teilnehmers herauszufinden und ihn anzurufen. Das Erstellen von Telefonbüchern ist eine sehr aufwendige Angelegenheit: Die Mühen beginnen bei der Sammlung und Pflege der Daten eines zentralen (elektronischen) Verzeichnisses und umfassen außerdem den Satz, den Druck und die Verteilung des Werkes. Der Aufwand ist jedoch gerechtfertigt: Würden alle Auskunftswünsche der Kunden durch *eine einzige* Auskunftstelle erledigt werden, oder läge an einer bestimmten Stelle nur *ein einziges* Verzeichnis aus, könnte diese zentrale Stelle den Ansturm nicht meistern. Die Nachteile der Realisierung einer Auskunft durch ein Telefonbuch liegen aber auf der Hand: Rufnummern ändern sich, Anschlüsse werden neu eingerichtet oder gesperrt. Ein Telefonbuch ist quasi schon bei seiner Auslieferung veraltet.

Wie das Beispiel verdeutlicht, ist Replikation eine in der Welt häufig anzutreffende Methode, um stark frequentierte Dienste zu realisieren.

Ausblick auf das Kapitel. In diesem Kapitel will ich die Methode der Replikation aus Sicht der Informatik beleuchten. Genauer werde ich zunächst erläutern, was man in der Informatik unter Replikation versteht, wo man sie als Methode einsetzt und was die Vor- und Nachteile dieser Technik sind. In der Forschungsliteratur gibt es eine große Anzahl von Replikationsalgorithmen verschiedenster Couleur, die alle ihre speziellen Einsatzgebiete und ihre eigenen Vor- und Nachteile besitzen und deren Vorstellung Bände füllen könnte. Ich muß mich darum in dieser Arbeit auf einen kleinen Teilbereich des Gesamtspektrums beschränken.

Replikation von Diensten. Mit dem Aufkommen des globalen Hypertext-Systems *World Wide Web* (WWW) ist der Bedarf an weltumspannenden elektronischen Diensten stark angewachsen. Der Begriff "Dienst" ist dabei eine in der Branche übliche Kurzform für "Dienstleistung." Das Spektrum derartiger Dienste ist riesengroß und umfaßt eine Vielfalt an Auskunfts- und Informationsdiensten. Hier ein paar bekannte Beispiele, um nur einige zu nennen:

- online-Telefonbücher und online-Versandhauskataloge,
- Suchmaschinen für das WWW,
- öffentliche und private Diskussionsforen (*bulletin boards*, *mailing lists*),
- elektronische Tageszeitungen und
- weltweite verteilte Bibliotheken und Software-Archive.

Konkret liegt der Schwerpunkt dieses Kapitels also auf der Replikation von Diensten in Weitverkehrsnetzen wie dem weltumspannenden Internet. Weil es sich gut dafür eignet, will ich das oben begonnene Beispiel der Telefonauskunft als laufendes Beispiel in den verschiedenen Unterabschnitten dieses Kapitels weiterführen.

3.2 Vor- und Nachteile von Replikation

In der Informatik bezeichnet man mit Replikation die Bereitstellung mehrerer identischer Kopien eines Datenelementes an verschiedenen Stellen in einem verteilten System. Es gibt zwei wesentliche Gründe, um Replikation einzusetzen:

- Steigerung der Leistung (*performance*) eines Dienstes
  Durch die Bereitstellung mehrerer Originale kann der Dienst mehr Anfragen in derselben Zeit beantworten. Außerdem ist es möglich, durch den Zugriff auf "nahe" Kopien, die "Zugriffskosten" erheblich zu reduzieren.
- Steigerung der Verfügbarkeit (*availability*) eines Dienstes
  In verteilten Systemen kann es immer vorkommen, daß Rechnerknoten ausfallen oder Nachrichten verlorengehen können. Durch das Vorhandensein mehrerer identischer Kopien steigt in solchen Fällen die Wahrscheinlichkeit, daß ein Benutzer noch auf den Dienst zugreifen kann.

Im Normalfall werden zu einem bestimmten Zeitpunkt alle im System vorhandenen Replikate den gleichen Wert (also den aktuellen Stand des Dienstes) besitzen. In einem solchen Fall spricht man von einem *konsistenten* Zustand des Systems. Die Notwendigkeit, den Wert des Datenobjektes auch einmal zu ändern, deckt die beiden miteinander verwobenen nachteiligen Eigenschaften der Replikation auf [25, S. 1f]. Diese sind:

- ein erhöhter Änderungsaufwand
  Da das replizierte Datenobjekt in der Regel nicht nur auf einem, sondern auf mehreren Knoten aktualisiert werden muß, führt jede Veränderung seines Wertes einen erhöhten Aufwand mit sich.

- die Gefahr von Inkonsistenzen
  Die Verteilung eines neuen Wertes auf alle Replikate kann durch Netzpartitionierungen oder Probleme bei der Nachrichtenübertragung fehlschlagen und somit zu einem inkonsistenten Zustand führen. Lesezugriffe auf bestimmte Datenobjekte können somit unerwünschte (weil veraltete) Informationen liefern.

Die Vor- und Nachteile von Replikation lassen sich gut anhand des Telefonbuch-Beispiels (Beispiel 3.1) verdeutlichen, welches hier fortgeführt wird.

3.2 Beispiel (Vor- und Nachteile der Replikation mittels eines Telefonbuchs) Die Vorteile eines Telefonbuches wurden bereits in Beispiel 3.1 deutlich: Eine einzige Auskunftsstelle könnte die Zahl der Anfragen nicht verkraften und ein Ausfall dieser einen Position könnte den gesamten Dienst lahmlegen. Die Realisierung des Dienstes mittels gedruckter Telefonbücher ist für die Telefongesellschaft sehr teuer, für den Telefonteilnehmer ist die Inanspruchnahme des Dienstes jedoch sowohl in zeitlicher als auch in pekuniärer Hinsicht sehr “billig” (es geht schnell und kostet nichts). Leider ist der Informationsstand, den ein Telefonbuch repräsentiert, nicht immer aktuell. Änderungen von Teilnehmerdaten resultieren schnell in einer Änderung der zentralen Auskunftskartei (welche die Grundlage für den Druck des Telefonbuches ist). Somit ist das “Auskunftssystem” praktisch immer in einem inkonsistenten Zustand. Durch die regelmäßige Neuausgabe versucht man, dieses Problem etwas zu abzumildern.

3.3 Beispiel (Vor- und Nachteile der Replikation mittels einer Telefonauskunft) Eine andere Art, diesen Auskunftsdienst zu realisieren, ist die telefonische Auskunft. Sie wird durch eine bundesweit einheitliche Rufnummer realisiert, über die man mit freundlichen Mitarbeitern der Telefongesellschaft in Kontakt treten kann, die eine Auskunft direkt aus der zentralen Informationsbasis der Gesellschaft beziehen können. Dieser Dienst ist ebenfalls repliziert: Es gibt derartige Auskunftsstellen in fast jeder größeren Stadt und selbst innerhalb dieser Auskunftsstellen gibt es zumeist mehrere “Plätze”, die parallel Anfragen bearbeiten können.

Die Aktualität der Daten muß ein Kunde jedoch mit drastisch erhöhten Auskunftsentgelten bezahlen. Allerdings ist dieser Dienst auch dann noch verfügbar, wenn die "lokale Kopie" der Auskunft nicht erreichbar ist: Man kann in solchen Fällen einfach die Vorwahl des Nachbarortes benutzen und ein anderes "Replikat" des Dienstes erreichen. Dadurch ist er praktisch so lange verfügbar, wie das Telefon funktioniert.[1]

3.3 Starke und schwache Konsistenz.

In der Praxis gibt es eine wesentliche Unterscheidung von replizierten Diensten in solche mit *starker* und solche mit *schwacher Konsistenz* (*strong* und *weak consistency*) [73, S. 6f]. Stark konsistente Replikationsschemata haben die Eigenschaft, daß das Gesamtsystem sich aus der Sicht der Anwendung wie ein nicht-repliziertes System verhält [25, S. 1][24, S. 30]. Dies wird formalisiert durch den Begriff der *1-Kopie-Serialisierbarkeit* oder der *1-Kopie-Äquivalenz.*

3.4 Definition (1-Kopie-Äquivalenz und 1-Kopie-Serialisierbarkeit [25, S. 2f]) Sei $T = (t_1, t_2, \dots)$ eine Folge von Lese- und Schreiboperationen auf ein repliziertes Datenelement (*schedule* [25, S. 2] oder *history* [24, S. 28f]). Die Folge T heißt *seriell* (*serial*), gdw. für je zwei Operationen t_i und t_j aus T gilt, daß alle Teiloperationen (also die Zugriffe auf die einzelnen Replikate) von t_i vor den Teiloperationen aus t_j durchgeführt werden.

Die Folge T heißt *serialisierbar* (*serializable*), gdw. es mindestens eine serielle Folge mit denselben Lese- und Schreiboperationen wie in T gibt, die den gleichen Endzustand der einzelnen Replikate erzeugen wie T.

Die Folge T von Lese- und Schreiboperationen auf ein repliziertes Datenelement r heißt *1-Kopie-serialisierbar* (*1-copy-serializable*), gdw. es mindestens eine serialisierbare Folge mit denselben Lese- und Schreiboperationen wie in T für nur ein einziges Replikat R gibt und R am Ende den gleichen Zustand hat wie der, der durch eine Leseoperation auf r nach der Ausführung von T geliefert wird.

Die 1-Kopie-Serialisierbarkeit bringt naturgemäß einen hohen Aufwand bei der Änderung des Datenbestandes mit sich. Der Aufwand läßt sich drastisch verringern, wenn man ein begrenztes Maß an Inkonsistenz zuläßt unter der Voraussetzung, daß das System irgendwann wieder in einen konsistenten Zustand übergeht. In solchen Fällen spricht man von Systemen mit schwacher Konsistenz [73, S. 1].

Starke Konsistenz ist immer dann notwendig, wenn ein Zugriff auf veraltete Daten dem Zweck der Anwendung widersprechen würde. Beispielweise wäre eine inkonsistente Sicht auf den eigenen Kontostand bei der Bank oft unerwünscht: In der Regel möchte man vor einer Abhebung im-

[1] Und wenn das Telefon nicht mehr funktioniert, hat sich der Bedarf an einer Telefonauskunft schnell erledigt.

mer wissen, wieviel Geld man noch auf dem Konto hat (um beispielsweise die Zahlung teure Überziehungszinsen zu vermeiden). Anwendungen der 1-Kopie-Serialisierbarkeit (und damit auch der starken Konsistenz) findet man also in allen Formen von Datenbanken, die Datenbestände an verschiedenen Stellen replizieren (den sogenannten *verteilten Datenbanken* [46]).

Schwache Konsistenz und Weitverkehrsnetze. Starke Konsistenz läßt sich in der Regel nur durch einen aufwendigen Mechanismus erreichen, in dem Änderungen des Datenbestandes in Form von *Transaktionen* (*transactions*) atomar und für alle Beteiligten einheitlich erfolgen [109]. In verteilten Systemen benötigen die entsprechend verteilten Protokolle normalerweise die Zustimmung der Mehrheit, wenn nicht sogar aller funktionierenden Stationen, die Replikate halten. Naturgemäß sind derartige Algorithmen sehr anfällig gegenüber unzuverlässigen Kommunikationsleitungen oder Netzwerkpartitionierungen [73, S. 7]. Im letzteren Fall kommt es sogar vor, daß das gesamte System blockiert, bis alle replizierten Datenelemente wieder an einem gemeinsamen Kommunikationsnetzwerk teilhaben können.

Kommunikation in Weitverkehrsnetzen ist im Vergleich zu lokalen Netzwerken notorisch unzuverlässig. Eine Studie von Golding [72] zeigt, daß der Austausch von Nachrichten über die Grenzen von Kontinenten hinweg im Bereich von mehreren hundert Millisekunden liegen kann, bei Verlustraten von 40 Prozent und mehr. Long, Carroll und Park [107] stellen fest, daß die Verfügbarkeit von Rechnern im Internet bei etwa 90 Prozent liegt, wobei die Zeit zwischen zwei Ausfällen durchschnittlich zehn Tage beträgt. Diese Daten zeigen, daß starke Konsistenz in Weitverkehrsnetzen ein sehr aufwendiges Unterfangen ist. Schwache Konsistenz ist hingegen — wie die Algorithmen aus Abschnitt 3.6 zeigen werden — in durchaus effizienter Weise realisierbar.

3.4 Replikation und Gruppenkommunikation

Man kann Replikation in allen seinen Facetten einfach erreichen, wenn man einen *Gruppenkommunikationsmechanismus* zur Verfügung hat [73, S. 5]. Ein solcher Mechanismus hält Methoden bereit, mit denen man entweder eine Nachricht an die Gesamtheit aller Prozesse (*broadcast*) oder nur an eine Teilmenge von ihnen senden kann (*multicast*). Die Teilmenge von Prozessen, an die im letzteren Fall die Nachricht versandt wird, nennt man auch eine *multicast*-Gruppe.

Empfang und Zustellung. Bei der Spezifikation der Funktionalität von *broadcast*- oder *multicast*-Primitiven macht man gewöhnlich eine Unterscheidung zwischen dem Empfang und der Zustellung einer Nachricht. Im

Englischen wird dies durch die beiden Begriffe *receive* (Empfangen) und *deliver* (Zustellen) ausgedrückt. Die Unterscheidung zielt auf unterschiedliche Ebenen einer Kommunikationshierarchie, die zur Strukturierung der komplexen Kommunikationsaufgaben [134, S. 10ff] eingeführt wurden. Dabei werden die verschiedenen Aufgaben (beispielsweise Übertragung einzelner Bits, Austausch von ganzen Bytes, Adressierung, Fehlererkennung) einzelnen Schichten einer siebenstufigen Hierarchie zugeordnet, an deren Sockel die physikalische Übertragung eines Signals über ein Transportmedium und an deren Spitze die konkrete Anwendung steht [134, S. 15ff].

Einen Gruppenkommunikationsmechanismus wird man auf einer Schicht unterhalb der Anwendungsebene anordnen. Die Anwendung selbst kann direkt durch den Aufruf von bereitgestellten Primitiven den Versand von Nachrichten anordnen und wird auf der anderen Seite durch die Kommunikationsschicht informiert, wenn eine neue Nachricht eingetroffen ist. Wird ein solches Nachrichtenpaket an die Anwendungsschicht ausgeliefert, spricht man von der *Zustellung* der Nachricht. Von dem *Empfang* einer Nachricht spricht man hingegen dann, wenn die Gruppenkommunikationsschicht eine Nachricht von der ihr unterliegenden Schicht (zum Beispiel dem Netzwerk) erhält (vergleiche auch Abbildung 3.1). Der Unterschied in der Terminologie ist durchaus wichtig, da der Empfang nicht sofort und automatisch auch die Zustellung einer Nachricht zur Folge hat. Beispielsweise wird die Kommunikationsschicht bestimmte Nachrichten eine zeitlang puffern, um noch auf logische Vorgänger zu warten. Doppelt empfangene Nachrichten wird sie gar nicht an die Anwendung zuzustellen, wenn sie bereits ein Mal zugestellt worden sind.

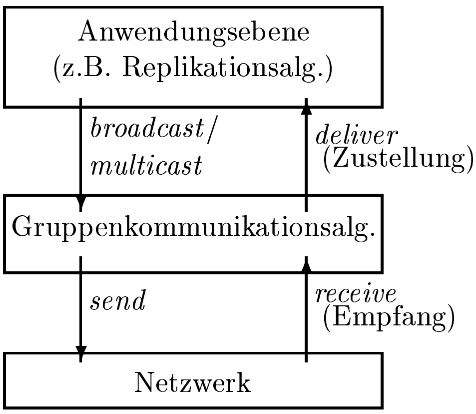

Abbildung 3.1: Anordnung der Gruppenkommunikation in der Kommunikationshierarchie und ihre Schnittstellen.

Chancen und Probleme der Gruppenkommunikation. Gruppenkommunikationsmechanismen sind nicht nur für die Realisierung von Replikationsverfahren nützlich. Generell sind sie ein gutes Hilfsmittel in verteilten

Systemen, damit sich eine Anzahl von Prozessen auf einen gemeinsamen Wert einigen, der von ihrem momentanen Zustand abhängt [80, S. 97]. In der englischsprachigen Literatur spricht man hier von dem Problem des *distributed consensus* [139]. Aus diesem Grunde sind zuverlässige *broad-* und *multicast*-Mechanismen bereits ausführlich untersucht [80, 105] und erfolgreich implementiert worden [28]. In ihrem Beitrag "fault-tolerant broadcasts and related problems" [80] zu einem Sammelband [118] haben Hadzilacos und Toueg einen verdienstvollen Versuch unternommen, die bereits bestehende Literatur zu diesem Thema zu sichten und sie in strukturierter Weise zusammenzufassen. Wichtig sind im Zusammenhang dieser Arbeit die Spezifikationen unterschiedlicher Arten von zuverlässigem *broadcast* sowie die vorgestellten Zusammenhänge der entsprechenden Algorithmen.

Informal müssen bei einem solchen *broadcast* alle funktionierenden Prozesse dieselbe Menge von Nachrichten zustellen, und diese Nachrichten sind genau diejenigen, die von funktionierenden Prozessen durch den Aufruf des *broadcast*-Primitives abgeschickt wurden (insbesondere dürfen Nachrichten pro funktionierendem Prozeß nur ein Mal zugestellt werden).

3.5 Definition (zuverlässiger (zeitbeschränkter) *broadcast* [80, S. 104]) Ein zuverlässiger *broadcast* ist ein Kommunikationsmechanismus (gegeben durch die beiden Primitive **broadcast** und **deliver**), welcher die folgenden drei Bedingungen erfüllt:

- Wenn ein korrekter Prozeß **broadcast**(m) aufruft, dann werden alle anderen korrekten Prozesse garantiert **deliver**(m) ausführen (*validity*).
- Wenn ein korrekter Prozeß **deliver**(m) ausführt, dann werden alle anderen korrekten Prozesse garantiert **deliver**(m) ausführen (*agreement*).
- Nur Nachrichten, auf denen zuvor **broadcast**(m) ausgeführt wurde, werden tatsächlich zugestellt und dies maximal ein Mal (*integrity*).

Ein zuverlässiger *broadcast* ist ein *zeitbeschränkter zuverlässiger broadcast* (*timed reliable broadcast* [80, S. 107] oder *bounded reliable broadcast* [73, S. 25]), gdw. der *broadcast* nach einer festen, endlichen Zeit garantiert abgeschlossen ist.[2]

Im Gegensatz zum *broadcast* muß man beim *multicast* noch zusätzlich beschreiben, wie die Gruppenzugehörigkeit und die damit verbundenen Ein- und Austrittssemantiken aussehen. Problematisch sind in diesem Zusammenhang Gruppenüberlappungen, da unterschiedliche Ordnungsgarantien auf der Zustellung der Nachrichten (die weiter unten noch besprochen werden) jeweils andere Semantiken haben, wenn man sie global

[2] Für den zeitbeschränkten zuverlässigen *broadcast* kann man noch die Unterscheidung machen, ob die Zeitschranke gemäß einer gedachten Globalzeit oder gemäß der lokalen Zeit der Prozesse gilt [80, S. 107f]. Ich werde immer die erstere Variante benutzen.

für das System oder lokal für die Gruppe betrachtet (das Problem der *hidden channels*). Eine Betrachtung dieses Problems würde jedoch an dieser Stelle zu weit führen.

Replikation und Gruppenkommunikation. Zwischen Replikationsalgorithmen und Gruppenkommunikation besteht ein enger Zusammenhang. Wie oben schon angesprochen kann man mittels eines *multicast* in relativ einfacher Art und Weise die Replikation von Datenelementen implementieren, indem man die Prozesse mit den Replikaten in einer *multicast*-Gruppe zusammenfaßt. Entsprechend der Zuverlässigkeit und der Ordnungsgarantien des *multicast*-Primitives ergeben sich hieraus verschiedene Grade von Konsistenz, die weiter unten noch angesprochen werden. Auf der anderen Seite implementiert man bei der Realisierung von Replikationsalgorithmen auch immer eine Art Gruppenkommunikationsmechanismus, der jedoch seiner Aufgabe entsprechend mehr leistet als ein reiner *multicast* und durch eine spezialisierte Anpassung auf das Problem unter Umständen auch effizienter gestaltet werden kann.

In seiner Dissertation [73] beschreibt Richard Golding ein flexibles Rahmenwerk (*framework*) für einen Gruppenkommunikationsmechanismus, das je nach Anforderung die gesamte Bandbreite der Replikationsalgorithmen implementieren kann [73, S. 16ff]. Gruppenkommunikationsprimitive lassen sich demnach mittels drei Eigenschaften "parametrisieren":

- Zuverlässigkeit der Zustellung (*delivery reliability*)
  Wie zuverlässig ist die Zustellung der *multicast*-Nachricht? Mögliche Ausprägungen dieses Parameters sind in Tabelle 3.1 dargestellt.

- Latenz der Zustellung (*delivery latency*)
  Laut Duden spricht man von "Latenz," wenn etwas bereits vorhanden aber noch nicht in Erscheinung getreten ist. In diesem Falle bezieht sich dies auf die Nachrichten und meint, wie lange der Versand und letztendlich auch die Zustellung der *multicast*-Nachricht dauert. Tabelle 3.2 enthält mögliche Grade dieser Eigenschaft.

- Ordnung der Zustellung (*delivery ordering*)
  In welcher Ordnung werden die *multicast*-Nachrichten zugestellt? Gemäß den Ausführungen weiter oben gibt es eine ganze Reihe von verschiedenen Zustellungsordungen, die in Tabelle 3.3 zusammengefaßt sind.

Bemerkung zur Latenz. Wie aus Tabelle 3.2 deutlich wird, bestimmen zwei unabhängige Faktoren die Latenz bei der Zustellung von *multicast*-Nachrichten: Die Verzögerung, mit der die Zustellung gestartet wird, und die Zeitdauer, in der die Zustellung auf allen funktionierenden Knoten abgeschlossen ist. Die vier Einträge aus Tabelle 3.2 ergeben sich aus den Kombinationsmöglichkeiten beider Faktoren [73, S. 26].

Ein verzögerter Beginn der Zustellung hat großen Einfluß auf die Fehlertoleranzeigenschaften eines Algorithmus. Nachrichten können senderseitig in einem Puffer zwischengespeichert werden und genau dann den Eingang in den Zustellungsmechanismus finden, wenn die Umstände günstig sind. Wenn beispielsweise ein mobiler Rechner zeitweise keine Verbindung zum Gesamtnetz hat, muß er die Nachrichten zurückhalten, bis er wieder an das Netz angeschlossen ist, oder er kann sie gemäß einer Priorität verzögern, bis eine entsprechende kostengünstige Kommunikation (über Funkmodem, durch eine lokale Infrarot-Übertragung oder ein paralleles Kabel) möglich ist [73, S. 26].

Vergleichbar ist die Anfangsverzögerung mit der Zustellung eines Briefes mit der Post. Der Zustellungsprozeß beginnt in dem Zeitpunkt, zu dem der Brief mit dem Poststempel versehen wird. Wenn man einen Brief in den Briefkasten wirft, hat die Zustellung somit noch nicht begonnen. Ein Postbeamter kann auswählen, welche Briefkästen er zuerst leert und bearbeitet. Wenn man allerdings den Brief direkt in die Postfiliale bringt, wird in der Regel der Poststempel sofort angebracht und der Beginn der Zustellung erfolgt ohne Verzögerung.

vollständig (*atomic*)	Nachricht wird entweder an alle Gruppenmitglieder oder an keines zugestellt. Nachricht wird auch nicht zugestellt, wenn mindestens ein Prozeß aus der Gruppe fehlerhaft ist.
zuverlässig (*reliable*)	Nachricht wird entweder an alle funktionierenden Gruppenmitglieder ausgeliefert oder an keines von ihnen. Wenn der Sender fehlerhaft ist, gibt es keine Garantien über die Zustellung.
k-zuverlässig (*Quorum*)	Nachricht wird an mindestens k funktionierende Gruppenmitglieder ausgeliefert. Wenn der Sender fehlerhaft ist, gibt es keine Garantien über die Zustellung.
bestmöglich (*best effort*)	Keine Garantien über die Zustellung der Nachricht an die Gruppenmitglieder, bis auf das Wissen, daß pro Prozeß ein Zustellversuch unternommen wird.

Tabelle 3.1: Verschiedene Grade der Zuverlässigkeit bei der Zustellung einer *multicast*-Nachricht [73, S. 25].

***broadcast*-Transformationen.** Der in Definition 3.5 vorgestellte zuverlässige *broadcast* enthält noch keine Garantien für die Zustellungsreihenfolge der Nachrichten sowie die zeitlichen Grenzen, an die sich ein Algorithmus zu halten hat. Wie in Tabelle 3.3 bereits angedeutet, sind die unterschiedlichen Grade der Zustellungsordnung sind in gewisser Weise modular. Konsequenterweise zeigen Hadzilacos und Toueg [80, S. 115ff], wie man mittels einfacher Erweiterungen aus einem gegebenen

sofort (*synchronous*)	Die Zustellung beginnt sofort und ist nach beschränkter Zeit garantiert beendet.
unmittelbar (*interactive*)	Die Zustellung beginnt sofort und ist nach einer festen, aber unbeschränkten Zeit garantiert beendet.
zeitbeschränkt (*bounded*)	Die Zustellung beginnt erst nach einer gewissen Wartezeit, ist aber nach einer beschränkten Zeit garantiert beendet.
allmählich (*eventual*)	Die Zustellung beginnt erst nach einer gewissen Wartezeit und ist nach einer festen, aber unbeschränkten Zeit garantiert beendet.

Tabelle 3.2: Verschiedene Grade der Latenz bei der Zustellung einer *multicast*-Nachricht [73, S. 26].

keine (*unordered*)	Die Zustellung erfolgt ohne jede bestimmte Ordnungsgarantie.
FIFO	Die Zustellung von Nachrichten eines Senders an die Empfänger erfolgt in der Absendereihenfolge.
kausal (*causal*)	Die Zustellung erfolgt unter Wahrung der Kausalrelation (Definition 2.13) unter den Nachrichten (kausale Ordnung ist immer auch *FIFO*-Ordnung).
total (*total*)	Die Zustellung der Nachrichten erfolgt bei allen Prozessen in derselben Ordnung.
total, *FIFO*	Kombination von totaler und *FIFO*-Ordnung.
total, kausal	Kombination von totaler und kausaler Ordnung.

Tabelle 3.3: Grade der Ordnung bei der Zustellung von *multicast*-Nachrichten [73, S. 29], vergleiche auch die Ausführungen auf Seite 49.

zuverlässigen *broadcast* ohne Ordnungsgarantien folgende Transformationen durchführen kann:

- *(timed) reliable broadcast* $\rightarrow$ *(timed) atomic broadcast*
- *(timed) (atomic) broadcast* $\rightarrow$ *(timed) (atomic) FIFO broadcast*
- *(timed) (atomic) FIFO broadcast* $\rightarrow$ *(timed) (atomic) causal broadcast*

Golding hat diese Transformationen bereits in seiner Dissertation vorausgeahnt [73, S. 75f] und sie sich in abgewandelter Weise zunutze gemacht. Sie bedeuten nämlich, daß man jeweils nur einen einfachen (und "ungeordneten") zuverlässigen *broadcast* implementieren muß und durch die Transformationen daraus automatisch alle anderen Arten von Zustellungsordnungen erzeugen kann.

3.5 Einsatzgebiete von Replikation

Im folgenden will ich einige Anwendungen und Einsatzgebiete für Replikation aufführen und sie jeweils durch ihren Platz im "Raum" der Gruppenkommunikationsmechanismen charakterisieren. Der Zweck dieser Unternehmung liegt darin, die in der Einleitung dieses Kapitels erwähnten Dienste genauer zu spezifizieren und einen Eindruck ihrer Systemanforderungen zu vermitteln.

3.5.1 Verteilte Datenbanken

Verteilte Datenbanken [46] wurden bereits oben im Zusammenhang mit dem Begriff der 1-Kopie-Serialisierbarkeit erwähnt. In ihnen fordert man in der Regel starke Konsistenz mit allen ihren Nebeneffekten. Es gibt eine Vielzahl an Protokollen, die nicht nur im Rahmen verteilter Datenbanken 1-Kopie-Serialisierbarkeit garantieren [25]. Im wesentlichen geht es dabei immer darum, wechselseitigen Ausschluß und Atomizität der Schreiboperationen auf das Replikat zu garantieren.

Zentralistische Ansätze. Einfache "absolutistische" [25, S. 5] Verfahren garantieren die Serialisierbarkeit dadurch, daß es eine Art Primärstation im Netzwerk gibt, die alle Schreibzugriffe auf das replizierte Datenobjekt quasi "kanalisiert." Will ein Prozeß das Replikat verändern, benötigt es die Zustimmung der Primärstation, und die Primärstation achtet darauf, daß zu jedem Zeitpunkt nur maximal eine andere Station das Schreibrecht besitzt. Beim Schreiben muß jedoch auch darauf geachtet werden, daß wirklich *alle* Replikate auf den neuesten Stand gebracht werden. Dieser Lösung haften jedoch alle diejenigen nachteiligen Eigenschaften an, die man auch bei anderen "zentralistischen" Lösungen in verteilten Systemen beobachten kann: Die zentrale Stelle entwickelt sich mit der Zeit zum Engpaß und die Verfügbarkeit des Gesamtsystems hängt stark von der Verfügbarkeit der ausgezeichneten Stelle ab [25, S. 5].

Verteilte Ansätze. Im Gegensatz zu den zentralistischen Ansätzen geht es bei den verteilten Protokollen darum, beim Lese- und Schreibzugriff die Zustimmung einer festgelegten Anzahl (das sogenannte *Quorum*) von Stationen zu erlangen, bevor die Operation ausgeführt werden kann. Man bezeichnet diese Ansätze darum auch als Abstimmungsverfahren (*voting*) [25, S. 5].

Bei den Abstimmungsverfahren erhält jeder Prozeß eine bestimmte Anzahl von "Stimmen" (häufig nur eine), die er bei konkurrierenden Anfragen einem der in Frage kommenden Prozesse zuschlägt. Die Stimmenzahl eines Prozesses ist häufig im voraus festgelegt (statisches Quorum) oder kann sich gar dynamisch an das Zugriffsverhalten anpassen (dynamisches Quorum) [25, S. 6]. Entsprechend kann sich auch das benötigte Quorum im Verlaufe der Anwendung verändern. Auf der anderen Seite unterscheidet man auch noch, ob sich das Quorum auf die Gesamtzahl der Prozesse bezieht oder nur auf eine Teilmenge. In letzterem Fall müssen jedoch die Replikate zumeist in Form einer Baum- oder Gitterstruktur angeordnet sein, damit das Verfahren funktioniert (man spricht dann auch von einer *strukturierten* Abstimmung) [25, S. 6].

Man kann zeigen, daß wechselseitiger Ausschluß bei Schreibzugriffen garantiert ist, wenn das sogenannte *Schreibquorum* Q_w mehr als die Hälfte der Gesamtstimmzahl beträgt [24, S. 298]. Zu beachten ist, daß die Schreiboperation auch nur auf den Replikaten im Schreibquorum durchgeführt werden muß. Wenn man nämlich sicherstellt, daß die Stimmenanzahl für das Lesen (das sogenannte *Lesequorum* Q_r) zusammen mit Q_w die Gesamtstimmzahl übersteigt, dann ist unter den gelesenen Werten immer ein aktueller Wert. Die Unterscheidung zwischen den einzelnen Werten erfolgt dabei über Zeitstempel oder Versionsnummern [25, S. 6].

Realisierung mittels Gruppenkommunikation. Bei der Realisierung von stark konsistenten Replikationsmechanismen mittels Gruppenkommunikation betrachtet man die Menge der Knoten, die jeweils Replikate tragen, als eine *multicast*-Gruppe. Für einen Schreibzugriff sendet ein Knoten den neuen Wert an die Primärstation, die diesen dann mittels eines vollständigen *multicast* an die Replikate verteilt (vergleiche Tabelle 3.1). Der *multicast* muß nach einer beschränkten Zeit beendet sein, damit die Stationen wissen, wann alle Replikate auf dem neuesten Stand sind und wieder konsistent gelesen werden können; hierfür kommen also nur sofortige oder zeitbeschränkte Zustellungen in Frage (vergleiche Tabelle 3.2). Es ist außerdem wichtig, daß alle Replikate die Nachrichten der Primärstation in der gleichen Reihenfolge wahrnehmen. Da aber die Aktualisierungen nacheinander ohne zeitliche Überschneidungen von einer zentralen Stelle aus abgeschickt werden, benötigt man keine besonderen Ordnungsgarantien auf dem benutzten *multicast*, wenn die notwendige totale *FIFO*-Ordnung bereits auf andere Weise sichergestellt ist.

Bei den "demokratischen" *voting*-Ansätzen benötigt man offensichtlich keinen vollständigen *multicast* mehr, sondern nur noch einen k-zuverlässigen, wobei k gerade dem Schreib- bzw. Lesequorum entspricht. Jedoch muß die Zustellung sofort oder mindestens zeitbeschränkt sein und alle Replikate müssen die Nachrichten in der gleichen Reihenfolge erhalten (totale Ordnung, vergleiche Tabelle 3.3).

3.5.2 Auskunftsdienste

Auskunftsdienste sind alle jene Dienste, die bei Bedarf dem Benutzer eine Information liefern. Beispiele sind das WWW, Telefonbücher oder Versandhauskataloge. Bei diesen Diensten ist schwache Konsistenz in der Regel akzeptabel, wenn gewährleistet ist, daß aktuelle Versionen der Dienste irgendwann beim Benutzer ankommen. Um diese Dienste also mittels Gruppenkommunikation zu realisieren, benötigt man nur einen zuverlässigen *multicast* mit allmählicher Zustellung (*reliable eventual delivery*). Bei selten aktualisierten Diensten benötigt man aus ähnlichen Gründen wie bei starker Konsistenz keine besonderen Ordnungsgarantien. Aber auch, wenn die Dienste öfter auf den neuesten Stand gebracht werden, werden sich auch bei einer möglichen Umordnung der neuen Werte während des *multicast* irgendwann die aktuellen Daten in den Replikaten durchsetzen.

Im Falle von Versandhauskatalogen oder großen Datenbanken im *WWW* ist es jedoch meist effizienter, nicht immer den kompletten neuen Wert des Repliaktes zu versenden, sondern immer nur eine Differenz zum letzten Wert (beispielsweise diejenigen Produkte oder Telefonnummern, die sich im Vergleich zum letzten Wert verändert haben). Nun kommt es allerdings auf die Ordnung der Zustellung der *multicast*-Nachrichten an, da eine Umordnung zweier Änderungen in einer auf Dauer inkonsistenten Datenbank münden könnte (man nennt dies auch die Problematik des *lost update* [24, S. 12]). Darum muß man in diesen Fällen wenigstens eine totale Ordnung auf der Nachrichtenzustellung fordern.

3.5.3 Individuelle Informationsdienste

Informationsdienste bestehen im Gegensatz zu Auskunftsdiensten aus regelmäßigen Zustellungen von aktuellen Informationen. Beispiele hierfür sind die Zusendung des regelmäßigen morgendlichen Börsentickers, Suchergebnisse "intelligenter" Agenten oder das Abonnement einer Tageszeitung, die nur die den betreffenden Benutzer interessierenden Meldungen enthält. Diese Dienste sind zwar in ihren Ausprägungen (wie sie der Kunde sieht) an sich nicht repliziert, da jeder Kunde in der Regel unterschiedliche Präferenzen hat, doch ist die Zahl möglicher Benutzer solcher Dienste vermutlich so groß, daß Replikation als wichtiger Beitrag zur Lösung der dadurch entstehenden Leistungs- und Verfügbarkeitsprobleme erscheint.

Bei der Realisierung durch Gruppenkommunikation hat man es hier mit einem ähnlichen Szenario wie bei den Auskunftsdiensten zu tun. Die Zustellung muß zuverlässig sein und — obwohl manche Kunden vielleicht eine zeitbeschränkte, unmittelbare oder gar sofortige Zustellung wünschen werden — genügt es meist, wenn Nachrichten allmählich eingehen.

3.5.4 Diskussionsforen

Diskussionsforen sind Informationsdienste, bei denen jeder einzelne Benutzer selbst eine Art Dienstanbieter ist. Die oft genannten Beispiele hierfür sind die bekannten *mailing lists* oder die Diskussionsforen (*newsgroups*) des *Usenet* [88, 89, 93]. Jeder neue Diskussionsbeitrag ist quasi ein eigener *multicast* an die beteiligten Personen. Bei derartigen Diensten genügt wieder ein schwach-konsistentes Replikationsschema, wobei allerdings die Zustellung in kausaler Reihenfolge geschehen muß, um beispielsweise zu verhindern, daß bereits Antworten auf anscheinend noch nicht gestellte Fragen bei einem Benutzer eingehen.

3.5.5 Verteilte Bibliotheken und Software-Archive

Bei verteilten Bibliotheken und Software-Archiven handelt es sich im wesentlichen auch um Auskunftsdienste, bei denen aber ein noch schwächerer Grad an Konsistenz vertretbar scheint. Ist ein Buch oder die neue Version eines Software-Paketes noch nicht in einer lokalen Kopie auf dem nächsten Replikat der Datenbasis vorhanden, kann man normalerweise auch noch ein paar Tage warten, bis man sich mit den neuen Informationen beschäftigt, oder notfalls auch ein entfernteres Replikat nach diesen Informationen anfragen. Hier genügt darum meist eine bestmögliche Zuverlässigkeit bei der Zustellung, sowie daß Aktualisierungen allmählich und ohne besondere Ordnung bei den Replikaten eintreffen.

3.5.6 Zusammenfassung

Die in den vergangenen Unterabschnitten beschriebenen Dienste decken einen großen Teil dessen ab, was in der heutigen technischen Welt anzutreffen ist. Sie machen zum einen deutlich, daß die zum Ende von Abschnitt 3.4 beschriebenen Grade von "Konsistenz" verschiedener Gruppenkommunikationsmechanismen durchaus praktische Relevanz besitzen; zum anderen habe ich gezeigt, daß man all diese Dienste durch einen *multicast* einfach realisieren kann. Insgesamt muß man also bei der Realisierung von replizierten Diensten sehr genau die Anforderungen des Problems und die Einschränkungen der Systemumgebung in Betracht ziehen, um eine effiziente und korrekte Lösung zu erarbeiten.

3.6 Ausgewählte Replikationsalgorithmen

Zum Abschluß dieses Kapitels möchte ich in diesem Abschnitt ein paar wenige ausgewählte Algorithmen für schwach-konsistente Replikation vorstellen. Ihre Beschreibung soll zeigen, daß es bereits viele gute und in der Praxis erprobte Lösungen für dieses Problem gibt und daß diese Algorithmen auch unter einem starken Fehlermodell noch durchaus effizient ablaufen. Die Algorithmen wurden gemäß ihrer Prominenz in der Forschungsliteratur ausgewählt und werden in chronologischer Reihenfolge vorgestellt.

3.6.1 *Usenet*

Eine frühe Realisierung von Gruppenkommunikation erfolgte im Rahmen des sogenannten *Usenet*, einem logischen Netzwerk aus vielen Rechnern, welches zur Verbreitung von Informationen und Daten typischerweise an tausende von Empfängern dient. In seiner ursprünglichen Form, die in einem der üblichen "Request for Comments" (RFC) festgehalten wurde [88, 89], besteht das *Usenet* aus einer Menge von Rechnerknoten, zwischen denen logische Verbindungskanäle bestehen. Jeder Knoten weiß von seinen unmittelbaren Nachbarn, welche Art von Information ihn interessiert und somit an ihn weitergeleitet werden sollte.

Eine Nachricht, beispielsweise ein neuer Diskussionsbeitrag, kann auf jedem Rechner des *Usenet* zur weiteren Verbreitung abgelegt werden. Dort wird er mit einer weltweit eindeutigen Marke (*message id*) versehen. Die Rechner nehmen regelmäßig Kontakt zueinander auf und tauschen die seit der letzten Verbindung neu hinzugekommenen Informationen untereinander aus (natürlich nur, wenn sie den jeweiligen Nachbarn interessieren). Die durch diese Übertragungen neu eingegangenen Nachrichten werden jeweils weiter an alle Nachbarn verteilt, die sich dafür interessieren. Durch diese Verfahrensweise (auch "Fluten" oder *flooding* genannt) erreichen alle Informationen schließlich alle Rechner des Netzes.

Probleme entstehen jedoch, wenn so Nachrichten zwischen zwei Knoten immer wieder hin und her geschickt werden. Ein zentraler Bestandteil des Verteilungsverfahrens ist also ein Mechanismus, mit dessen Hilfe sich die einzelnen Rechnerknoten merken, welche Nachrichten sie bereits erhalten haben (*history mechanism*). Solche Nachrichten werden kein zweites Mal an die Nachbarn weitergeleitet und das oben genannte Problem kann nicht mehr auftreten.

Es gibt zahlreiche mögliche Optimierungen dieses einfachen Verfahrens, die dafür sorgen, daß die Bandbreite des zur Verfügung stehenden Kommunikationsmediums besser ausgenutzt wird. Der RFC macht hierfür ein paar Vorschläge, doch sind sie nicht unmittelbar Teil des Standards [88, S. 18]. Dies wird durch die Spezifikation des *network news transfer protocol* (NNTP) von Kantor und Lapsley [93] für verbindungsorientierte (*stream based*) Übertragungswege (beispielsweise mittels des TCP-Protokolls [134, S. 373ff]) teilweise nachgeholt. Darin ist vorgesehen, daß sich einzelne Rechner vor der eigentlichen Übertragung der Daten zuerst darüber verständigen, welche Nachrichten tatsächlich gesendet werden müssen und welche nicht [93, S. 4].

Der Gruppenkommunikationsmechanismus des *Usenet* ist einer der ersten seiner Art. Er ist aber nicht nur aus historischen Gründen interessant sondern auch ein Beispiel dafür, daß sich einfache Verfahren in der Praxis gut bewähren können. (Auch die Neuformulierung des ersten RFC [89] ändert an der ursprünglichen Verteilungsprozedur nichts.) Für die Zwecke dieser Arbeit ist auch bemerkenswert, daß der Standard keinerlei Ordnungsgarantien für die Verteilung der Daten fordert. Die Herstellung von Ordnungen bleibt der "Zustellungsapplikation" (beispielsweise dem *newsreader*) überlassen. Das Protokoll sieht aber Felder innerhalb der Nachrichten vor, welche die kausale Ordnung der aktuell vorhandenen Nachrichten vor der Zustellung erleichtern [89, S. 9].

3.6.2 *Timestamped anti-entropy*

Zusätzlich zu dem bereits erwähnten flexible Rahmenwerk für einen Gruppenkommunikationsmechanismus hat Richard Golding in seiner Dissertation [73] auch eine Realisierung des Konzeptes in Form eines Replikationsalgorithmus vorgeschlagen. Golding nennt diesen Ansatz das *timestamped anti-entropy*-Protokoll (*TSAE*-Protokoll). Dieses Protokoll realisiert zuverlässige, allmähliche Zustellung von *multicast*-Nachrichten. Durch die unterschiedliche Ausgestaltung einer besonderen Moduls (*message ordering component*) können zusätzlich verschiedene Ordnungsgarantien auf der Zustellung erreicht werden.

Kommunikation. Die grundlegende Verfahrensweise des *TSAE*-Protokolls ist einfach: Die einzelnen Prozesse, die Replikate halten, bauen regelmäßig Kontakt zu ihren Nachbarn im Netzwerk auf und tauschen Informationen darüber aus, welche *multicast*-Nachrichten von welchen Prozessen sie jeweils schon empfangen haben und welche nicht. Anschließend senden sich die Kommunikationspartner die jeweils dem anderen fehlenden Nachrichten zu, die sie in einem speziellen Nachrichten-Archiv (*message log*) gespeichert haben. Ein solcher Durchgang wird eine *anti-entropy session* genannt, da die Entropie innerhalb des Systems (hier verstanden als ein Maß für die "Unordnung" oder Inkonsistenz) durch den Nachrichtenaustausch immer abnimmt. Durch die jeweilige Kontaktierung anderer Nachbarn ist gewährleistet, daß *multicast*-Nachrichten, die ein Knoten erzeugt, über indirekte Pfade von allen anderen Knoten erhalten wird.

Einen wichtigen Bestandteil des *TSAE*-Protokolls bilden Zeitstempel logischer Uhren. Logische Uhren messen eine abstrakte "logische" Zeit und gehen auf einen bekannten Artikel von Leslie Lamport zurück [101]. Die Auflösung dieser Uhren muß im Falle von *TSAE* fein genug sein muß, so daß alle wesentlichen Ereignisse des Prozesses zu unterschiedlichen Zeitpunkten stattfinden [73, S. 50]. Zeitstempel im *TSAE*-Protokoll enthalten immer auch einen Hinweis auf den Prozeß, von dem die jeweilige Nachricht stammt, so daß man eine gegebene Nachricht immer ihrem Absender zuordnen kann.

Mit Hilfe der Zeitstempel ist es einfach möglich zu überprüfen, welche Nachrichten ein Prozeß bereits erhalten hat und welche ihm noch fehlen. Zu diesem Zweck verwendet Golding die sogenannte *Vektor-Zeit* [111], eine Weiterentwicklung von Lamports logischer Zeit. Dazu werden auf einem Knoten die jeweils "spätesten" Zeitstempel von Nachrichten anderer Knoten in Form eines Vektors angeordnet. Dieser "Empfangsvektor" (Golding nennt ihn *summary vector* [73, S. 52]) faßt in kompakter Weise den momentanen Empfangszustand (also die Sicht des globalen Zustandes) eines Prozesses zusammen.

Damit das Nachrichtenarchiv eines Prozesses nicht über alle Maßen anwächst, müssen Nachrichten, die jedem Prozeß bereits zugestellt wurden, irgendwann daraus entfernt werden. Hierzu werden auch die Informationen aus dem Empfangsvektor benutzt. Das minimale Element über alle Vektoreinträge eines erhaltenen Empfangsvektors ist eine implizite Bestätigung darüber, daß alle Nachrichten mit einem niedrigeren Zeitstempel bereits von diesem Prozeß empfangen worden sind. Dieser Wert wird pro Prozeß in einem zusätzlichen *Bestätigungsvektor* (*acknowledgement vector*) gespeichert [73, S. 53]. Eine Nachricht, die einen niedrigeren Zeitstempel hat als alle Einträge im lokalen Bestätigungsvektor ist von allen anderen Prozessen bereits gesehen worden und kann darum gefahrlos aus dem Nachrichtenarchiv gelöscht werden, sofern sie bereits an die Anwendungsebene des Prozesses zugestellt wurde.[3]

Gruppenzugehörigkeit. Die Regelungen bezüglich der Gruppenzugehörigkeit haben im *TSAE*-Protokoll ebenfalls schwach-konsistenten Charakter. Jeder Prozeß verfügt über ein ihm eigenes Wissen (*view*) darüber, welche anderen Prozesse alle zur Gruppe gehören und welche nicht [73, S. 81]. Dieses Wissen wird wie das Wissen um erhaltene *multicast*-Nachrichten innerhalb einer *anti-entropy session* ausgetauscht und so im Netzwerk verteilt. So ist garantiert, daß — wenn keine weiteren Austritte und Neuaufnahmen auftreten — alle Gruppenmitglieder nach einer gewissen

[3] Es genügt sogar, daß die zu löschende Nachricht einen niedrigeren Zeitstempel hat als alle Einträge des Bestätigungsvektors, die nur den jeweiligen Nachbarn im Kommunikationsgraphen zugeordnet sind [73, S. 67].

Zeit das gleiche Wissen über den Zustand der Gruppe haben.

Die Beitritts- und Austrittssemantiken sind so gewählt, daß ein Prozeß praktisch ohne Vorlaufzeit neu zur Gruppe hinzustoßen [73, S. 83], daß aber ein Austritt erst erfolgen kann, wenn alle anderen Prozesse davon in Kenntnis gesetzt wurden [73, S. 92][4]. Eine Nachricht, die zu einem Globalzeitpunkt t abgeschickt wurde, wird von allen Prozessen empfangen werden, die zum Zeitpunkt t Gruppenmitglieder sind. Alle Prozesse, die nach t zur Gruppe stoßen, erhalten entweder die Nachricht selbst oder einen Zustand, in dem der Erhalt der Nachricht bereits eingearbeitet ist.

Um beizutreten, sucht ein neuer Prozeß sich eine Menge von sogenannten *Sponsoren* (*sponsors*). Die Sponsoren sind anfangs das Tor zur Außenwelt: Von ihnen erhält der Prozeß initiale Kopien von Gruppenzustand und -sicht, und die Sponsoren wiederum leiten ihr aktualisiertes Wissen über den veränderten Gruppenzustand an die anderen Gruppenmitglieder weiter. Sobald sich ein Prozeß bereiterklärt hat, einen anderen "zu sponsern," kann der neue Prozeß aktiv am Leben in der Gruppe teilnehmen und Nachrichten versenden. Aus Fehlertoleranzgesichtspunkten kann man jedoch fordern, daß erst eine bestimmte Anzahl k von Sponsoren gefunden werden müssen, bevor der Prozeß "loslegen" darf. Wenn nämlich im Falle $k = 1$ der Sponsor-Prozeß ausfällt, nachdem er den neuen Gruppenzustand weitergegeben hat, dann werden alle Prozesse nach einer bestimmten Zeit denken, der "gesponserte" Prozeß gehöre zur Gruppe. Da aber die einzige Verbindung dieses Knotens zum restlichen Netzwerk über den Sponsor verlief, ist der neue Knoten jetzt vom Rest des Netzes getrennt. Nachrichten, die an ihn verschickt werden oder die er selbst verschickt, werden nie zugestellt werden können.

Bewertung des Algorithmus. Der Speicherplatz, der sowohl in den Nachrichten als auch innerhalb der Prozesse zusätzlich benötigt wird, ist relativ gering ($O(1)$, bzw. $O(n)$, wobei n die Anzahl der Prozesse bezeichnet). Die Zeit, die durchschnittlich benötigt wird, um Informationen im Netzwerk zu verbreiten, hängt von einer ganzen Reihe von Parametern ab (Netzwerktopologie, Häufigkeit einer *anti-entropy session*, Auswahl des Kommunikationspartners, Anzahl der beteiligten Prozesse). Golding hat seinen Algorithmus mit verschiedenen Parametern simuliert und kommt bezüglich der Latenz der Zustellung zu dem Ergebnis, daß sich Nachrichten erstaunlich schnell im Netzwerk verbreiten [73, S. 101f]. Ein vielleicht noch wichtigeres Ergebnis ist, daß sich die Ergebnisse mit steigender Anzahl von Prozessen nicht überproportional verschlechtern, sich der Algorithmus also gut skalieren läßt. Ein weiterer Vorteil der Methode ist, daß Nachrichten immer in Blöcken (*batches*) ausgetauscht werden können, was in Weitverkehrsnetzen oft effizienter ist als viele kleine Nachrichten zu verschicken [73, S. 49].

[4] Genauer gesagt bedeutet dies: Der tatsächliche Austrittsschritt muß kausal abhängen von je einem Bestätigungs-Ereignis in allen Prozessen, die wiederum alle mit der Ankündigung des Austritts in kausaler Abhängigkeit stehen müssen.

Kumar und Adly bemerken jedoch, daß ein Platzbedarf von $O(n)$ und die festgestellte Skalierbarkeit sich eigentlich widersprechen [99, S. 2]. Szenarien mit Tausenden von Replikaten würden zu einem Speicherbedarf von mehreren hundert Kilobyte *pro Prozeß* führen. Dieses Dilemma ist durch eine effizientere Ausgestaltung des Algorithmus nicht lösbar, da alle Prozesse immer zu einem gewissen Maße um den Gesamtzustand der Gruppe wissen müssen und die Größenordnung der verwendeten Vektoren immer in $O(n)$ bleiben muß [37]. Außerdem erlaubt Goldings Protokoll keine Gruppenüberlappungen [73, S. 81]; der Autor umgeht also die damit verbundenen Probleme.

Die oben vorgestellte Form des Löschens von Nachrichten funktioniert auch nur, wenn die Uhren der einzelnen Prozesse etwa in der gleichen Geschwindigkeit voranschreiten und birgt Probleme in sich, wenn man rein asynchrone Systeme betrachtet. In ihnen kann es vorkommen, daß die (logische) Uhr eines Prozesses um ein beliebiges Vielfaches schneller läuft als die eines anderen Prozesses. Da immer nur das Minimum des Empfangsvektors als implizite Bestätigung benutzt wird, kann es vorkommen, daß eine "späte" Nachricht erst nach einer beliebigen Zeit aus dem Archiv gelöscht werden darf mit dem Resultat, daß die Größe des Archives jede Schranke übersteigt. In solchen Fällen muß man den gesamten Empfangsvektor eines jeden Prozesses speichern (der ja ein genauerer Zeuge über dessen Weltsicht ist) und zu einer Bestätigungs*matrix* zusammenfassen [73, S. 73].[5] Der Speicherplatzbedarf pro Prozeß steigt damit auf $O(n^2)$ [73, S. 74]. In der Praxis existieren jedoch Standardalgorithmen, die eine ausreichend genaue Uhrensynchronisation gewährleisten können, so daß dieser Fall eher von theoretischen Interesse ist. Dennoch muß man sich in der Praxis überlegen, welche Maßnahmen bei einem drohenden Überlauf des Nachrichten-Archives zu treffen sind.

Bewertung der Fehlertoleranz. Da das Verfahren auf schwache Konsistenz der Replikate abzielt, legt es bezüglich der Fehlertoleranz ein gutes Verhalten an den Tag. Vorübergehende Fehler innerhalb des Netzwerkes, die die Kommunikation behindern, oder Netzknoten, die vorübergehend ausfallen, sind kein großes Problem mehr: Durch die periodische Initiierung des Nachrichtenaustausches mit den Nachbarn werden sogar Knoten, die (durch Fehler oder absichtlich) längere Zeit vom Netzwerk getrennt waren bei der nächsten *anti-entropy session* automatisch in einen (nahezu) konsistenten Zustand nachgezogen. Voraussetzung dafür ist jedoch, daß Prozesse ihre internen Werte auf einem *stabilen Speichermedium* (*stable storage*) sichern, um auch nach dem Hochfahren im Anschluß an einen Systemabsturz aus einem wohldefinierten Zustand zu starten. Die *anti-entropy sessions* haben dabei einen ähnlichen Status wie Transaktionen in verteilten Datenbanken: Bei einem Fehler während einer *session* werden die beteiligten Prozesse in den Ausgangszustand zurückgesetzt, den

[5] In diesem Falle wird jeder Prozeß zu einem Beobachter [127, S. 161ff] des Gesamtsystems.

sie vor Beginn der *session* hatten. Unter der Voraussetzung von stabilem Speicher kann der Algorithmus (auch der Gruppenzugehörigkeitsmechanismus) also eine beliebige Anzahl von vorübergehenden Fehlern verkraften.

Bei permanenten Fehlern kommt es nur auf die Konnektivität des zugrundeliegenden Netzwerkes an. Damit der Algorithmus k permanente Fehler tolerieren kann, muß das Netzwerk immer mindestens $(k+1)$-verbunden sein und ein neu hinzukommender Prozeß benötigt $k+1$ Sponsoren, bevor er einer Gruppe beitreten kann.

3.6.3 *flood-d*

In Fortführung der Ideen von Golding hat sich Katia Obraczka in ihrer Dissertation [119] mit dem lokalen Platzbedarf und der Effizienz des Nachrichtenaustausches des *TSAE*-Protokolls beschäftigt. Sie stellt dabei fest, daß sich Gruppenkommunikation mit einer festen Kommunikationstopologie und einer einzigen großen *multicast*-Gruppe nicht in beliebiger Größe im heutigen Internet realisieren läßt. Die Topologie des Internet ist weder stabil noch vollständig bekannt [119, S. 16] und die Autonomie seiner Verwaltungseinheiten wird sehr schlecht durch eine einzige *multicast*-Gruppe wiedergespiegelt [119, S. 12].

Obraczka zieht aus ihrer Analyse der bestehenden Protokolle die folgenden Schlußfolgerungen [119, S. 9]:

- Zur Verbesserung der Skalierbarkeit muß eine Hierarchie von Replikationsgruppen eingeführt werden.
- Die zugrundeliegende Kommunikationstopologie muß sich dynamisch an wechselnde Gegebenheiten anpassen können.

Realisiert werden diese Ideen in einem hierarchischen Replikationsprotokoll namens *flood-d*.

Kommunikation. In *flood-d* sind jeweils eine kleine Anzahl von Replikaten innerhalb von *Replikationsgruppen* (*replication groups*) zusammengefaßt. Die Kommunikation innerhalb einer Replikationsgruppe erfolgt ganz normal wie im *TSAE*-Protokoll von Golding mittels periodisch stattfindenden *anti-entropy sessions* zwischen den einzelnen Replikaten. Durch *Randreplikate* (*corner replicas*), die also Mitglied mehrerer Replikationsgruppen sind, wird die Kommunikation über die Grenzen von Replikationsgruppen hinweg ermöglicht [119, S. 14].

Während der normalen Operation werden Daten über die aktuelle Leistungsfähigkeit des physikalischen Netzwerkes gesammelt; von Zeit zu Zeit berechnet damit ein ausgewählter Prozeß in jeder Gruppe einen kostenoptimalen Verbindungsgraphen für die Gruppe mit einer gewünschten

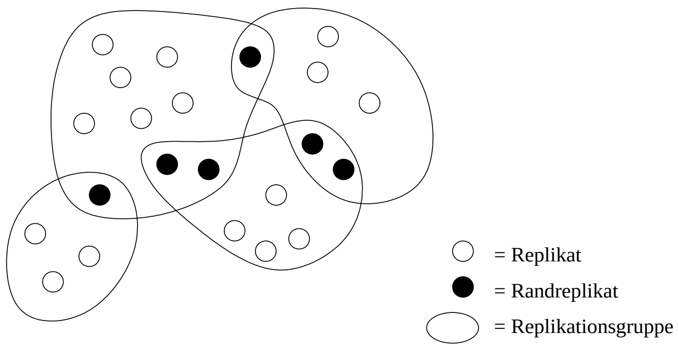

Abbildung 3.2: Replikationsgruppen und Randreplikate in *flood-d*.

Konnektivität [119, S. 20]. Falls die neue Topologie von den Gesamtkosten her besser abschneidet als die bereits vorhandene, formiert sich die Gruppe gemäß der neuen (logischen) Topologie um. Obraczka geht davon aus, daß einzelne Verwaltungseinheiten (*administrative domains*) jeweils eine eigene Replikationsgruppe bilden und daß die Verbindungsstruktur zwischen diesen Verwaltungseinheiten durch die Organisation des Internet im wesentlichen bereits hierarchisch geordnet ist [119, S. 16].

Bewertung des Algorithmus. Gegenüber dem ursprünglichen *TSAE*-Protokoll hat die Weiterentwicklung von Obraczka einige wesentliche Vorteile. Durch die Einführung einer Hierarchie ist es besser möglich, die hierarchische Gliederung des Internet-Namensraumes [117] (und damit auch seiner Verwaltungseinheiten) abzubilden. Entscheidungen auf einer Ebene der Hierarchie betreffen dann nicht automatisch alle anderen Replikate oder Replikationsgruppen. Auch geht damit eine Verringerung des Platzbedarfes pro Prozeß einher. Die Zustandsvektoren innerhalb eines Prozesses haben jetzt nur noch eine Größe von $O(r)$, wobei r die Anzahl der Mitglieder der Replikationsgruppe ist. Nur Randreplikate müssen noch zusätzlich Informationen halten über den Zustand jeder Gruppe, in der sie Mitglied sind, aber selbst das ist noch weniger als der ursprüngliche Platzbedarf von $O(n)$ [119, S. 14]. Die Simulationsergebnisse in Obraczkas Dissertation zeigen auch, daß sich durch die Optimierungsstrategien für die logische Kommunikationstopologie die Kosten für den Nachrichtenaustausch innerhalb durchaus akzeptabler Grenzen bewegen und sich die Kommunikationskosten um 30 bis 50 Prozenz verringern ließen [119, S. 116]. Kumar und Adly merken jedoch an, daß durch den periodischen Nachrichtenaustausch (wie ebenso beim *TSAE*-Protokoll) immer noch ein gutes Stück an Bandbreite "verschenkt" wird [99, S. 2].

3.6.4 *HPP*

Mit ihrem erst kürzlich veröffentlichten *hierarchical propagation protocol* (*HPP*) stellen Kumar und Adly eine konsequente Weiterentwicklung und Optimierung von *flood-d* vor [1, 99]. Ihr Ziel war es, verläßliche, allmähliche und kausale Zustellung von Nachrichten zu erzielen bei gleicher Toleranz gegenüber vorübergehenden und permanenten Fehlern wie bei *flood-d* und gleichen Möglichkeiten, die logische Kommunikationstopologie gefahrlos dynamisch zu ändern. Ein zusätzliches Entwurfsziel war die unbedingte Vermeidung überflüssiger Nachrichten, wie sie in den beiden vorhergenannten Protokollen entstehen [99, S. 4].

Kommunikation. Das *HPP*-Protokoll ist eine logische Fortführung von *flood-d*, in welcher die Replikationsgruppen nur noch zwei Mitglieder haben. Die Hierarchie, in denen die Replikate angeordnet sind, ist ein Baum, wobei jedes Replikat Mitglied in der Replikationsgruppe seines Elternknotens und in denen seiner Kinderknoten ist [99, S. 5] (siehe auch Abbildung 3.3).

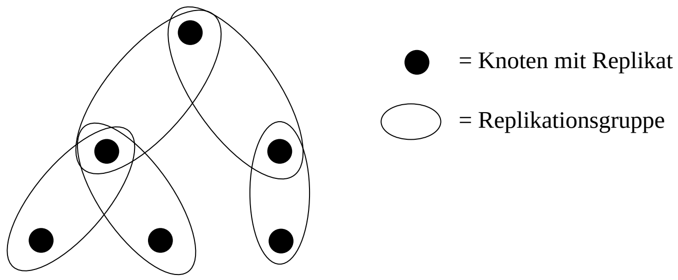

Abbildung 3.3: Die Hierarchie von *HPP* als Fortführung der Konzepte aus *flood-d*.

Wenn ein Knoten eine Nachricht an alle Replikate senden will, verschickt er sie einfach innerhalb jeder Replikationsgruppe, der er angehört. Das bedeutet, daß bei fehlerfreier Ausführung sowohl der Elternknoten als auch alle seine Kinder die Nachricht erhalten. Erhält ein Knoten eine Nachricht, tut er das gleiche, nur spart er natürlich die Gruppe mit dem Knoten aus, von dem er die Nachricht bekam. Dies wird rekursiv fortgesetzt und endet an den Blättern des Baumes. Es ist einfach zu sehen, daß so bei fehlerfreier Ausführung jeder Knoten die Nachricht genau einmal erhält und somit überflüssige Nachrichten nicht auftreten [99, S. 5]. Es ist ebenso leicht einzusehen, daß im Falle von *FIFO*-Kanälen sogar die kausale Ordnung der Nachrichten eingehalten wird, da es zwischen zwei Knoten nur einen einzigen Pfad und somit keine Überholmöglichkeit gibt [99, S. 8].

Das so gestaltete Protokoll wurde nun erweitert, um auch bestimmte

Fehler zu tolerieren. Auf den einzelnen Kommunikationsleitungen wird das *stop and wait*-Protokoll verwendet [134, S. 143ff], bei dem Nachrichten durch eindeutige Sequenznummern gekennzeichnet werden und die Kommunikationspartner immer auf "die nächste" Nachricht warten (und sie bei Empfang unter Angabe der Sequenznummer quittieren). So wird auf der einen Seite die *FIFO*-Ordnung beibehalten und auf der anderen Seite das Duplizieren oder der Verlust von Nachrichten ausgeschlossen [99, S. 8]. Hierfür ist es jedoch notwendig, daß jeder Knoten Informationen über die jeweils zuletzt empfangene Nachricht eines seiner Nachbarn hält.

Das *HPP*-Protokoll hat die Fähigkeit, temporär oder langfristig ausfallende Knoten zu überspringen. Zu diesem Zwecke halten die einzelnen Knoten nicht nur Informationen über den Sende- und Emfangszustand ihrer direkten Nachbarn, sondern auch darüber, was diese wieder über ihre Nachbarn wissen. Im Fehlerfalle kann so ein gewählter Koordinator temporär die Aufgaben des ausgefallenen Knotens übernehmen. Falls der fehlerhafte Knoten irgendwann wieder hochfährt, gibt der Koordinator seine Aufgabe wieder an ihn ab.

Gruppenzugehörigkeit. Wenn ein Knoten in die Hierarchie aufgenommen werden möchte, sucht er sich einen beliebigen Elternknoten und stellt einen "Adoptionsantrag" (*join request*). Der Elternknoten teilt diese Tatsache allen seinen Nachbarn mit und schickt als positiven Bescheid die eigene (begrenzte) Zustandssicht der Hierarchie an das neue Mitglied [99, S. 9f]. Beim Verlassen der Hierarchie (*leave*) muß ein Knoten so lange warten, bis der Elternknoten und dessen Nachbarn dies realisiert haben, und kann sich dann gefahrlos entfernen. Alle seine Nachkommen werden durch diesen Austritt ebenfalls aus der Hierarchie entfernt. Die Semantik der Gruppenkommunikation ist dabei dieselbe wie bei den beiden vorhergenannten Protokollen.

Umorganisation der Hierarchie. Im Gegensatz zu *TSAE* und *flood-d* ist ein Mechanismus zur Umorganisation der Hierarchie ein expliziter Bestandteil von *HPP*. Eine solche Umorganisation kann beispielsweise zur Optimierung der Kommunikationskosten notwendig sein. Jedoch ist die Restrukturierung nicht durch ein einfaches Verlassen mit einem anschließenden Beitritt getan; es muß zusätzlich sichergestellt werden, daß die kausale Ordnung eingehalten wird und daß auch im Fehlerfalle alle Nachrichten den sich bewegenden Knoten erreichen. Ein Nachrichtenverlust kann zum Beispiel eintreten, falls eine Nachricht beim neuen Elternknoten schon vorbeigekommen ist, den alten Elternknoten aber (aus Richtung der "Großeltern" kommend) noch nicht passiert hat. Der Mechanismus löst dieses Problem, indem sowohl auf Seiten des neuen Elternknotens als auch auf Seiten des sich bewegenden Knotens Nachrichten für die Dauer der Aktion gepuffert werden und durch den Versand von speziellen Markierungen (*special markers*) jeder Knoten weiß, wie und wann er die gepufferten Nachrichten in seinen eigenen Nachrichtenstrom einsortieren muß [99, S. 11].

Bewertung. Das *HPP*-Protokoll ist in den Grundzügen eine degenerierte Form des *flood-d*-Protokolls von Obraczka. Die Kommunikationstopologie ist ein Baum (ein 1-verbundener Graph), die Replikationsgruppen haben sämtlich die Größe zwei und jeder Knoten ist Mitglied in den Replikationsgruppen seiner jeweiligen Nachbarn. Bei der Regelung der Gruppenzugehörigkeit macht ein neu hinzukommender Knoten quasi eine neue Zweiergruppe mit dem Elternknoten auf. Die Kommunikation, die bei Erhalt oder der Erzeugung von neuen Nachrichten stattfindet, könnte man als eine bedarfsgerechte Auslösung von *anti-entropy sessions* sehen (*anti-entropy sessions on demand*).

Die Reduzierung der Kommunikationstopologie und der Gruppengröße bringt Vorteile vor allem in der Effizienz mit sich: Es entstehen keine redundanten Nachrichten und der Speicherplatz, der für die Verwaltung des Knotenzustandes benötigt wird ist nur durch den Verzweigungsgrad des Baumes beschränkt. Auch ist die Herstellung der kausalen Ordnung auf der Zustellung durch die gewählte Kommunikationstopologie begünstigt. Der verringerte Aufwand bei normaler Operation wird erkauft durch eine komplexere Behandlung von Fehlersituationen. Während die Fehlertoleranz gegenüber vorübergehenden oder permanenten Fehlern integraler Bestandteil von *flood-d* und *TSAE* war, wirken die in *HPP* vorhandenen Mechanismen aufgesetzt und sind mit Abstand die komplexesten Bestandteile des Protokolls. Während die Entscheidung, wieviele permanente Knotenausfälle das System vertragen soll, bei *TSAE* und *flood-d* eine Entscheidung zur Laufzeit war (und sich in der Konnektivität des Netzwerkes äußerte), muß sie bei *HPP* bereits zur Übersetzungszeit getroffen werden, da von ihr die Gestaltung der internen Datenstrukturen des Knotens abhängt. Im Gegensatz zu den anderen Protokollen bringt ein Knotenausfall in *HPP* immer auch eine Netzpartitionierung mit sich. Die Fähigkeit von *HPP*, gestörte Knoten zu überspringen, ist somit bei vorübergehenden Fehlern eine Effizienzsteigerung und bei permanenten Fehlern eine Überlebensgarantie, die in *TSAE* und *flood-d* strukturell nicht notwendig ist.

3.7 Ausblick

In diesem Kapitel habe ich Replikation als Methode in der Informatik mit ihren inhärenten Vor- und Nachteilen vorgestellt (Steigerung der Leistungsfähigkeit und Verfügbarkeit auf Kosten von Änderungsaufwand und Konsistenzsicherung). Dabei habe ich mich auf die Replikation von Diensten in Weitverkehrsnetzen konzentriert und gezeigt, daß schwache Konsistenzbedingungen für die Replikationskontrolle in diesen Fällen nicht nur effizienzsteigernd wirken, sondern auch viele Dienste erst realisierbar machen. Außerdem wurde der enge Zusammenhang zwischen Replikation und Gruppenkommunikation abstrakt und anhand von Beispielen beschrieben. Abschließend habe ich ein paar ausgewählte Replikationsalgorithmen vorgestellt.

Im nächsten Kapitel geht es um eine Ausprägung von Fehlertoleranz. Konsistenz durch Replikationsverfahren und Fehlertoleranz haben komplexe und vielfältige Wechselwirkungen untereinander [27]. Nach dem Lesen dieses Kapitels sollte klar geworden sein, was Replikation ist, wie man sie erreichen kann, und daß gerade bei den abschließend vorgestellten Algorithmen Aspekte der Fehlertoleranz eine wichtige Rolle spielen.

4

Selbststabilisierung

Inhaltsübersicht

Kurzfassung

Selbststabilisierung ist ein sehr natürliches Phänomen: Es meint, daß sich Lebewesen oder andere natürliche Systeme nach einer Störung ihres inneren Gleichgewichtes selbst wieder zu einem ausgeglichenen Zustand stabilisieren. Einfache Beispiele sind Springfedern aus alten Matratzen oder das bekannte Kinderspielzeug "Stehaufmännchen." In diesem Kapitel wird der Begriff der Selbststabilisierung im Rahmen verteilter Systeme formal definiert und durch Beispiele verdeutlicht. Dabei wird deutlich, daß sich Selbststabilisierung als Methode eignet, um Computersysteme unempfindlicher gegenüber bestimmten Störeinflüssen zu machen, beispielsweise wenn in einem Netzwerk Kommunikationsleitungen kurzzeitig ausfallen.

4.1 Einführung

In diesem Kapitel geht es um Selbststabilisierung als allgemeine Methode zur Realisierung von Fehlertoleranz. Nach dem aktuellen Forschungsstand ist Selbststabilisierung ein allgemeines Konzept, welches einen großen Teil der bisher betrachteten Aspekte der Fehlertoleranz in einem einzelnen Modell zusammenfaßt und gleichartig zu behandeln versucht. Daß dieses Konzept alltäglich ist, sollen die folgenden zwei einfachen Beispiele verdeutlichen:

4.1 Beispiel Sie sind nur noch in alten Matratzen und Sofakissen enthalten, doch haben die guten alten Springfedern noch lange nicht ausgedient. Als Experimentierobjekt im Physikunterricht lernt sie jeder Schüler kennen, berechnet Kraft und Federkonstante, und weiß, daß ihr mit herkömmlichen Mitteln kaum beizukommen ist. Man kann sich auf sie setzen, man kann sie in die Länge ziehen, man kann dies sogar beliebig lange tun, doch sobald man sie in Ruhe läßt, zieht sie sich zusammen oder dehnt sie sich aus, und zwar genau auf die Länge, die sie vorher hatte.

4.2 Beispiel Man bekommt sie heute nur noch in ausgewählten Spielzeugläden: Es sind die kleinen, kegelförmigen Spielzeugfiguren, die wegen ihres niedrigen Schwerpunktes und ihres halbkugelförmigen Unterleibes aus jeder Lage wieder in die Senkrechte zurückschwingen, die sogenannten "Stehaufmännchen." Ihren Namen benutzt man heute eigentlich nur noch übertragen: Mit "er ist ein Stehaufmännchen" meint man umgangssprachlich einen Menschen, der sich nicht unterkriegen läßt. Das eigentliche Spielzeug läßt sich auch tatsächlich nicht aus der Ruhe bringen: Man kann es auf den Kopf stellen, auf die Seite legen, durch die Luft wirbeln; immer wird es nach einer bestimmten Zeit des Hin- und Herschwingens in einen stabilen Zustand kommen und dort verharren.

Das Wesentliche an den beiden eben gezeigten Beispielen ist, daß die vorgestellten Systeme aus sich heraus, ohne äußeres Zutun nach einer endlichen Zeit in einen zumeist vorher bekannten, stabilen Ruhezustand übergehen, sobald die äußeren Störungen aufhören. Abstrakt gesehen ist dies genau eine Eigenschaft, die fehlertolerante Rechnersysteme erfüllen sollen.

Ich werde in diesem Kapitel den Begriff der Selbststabilisierung im Zusammenhang mit verteilten Algorithmen definieren und zur Verdeutlichung eine Reihe von Beispielen für derartige selbststabilisierende Algorithmen vorstellen. Diese Beispiele sind der Anknüpfungspunkt für den darauffolgenden Abschnitt, der sich abstrakt mit dem Potential der Selbststabilisierung beschäftigt und diese Eigenschaft mit dem älteren und bekannteren Begriff der Fehlertoleranz in Verbindung setzt. Der letzte Abschnitt beschäftigt sich mit den bisher in der Literatur bekannten zum Teil automatischen Entwurfsmethoden für selbststabilisierende Algorithmen.

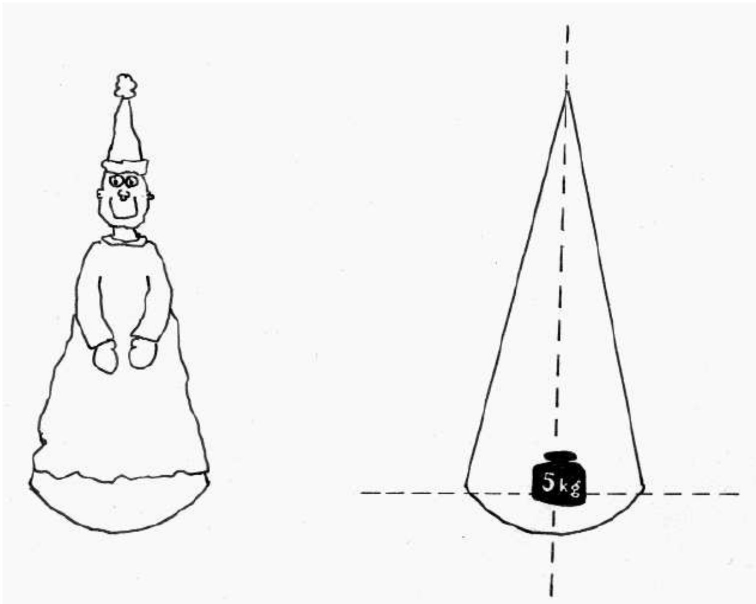

Abbildung 4.1: Skizze eines Stehaufmännchens (links) und Bauprinzip (rechts).

Darin werden die bisher veröffentlichten Vorgehensweisen in strukturierter Weise präsentiert und gegeneinander abgegrenzt.

Stabilisierung oder Selbststabilisierung. In der Literatur benutzt man zwei Begriffe für ein Konzept: Während in den Anfängen der Forschung auf diesem Themengebiet ausschließlich der Ausdruck "Selbststabilisierung" (*self-stabilization*) verwendet wurde [36, 49, 50, 98] findet man heute oft auch den vereinfachten Ausdruck "Stabilisierung" vor [31, 75, 77, 79]. Während letzterer häufig elegantere Satzkonstruktionen erlaubt, drückt ersterer meiner Meinung nach besser aus, worum es sich handelt (das Reflexivpronomen "sich" verdeutlicht eine innere Fähigkeit des Systems). Da es in diesem Kapitel um den Gegenstand an sich geht, will ich mich nicht mit einer unnötigen Verfeinerung der Nomenklatur aufhalten und im folgenden beide Begriffe synonym verwenden.

4.2 Selbststabilisierung

Wenn man eine Arbeit über selbststabilisierende Algorithmen schreibt, kommt man nicht umhin, im ersten Satz den Artikel "Self-stabilizing systems in spite of distributed control" [49] von Edsger W. Dijkstra zu erwähnen. In diesem Artikel, den manche für seinen besten halten [102], beschreibt er bereits im Jahre 1974 drei verteilte Systeme, die — egal, wie sehr man sie durcheinanderbringt — sich selbst wie von Geisterhand immer auf einen vorher genau definierten Zustand einpendeln.

Die Systeme, welche Dijkstra betrachtete, bestanden jeweils aus einem Ring von Prozessoren mit gemeinsamem Speicher, und der genau definierte Zustand war der wechselseitige Ausschluß auf dem Ring. Natürlich hat es solche Systeme auch schon vor 1974 gegeben und eines der bekanntesten Konzepte zur Lösung des Problems war bereits sechs Jahre früher von Dijkstra selbst unter dem Namen "Semaphore" vorgestellt worden [48]. Der kleine, aber entscheidende Unterschied zwischen dem alten und dem neuen "Szenario" bestand aber darin, daß der gemeinsame Speicher der Prozesse nur noch "teilweise gemeinsam" war; Prozesse teilten sich nur mit ihren direkten Nachbarn gemeinsame Speicherbereiche, Kommunikation und Synchronisation mit den restlichen Prozessen war nur indirekt möglich. Und so schieden die herkömmlichen Lösungen aus und es entstand der Bedarf nach etwas, was Dijkstra damals *distributed control* nannte, also die Notwendigkeit, aufgrund von lokaler Information und lokalen Aktionen ein globales Ziel, also beispielsweise wechselseitigen Ausschluß zu erreichen.

Natürlich gab es zur damaligen Zeit auch schon solche verteilten Algorithmen, doch Dijkstra hatte beobachtet, daß

> "[...] when once [they were] (erroneosly) in an illegitimate state, they could—and usually did!—remain so forever." [49, S. 643]

Die Fähigkeit, aus diesen "illegalen" Zuständen wieder herauszukommen und im normalen Ablauf des Algorithmus fortzufahren, nannte Dijkstra damals *self-stabilization* und prägte damit einen Begriff, der heute im Zusammenhang mit der Fehlertoleranz in verteilten Systemen wieder von großem Interesse ist (siehe Abschnitt 4.2.4). Genauer schreibt er:

> "We call a system 'self-stabilizing' if and only if, regardless of the initial state [...] the system is guaranteed to find itself in a legitimate state after a finite number of moves." [49, S. 643]

Mit den "legalen" oder "sicheren" Zuständen sind die Bereiche der Ausführung gemeint, in denen die Invariante des Algorithmus gilt, also im Beispiel des wechselseitigen Ausschlusses diejenigen Zustände, in denen zu jeder Zeit nur maximal ein einziger Prozeß ein exklusives Zugriffsrecht (im Englischen manifestiert durch den Ausdruck *token*) besitzt. Illegal wären dann Zustände, in denen kein oder mehrere solcher *tokens* im System vorhanden sind. Natürlich muß diese Definition noch verfeinert und geeignet formalisiert werden, doch will ich zunächst zur Verdeutlichung im folgenden Abschnitt das erste der drei Systeme Dijkstras vorstellen.

4.2.1 Dijkstras Algorithmus

In seinem Artikel [49] stellt Dijkstra drei zunehmend komplexere verteilte Algorithmen vor, die alle wechselseitigen Ausschluß auf einem Ring von $N+1$ Prozessen erreichen. Die drei Lösungen unterscheiden sich in der Zahl der Zustände, die die einzelnen Prozesse jeweils haben müssen, angefangen von einer beliebigen Zahl K (mit $K > N$), gefolgt von einer Lösung mit nur noch vier Zuständen. Abgeschlossen wird der Artikel mit einer Lösung, die nur noch drei Zustände benötigt. In dem Systemmodell, welches Dijkstra benutzt, können die Prozesse nur auf den eigenen Zustand und auf den lokalen Zustand ihres jeweiligen linken Nachbarn zugreifen. Da sich die einzelnen Anweisungen in den Prozessen gegenseitig behindern, nimmt Dijkstra an, daß keine zwei Aktionen gleichzeitig ausgeführt werden dürfen.

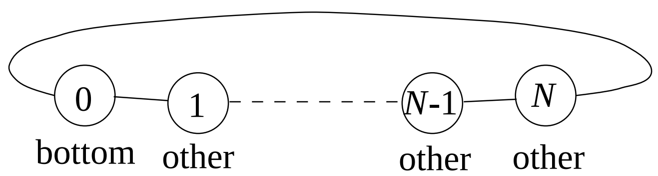

Abbildung 4.2: Ring-Topologie mit Bezeichnungen für Dijkstras Algorithmus aus Beispiel 4.3.

4.3 Beispiel (Dijkstras Token-Ring mit K Zuständen) Gegeben ist ein Ring von $N+1$ Prozessen (dargestellt in Abbildung 4.2). Die Prozesse sind Automaten mit K Zuständen (jeweils gespeichert in der lokalen Variablen S) und können auf die lokale Variable ihres linken Nachbarn zugreifen (dargestellt durch die Variable L). Bezeichnet man beispielsweise den Zustand von Prozeß i mit S_i, ist

$$L_i = S_{(i-1) \bmod (N+1)}.$$

Alle Prozesse haben zwar eine unterschiedliche Prozeßidentität, sind aber ansonsten identisch, bis auf den Prozeß mit der Nummer 0, auf dem ein anderer lokaler Algorithmus abläuft. Dijkstra bezeichnet diese beiden Typen von Prozessen mit *bottom machine* für Prozeß 0 und *other machines* für die anderen (vergleiche Abbildung 4.2). Der lokale Algorithmus von Prozeß 0 ist in Abbildung 4.3 dargestellt. Der Algorithmus, der auf allen anderen Maschinen läuft, folgt in Abbildung 4.4. Die Bezeichnung *CS* steht für *critical section* und bezeichnet diejenigen Aktionen, die nur mit dem exklusiven Zugriffsrecht ausgeführt werden können. Die boolesche Bedingung davor zeigt also gewissermaßen an, unter welchen Voraussetzungen das *token* in Besitz des Prozesses ist. Mit den dann folgenden Anweisungen verletzt der Prozeß absichtlich wieder diese Voraussetzungen und gibt das *token* wieder ab, damit auch die anderen Prozesse die Chance haben, einmal an die Reihe zu kommen.

process *bottom*
  var $S \in \{0 \dots K-1\}$
  begin
    $L = S \quad \longrightarrow \quad CS;\ S := (S+1) \bmod K$
  end

Abbildung 4.3: Der lokale Algorithmus des *bottom*-Prozesses.

process *other*
  var $S \in \{0 \dots K-1\}$
  begin
    $L \neq S \quad \longrightarrow \quad CS;\ S := L$
  end

Abbildung 4.4: Der lokale Algorithmus der *other*-Prozesse.

Die Funktionsweise des Algorithmus läßt sich am besten an einem konkreten Ablauf verdeutlichen. In Tabelle 4.1 ist für die Werte $N = 3$ (also vier Prozesse!) und $K = 4$ ein solcher Ablauf aufgezeichnet. Die Zustände der Prozesse nach jeweils einem atomaren Schritt sind in den vier rechten Spalten der Tabelle eingetragen und wurden anfangs zufällig gewählt. Die Pfeile symbolisieren einen Zugriff auf den linken Nachbarn und deuten an, daß die Bedingung für den Besitz des *tokens* erfüllt sind. Man kann sich jetzt einen der Pfeile auswählen und die dazugehörige Zustandsänderung innerhalb des jeweiligen Prozesses ausführen. Der jeweils ausgeführte Schritt ist durch durch einen Pfeil mit Haken dargestellt (nach Voraussetzung dürfen keine zwei Aktionen gleichzeitig stattfinden). In der Tabelle

Prozeßnummer	0	1	2	3
Zustände	3	$\hookleftarrow$ 0	$\leftarrow$ 2	$\leftarrow$ 1
	3	3	$\leftarrow$ 2	$\hookleftarrow$ 1
	3	3	$\hookleftarrow$ 2	2
	3	3	3	$\hookleftarrow$ 2
	$\hookleftarrow$ 3	3	3	3
	0	$\hookleftarrow$ 3	3	3
	0	0	$\hookleftarrow$ 3	3
	0	0	0	$\hookleftarrow$ 3
	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Tabelle 4.1: Ein möglicher Ablauf des Algorithmus aus Beispiel 4.3 mit $N = 3$ und $K = 4$.

ist es zuerst Prozeß 1, dann anschließend Prozeß 3. Das Ergebnis ist wie erwartet, daß der Pfeil jeweils verschwindet. Nun bleibt keine Wahl mehr und nur noch Prozeß 2 kann eine Zustandsänderung vornehmen. Dies bewirkt zwar wieder das Verschwinden des eigenen Pfeiles, doch durch die eigene Zustandsänderung bewirkt er, daß die *token*-Bedingung seines rechten Nachbarn (Prozeß 3), der ja auf seinen Zustand zugreift, plötzlich wahr wird; das *token* wird quasi weitergereicht.

Zu beachten ist, daß hier schon nach zwei Zustandsänderungen (also ab Zeile drei der Tabelle) das System bereits sein Ziel erreicht hat: *Ein* Pfeil bedeutet, daß nur ein *token* im System ist; wechselseitiger Ausschluß ist garantiert. Man kann den Algorithmus auch noch an anderen Beispielen durchspielen, wird aber immer wieder auf dasselbe Ergebnis kommen.

4.2.2 Formale Definition

Mit der Präsentation des Algorithmus von Dijkstra und eines Beispielablaufes ist natürlich noch kein Beweis erbracht, daß er tatsächlich auch das Geforderte (=Selbststabilisierung) leistet. Hierfür muß allerdings der Begriff der Selbststabilisierung noch präzisiert werden. In diesem Abschnitt will ich ein paar gängige Definitionen vorstellen und sie untereinander vergleichen.

Es gibt unterschiedliche Möglichkeiten, die sogenannten "sicheren" und "unsicheren" Zustände zu unterscheiden. Eine gängige Möglichkeit ist die Benutzung eines logischen Prädikates P auf der Menge der Konfigurationen C. Das Prädikat, dargestellt durch einen booleschen Ausdruck (oder äquivalent durch eine Menge von Konfigurationen) ist eine Art Spezifikation des Problems: Beispielsweise könnte man für das Problem des wechselseitigen Ausschlusses P informal definieren als:

$$P = \{c \in C | \text{in Konfiguration } c \text{ ist nur ein } token \text{ vorhanden}\}$$

Eine Standarddefinition ist die folgende:

4.4 Definition (Selbststabilisierung nach Marco Schneider [126, S. 47]) Ein System S stabilisiert sich selbst zu einem Prädikat P auf Konfigurationen C, gdw. die folgenden beiden Bedingungen gelten:

- Falls P in der Konfiguration c gilt, dann gilt P auch in jeder Folgekonfiguration von c.
- Ausgehend von einer beliebigen Systemkonfiguration wird S in einer endlichen Anzahl von Schritten eine Konfiguration erreichen, in der P gilt.

Die beiden Bedingungen werden *Abgeschlossenheit* (*closure*) und *Konvergenz* (*convergence*) genannt.

Die zweite Bedingung stellt sicher, daß das System irgendwann mit Sicherheit zu einem legalen Gesamtzustand konvergiert, während die erste Bedingung sicherstellt, daß darauf nur noch sichere Zustände folgen werden. Wichtig ist, daß in der Konvergenz-Bedingung keinerlei Aussage über den Startzustand gemacht wird; es könnte *jeder* Zustand sein. Damit versucht man das zu formalisieren, was Dijkstra mit "regardless of it's initial state" meinte. Ein solcher "zufälliger" Zustand modelliert die Auswirkungen von Fehlern implizit mit.

Es ist interessant zu beobachten, daß selbststabilisierende Systeme keinerlei Initialisierung benötigen, da sie aus jedem Zustand irgendwann in einen legalen Zustand übergehen. Manchmal wird diese Eigenschaft als alternative Definitionsmöglichkeit angeführt [17, S. 268]. Konsequenterweise modelliert Tel selbststabilisierende Algorithmen als Automaten ohne Startzustände. Die Korrektheitsbedingung, also die Unterscheidung der legalen und illegalen Zustände, definiert er hier — anders als Definition 4.4 — durch eine Eigenschaft, also eine Menge von Konfigurationsfolgen.

4.5 Definition (Selbststabilisierung nach Tel [136, S. 460]) Im Zusammenhang mit Selbststabilisierung ist ein System S ein Automat (C, δ) ohne Startzustände, wobei C die Menge der Konfigurationen und δ die Zustandsübergangsrelation ist. Eine Ausführung von S ist eine maximale Folge $E = (c_0, c_1, c_2, \dots)$ von Konfigurationen, wobei für alle i gilt: $(c_i, c_{i+1}) \in \delta$.

Ein solches System stabilisiert sich selbst zu einer Eigenschaft P, falls es eine Teilmenge $L \subseteq C$ von "legalen" Konfigurationen gibt, für die die folgenden beiden Bedingungen gelten:

1. Jede Ausführung, die in L beginnt, hat die Eigenschaft P.
2. Jede Ausführung enthält eine Konfiguration aus L.

Die beiden Bedingungen werden *Korrektheit* (*correctness*) und *Konvergenz* (*convergence*) genannt.

Ein System, welches dieser Definition von Selbststabilisierung genügt, wird nach endlicher Zeit ein bestimmtes, gewolltes Verhalten zeigen. (Dies ist übrigens auch das, was Leslie Lamport unter Selbststabilisierung versteht [103, S. 335].)

Unterschiede zwischen den Definitionen. Oberflächlich besteht der wesentliche Unterschied zwischen Schneiders Definition (4.4) und der von Tel (4.5) in der beabsichtigten Wirkung der Selbststabilisierung, ausgedrückt durch das "Zielprädikat" P: Bei Schneider geht es darum, eine *safety*-Eigenschaft zu erfüllen, also eine Menge von "sicheren" Zuständen möglichst nie zu verlassen. Im zweiten Fall soll das System sich zu einer beliebigen (*safety*- oder *liveness*-) Eigenschaft stabilisieren, welche sich durch eine Menge L von "legalen" Zuständen bloß charakterisieren läßt. Die Menge L muß nicht abgeschlossen sein, kann also auch im korrekten Systemablauf durchaus verlassen werden. Wesentlich ist, daß Ausführungen, die in L starten, die Eigenschaft P haben.

Man kann darüber streiten, welche Definition die praktischere ist; Tatsache ist jedoch, daß nur wenige sinnvolle Spezifikationen für die Selbststabilisierung gibt, die sich nicht als Prädikat auf Konfigurationen beschreiben ließen. Das bedeutet, daß die Definitionen von Schneider und Tel eigentlich gleichwertig sind, obwohl natürlich Tels Definition allgemeiner ist. (Tel selbst schreibt sogar: "The set of legitimate configurations is usually closed [...]" [136, S. 460].) Auch entspricht die Vorstellung von einer Menge von "sicheren" Zuständen, die nie verlassen wird, eher meiner Intuition als die Vorstellung, daß diese auch verlassen werden dürfen. Man darf aber nicht vergessen, daß sich Systeme unter diesen Vorzeichen nur zu *safety*-Eigenschaften stabilisieren können.

Alternative Definitionen. Es gibt neuerdings auch alternative Definitionen der Selbststabilisierung, die verschiedene Grade dieser Eigenschaft vorsehen. Verschiedene Grade der Selbststabilisierung sind vorteilhaft aus zwei Gründen [77, S. 4]:

1. Die Forderung nach vollständiger Selbststabilisierung, wie sie bisher immer betrachtet wurde, erfordert in manchen Situationen sehr komplexe und damit "teure" Algorithmen oder mag in Anbetracht von existierenden Unmöglichkeitsresultaten gar nicht realisierbar sein. In solchen Fällen kann man schwächere Formen dieser Eigenschaft betrachten.

2. Verschiedene Grade der Selbststabilisierung wären ein erster Schritt, um selbststabilisierende Systeme strukturell zu vergleichen und gegenüberzustellen.

Einen fundierten Ansatz hierzu lieferte Gouda 1995 mit seiner "Theorie der Selbststabilisierung" [77], deren Stabilitätsdefinitionen verschiedene Grade von Abgeschlossenheit und Konvergenz enthalten [77, S. 2]. In das von ihm aufgezeigte Spektrum lassen sich auch probabilistische Selbststabilisierung [77, S. 3] und das von Burns, Gouda und Miller vorgeschlagene Konzept der *Pseudo-Stabilisierung* [31] einordnen.

Goudas verschiedene Grade der Selbststabilisierung sind nur ein Teil seiner Theorie, die zusätzlich noch Gesetzmäßigkeiten, Beweisverpflichtungen und ein Kompositionstheorem enthält und sicherlich als ein Meilenstein auf dem Weg zu einem besseren Verständnis der Thematik angesehen werden kann. Doch trotz der theoretischen Eleganz ist ein Kritikpunkt anzumerken. Er betrifft die praktische Anwendbarkeit seiner Resultate. Bisher sind nämlich bis auf einige wenige Ausnahmen ausschließlich Algorithmen veröffentlicht worden, die dem ursprünglichen Verständnis des Sachverhaltes entsprechen. Eine sinnvolle Anwendung für die anderen Grade ist noch nicht recht abzusehen. Außerdem mag es noch andere Formen der Selbststabilisierung geben, die nicht durch diese Definitionen abgedeckt werden. All dies sind offene Forschungsprobleme.

4.2.3 Beweis der Selbststabilisierung

Nach der formalen Fassung der Selbststabilisierung kann man nun auch nachweisen, daß bestimmte Algorithmen sich selbst stabilisieren. Als Beispiel will ich wieder Dijkstras Algorithmus aus Beispiel 4.3 heranziehen.

Zum Nachweis der Selbststabilisierungseigenschaft und damit der Korrektheit eines selbstabilisierenden Algorithmus muß man in der Regel zwei getrennte Beweise führen:

1. einen herkömmlichen Beweis der *safety*, also der Abgeschlossenheit des Prädikates P, und

2. einen Beweis der *liveness*, also der Konvergenz zu einem legalen Zustand.

Bei beiden werden die klassischen Mittel der Programmverifikation in erweiterter Form wiederverwendet: Invarianten und Terminierungsfunktionen. Im ersten Fall ist die Invariante gerade das Prädikat, welches die legalen Zustände beschreibt; man muß zeigen, daß die Aktionen, die in legalen Zuständen ausgeführt werden, die Invariante nicht verletzen. Im zweiten Fall benötigt man eine Funktion von den Konfigurationen in eine wohlfundierte Menge (beispielsweise die natürlichen Zahlen) und zeigt, daß sie streng monoton fällt und am Ende die Invariante gilt.

Betrachtet man den Algorithmus von Dijkstra, werden die beiden Teilbedingungen durch die nächsten beiden Lemmata nachgewiesen. Zuvor muß jedoch der Begriff "wechselseitiger Ausschluß" noch definiert werden.

4.6 Definition (Wechselseitiger Ausschluß [103, S. 328f]) Wechselseitiger Ausschluß ist eine Eigenschaft P, die folgenden Bedingungen genügt:

- Zu jeder Zeit hat maximal ein Prozeß das *token*.
- Ein Prozeß wartet nur dann unendlich lange auf das *token*, wenn das *token* von den anderen Prozessen unendlich oft hin- und hergereicht wird.
- Wenn ein Prozeß das *token* haben will, erhält er es auch nach endlicher Zeit.

Während die erste Bedingung den eigentlichen wechselseitigen Ausschluß garantiert, schützt die zweite Bedingung vor den sogenannten *Verklemmungen* (*deadlocks*), in denen einer oder mehrere Prozesse das *token* haben wollen, aber kein Prozeß es jemals erhält. Die letzte Bedingung ist eigentlich überflüssig, doch erzwingt sie eine einfache Art von Fairness auf den Lösungen.

Eine wichtige Beobachtung vorweg: Es läßt sich zeigen, daß es bei Dijkstras Algorithmus keinen Systemzustand gibt, in dem das *token* gänzlich fehlt. Um dies zu sehen, nimmt man an, es gäbe kein *token* im System. Das bedeutet, daß alle *token*-Bedingungen der Prozesse verletzt sein müssen. Also muß $L_{i-1} = S_i$ für $1 \leq i \leq N$ und $L_N \neq S_0$ gelten. Wenn aber S_0 den Wert a hat, haben alle anderen S_i durch die erste Bedingung auch den Wert a, insbesondere S_N. Somit gilt aber auch $S_N = L_N = a = S_0$ und wegen $L_N = S_0$ hat Prozeß 0 das *token*. Ein *deadlock*, wie ihn die zweite Bedingung des wechselseitigen Ausschlusses verbietet, kann somit nie auftreten. Es genügt also der Nachweis der ersten Bedingung, die somit zum Legalitätsprädikat P wird. Eine Betrachtung der Fairness ist Teil des folgenden Lemmas 4.7.

4.7 Lemma (Abgeschlossenheit von Dijkstras Algorithmus) Dijkstras Algorithmus aus Beispiel 4.3 ist abgeschlossen unter wechselseitigem Ausschluß.

Beweis Für den Beweis genügt es zu zeigen, daß in einem System A das Prädikat P bei fehlerfreier Ausführung des Algorithmus nicht verlassen wird.

Sei A also in einer Konfiguration $c \in P$. Dies bedeutet, genau ein Prozeß i hat das *token*. Dieser Prozeß wird seinen kritischen Abschnitt ausführen und anschließend die eigene *token*-Bedingung durch die Zuweisung verletzten, also das *token* abgeben. Da es keinen Zustand des Systems gibt, in dem kein *token* existiert, muß es nun in Besitz eines anderen Prozesses sein.

Zur Beantwortung der Frage, an wen das Zugriffsrecht weitergegeben wird, genügt folgende Betrachtung: Bis auf die Variable S von Prozeß i ändert sich in diesem Systemschritt nichts. Die einzigen Prozesse, die diese Änderung beeinflussen könnten, sind der Prozeß i selbst und sein rechter Nachbar $(i + 1) \bmod (N + 1)$, da nur diese auf die Variable S zugreifen. Prozeß i hat das *token* anschließend nicht mehr, darum muß es sein linker Nachbar haben. So wird das Zugriffsrecht jeweils immer im Kreis weitergereicht und jeder Prozeß kommt innerhalb von $N + 1$ Konfigurationen jeweils einmal an die Reihe. Damit ist auch die Fairness-Bedingung erfüllt. $\square$

4.8 Lemma (Konvergenz von Dijkstras Algorithmus) Dijkstras Algorithmus aus Beispiel 4.3 wird in einer endlichen Anzahl von Schritten eine Konfiguration erreichen, in der wechselseitiger Ausschluß gilt.

Beweis Da $K > N$ gilt, gibt es im System ohne Prozeß 0 mindestens $K - N$ Zustände, die von keinem anderen Prozeß benutzt werden und die ich als *neue* Zustände bezeichnen möchte. Da die *other*-Prozesse immer nur Zustände kopieren, kann nur der *bottom*-Prozeß irgendwann einen dieser neuen Zustände annehmen. Für den Beweis genügt es darum, die folgenden beiden Behauptungen zu zeigen:

(1) Sobald der *bottom*-Prozeß einen neuen Zustand annimmt, geht das

System innerhalb endlicher Zeit in einen Zustand mit wechselseitigem Ausschluß über.

(2) Der *bottom*-Prozeß nimmt innerhalb endlicher Zeit einen solchen neuen Zustand an.

Zu (1): Sagen wir, der *bottom*-Prozeß ist in einem Zustand, den es im Ring noch nicht gibt. In diesem Fall wird er das *token* erst wieder bekommen, wenn alle anderen Prozesse diesen Zustand in ihre eigene Zustandsvariable kopiert haben, also $S_1 = S_2 = \ldots = S_N = S_0$ gilt. Um dies zu zeigen, kann man die folgende Terminierungsfunktion benutzen:

$$t' = \sum_{i=1}^{N} p(i)$$

wobei

$$p(i) = \begin{cases} (N-i) & \text{falls Prozeß } i \text{ das } \textit{token} \text{ hat} \\ 0 & \text{sonst} \end{cases}$$

Da in jedem Schritt Prozeß i das *token* an Prozeß $i+1$ abgibt, wird mit jedem Schritt jeweils ein Summand $N-i$ durch $N-i-1$ ersetzt, wodurch t' jeweils um 1 erniedrigt wird. Im Falle $t' = 0$ gilt wie bereits gesagt $S_1 = S_2 = \ldots = S_N = S_0$ und das System ist somit in einem legalen Zustand.

Zu (2): Um zu sehen, daß der *bottom*-Prozeß irgendwann einen neuen Zustand annimmt, zeigt man, daß jede Ausführung unendlich viele Zustandsänderungen von Prozeß 0 enthält. Dazu benutzt man die Funktion t' aus dem ersten Teilbeweis, um zu zeigen, daß zwischen zwei Schritten von Prozeß 0 nur endlich viele Schritte anderer Prozesse stattfinden können. Ausführungen des Dijkstra-Algorithmus terminieren nicht und sind somit immer unendlich. Nach jeweils endlich vielen Schritten anderer Prozesse kommt somit der *bottom*-Prozeß immer wieder an die Reihe und inkrementiert seinen Zustandszähler modulo K. Prozeß 0 kommt darum nach spätestens $N+1$ eigenen Zustandsänderungen in einen neuen Zustand. □

Der folgende Satz folgt aus den beiden Lemmata und der Definition 4.4.

4.9 Satz (Korrektheit von Dijkstras Algorithmus aus Beispiel 4.3) Dijkstras Algorithmus aus Beispiel 4.3 stabilisiert sich selbst zu wechselseitigem Ausschluß.

4.2.4 Eine kurze Geschichte der Selbststabilisierung

Wie in der Einleitung dieses Kapitels bereits angedeutet wurde, ist das Paradigma der Selbststabilisierung ein sehr natürliches Prinzip, welches man oft in der Natur beobachten kann. In den formalen Wissenschaften ist dieses Phänomen auch schon relativ früh in der Mathematik untersucht worden, jedoch nicht unter dem Gesichtspunkt der Fehlertoleranz sondern unter dem Blickwinkel der Konvergenz [79, S. 449]. Gouda und Multari führen in diesem Zusammenhang das bekannte Newton-Rhapson-Verfahren zur numerischen Nullstellenbestimmung an. Es gehört zur Klasse der einfachen Interationsverfahren und kann vereinfacht durch folgenden Pseudocode beschrieben werden:

while $f(x) \neq 0$ **do begin**
  $x := x - (f(x)/f'(x))$
end

Das Verfahren konvergiert gegen eine Lösung von $f(x) = 0$ genau dann, wenn die Funktion f eine bestimmte Konvergenzbedingung erfüllt, sich also in gewisser Weise "freundlich" verhält. Die Analogie zur Selbststabilisierung besteht darin, daß man die Zustände x mit $f(x) = 0$ als sichere Zustände betrachtet und alle anderen als unsichere Zustände. Die Funktion f wäre dann in diesem Sinne "selbststabilisierend," falls man nachweisen kann, daß sie die Konvergenzbedingung erfüllt, also nach einer endlichen Anzahl von Interationen in einen sicheren Zustand übergeht. Die Eigenschaften derartiger Iterationssysteme (*iteration systems*) sind in einem Artikel von Arora, Attie, Evangelist und Gouda [7] bereits grundlegend mathematisch untersucht worden.

Die Anwendung dieser Technik auf die Informatik begann mit dem kurzen, aber gehaltvollen Artikel von Edsger W. Dijkstra 1974 in den *Communications of the ACM*. Lange Zeit wurde die praktische Anwendbarkeit dieses Prinzips nicht erkannt. Erst Mitte der achtziger Jahre begann das Interesse daran wieder aufzuleben. Vor allem den Schülern und Kollegen von Dijkstra an der Universität von Texas ist es zu verdanken, daß dieses Gebiet Anfang der neunziger Jahre einen starken Aufschwung erfuhr und als theoretisches Konzept zur Beschreibung und Realisierung von Fehlertoleranz in Computersystemen Geltung erlangt hat. Die aktivsten Forschergruppen auf diesem Gebiet befinden sich heute an der Universität von Texas in Austin (Prof. Mohamed Gouda) sowie an der Universität von Iowa (Prof. Sukumar Ghosh). Ted Herman von der Universität Iowa hat eine umfassende Bibliographie zu diesem Thema zusammengestellt, die im Internet frei erhältlich ist [86]. Die letzte Version dieser Bibliographie vom 5. Juli 1997 enthielt 234 Einträge, die in Abbildung 4.5 in einem Balkendiagramm dargestellt sind: Die Höhe eines Balkens entspricht der Anzahl der Veröffentlichungen zu diesem Thema in einem bestimmten Jahr. Hier ist deutlich zu sehen, wie zu Beginn der neunziger Jahre die Zahl der publizierten Artikel sprunghaft angestiegen ist. Dies muß zwar nicht mit einer entsprechend rasanten Fortschritt auf dem Gebiet einhergehen, doch sind in relativ kurzer Zeit schon einige beachtliche Ergebnisse erzielt worden, zu denen mit Goudas "Theorie der Selbststabilisierung” [77] auch eine fundamentale und vereinheitlichte Theorie zur Verfügung steht. Foren für die Diskussion derartiger Themen sind die jährlich stattfindenden Symposien *Principles of Distributed Computing* und die Arbeitstreffen *Workshop on Distributed Algorithms*. In unregelmäßigen Abständen findet auch ein *Workshop on Self-Stabilizing Systems* statt, der bisher drei Mal stattgefunden hat (1989 an der Universität von Texas in Austin, 1995 an der Universität von Nevada in Las Vegas und 1997 an der Universität von Kalifornien in Santa Barbara).

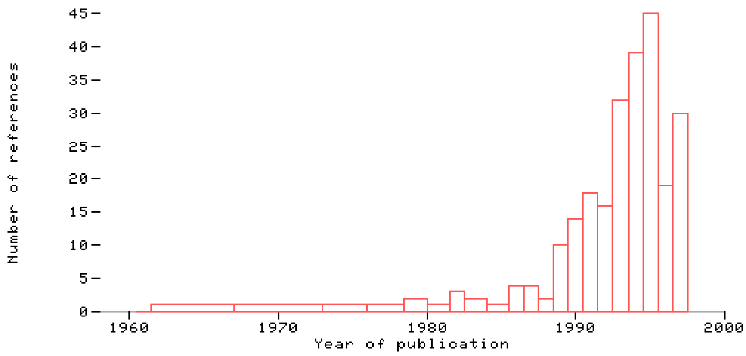

Abbildung 4.5: Verteilung der Einträge in Ted Hermans umfassender Bibliographie über Selbststabilisierung nach Jahreszahl (Stand: 5. Juli 1997).

4.2.5 Ausgewählte Beispiele

Aus der großen Menge selbststabilisierender Algorithmen, die es schon gibt, möchte ich jetzt drei Beispiele genauer präsentieren. Die Auswahl ist nicht repräsentativ für das Gesamtgebiet, doch habe ich drei verschiedene Verfahren für unterschiedliche Problemstellungen und Modelle ausgewählt, um durch die genauere Analyse ein gewisses Gefühl für die ihnen eigene Wirkungsweise und die dahinterstehenden Entwurfsprinzipien zu vermitteln. Im Abschnitt 4.3.3 werde ich dann einen kurzen Überblick über die wichtigsten Algorithmen präsentieren, zu denen mir Informationen vorlagen.

Ein selbststabilisierender aufspannender Baum. Chen, Yu und Huang präsentieren in ihrem Artikel “A self-stabilizing algorithm for constructing spanning trees” [39] einen ebensolchen. Das Modell und die Voraussetzungen sind im wesentlichen die gleichen wie bei Dijkstras Algorithmus für den wechselseitigen Ausschluß im Ring (vgl. Beispiel 4.3): n Prozesse kommunizieren über lokal gemeinsame Variablen in einem beliebig zusammenhängenden Kommunikationsgraphen, es gibt einen ausgewählten Prozeß *root* als auf allen anderen, und wieder beeinflussen sich die einzelnen Anweisungen gegenseitig (nur das gleichzeitige Ausführen einer bewachten Anweisung ist erlaubt).

4.10 Algorithmus (Berechnung eines selbststabilisierenden aufspannenden Baumes) Der aufspannende Baum, den der Algorithmus herstellen soll, ist in Form lokaler Variablen p und l gespeichert, wobei p (für *parent*) die Knotenidentität des "Vorgängers" im Baum enthält und l (für *level*) die Ebene des Knotens im Baum (die Länge des Pfades bis zur Wurzel). Jeder Prozeß kennt die maximale Zahl n von Prozessen und auch seine jeweiligen Nachbarn (gespeichert in der Menge *neighbours*). Ziel ist es, daß die Werte der Variablen p in jedem Prozeß so gesetzt werden, daß der dadurch definierte Graph einen aufspannenden Baum mit Wurzel-Prozeß *root* bildet und daß die jeweiligen l-Variablen jeweils die korrekte Stufe anzeigen. Ausgedrückt wird diese Tatsache durch das Legalitätsprädikat:

$$P \equiv \forall i, p \in Nodes . i \neq root \wedge p = p_i \Rightarrow l_i = l_p + 1$$

Die lokalen Algorithmen der Wurzel und der anderen Knoten sind in den Abbildungen 4.6 und 4.7 dargestellt. Die Wurzel selbst muß keine konkreten Anweisungen ausführen, sondern lediglich ihre konstante Stufe 0 ihren Nachbarn zur Verfügung stellen. Der Zugriff auf die gemeinsamen Variablen ist durch eine Index-Notation dargestellt: beispielsweise steht l_i für den Inhalt der Variablen l von Nachbarknoten i. (Ein solcher Zugriff ist natürlich nur erlaubt, falls $i \in neighbours$.) Die Schreibweise $m \in neighbours$ in der dritten bewachten Anweisung in Abbildung 4.7 soll ausdrücken, daß m ein beliebiger Nachbar des Knotens ist. Strenggenommen müßte diese Zeile n Mal geschrieben werden, wobei m alle möglichen Knotenidentitäten durchläuft.

Die Idee des Algorithmus ist einfach: Falls ein lokaler Knoten die Verletzung des Legalitätsprädikates feststellt (also $l \neq l_p + 1$), setzt er seine eigene Variable l auf die Stufe seines Elternknotens plus 1. Genau dies wird durch die erste Anweisung erreicht. In einem aufspannenden Baum mit n Knoten gibt es maximal $n - 1$ Kanten; die maximale Stufe, die ein beliebiger Knoten erreichen kann, ist also $n - 1$. Der Wert $l = n$ kann also dazu benutzt werden, um einen Fehlerzustand zu signalisieren. Der Algorithmus ist so konstruiert, daß ein Fehlerzustand in Richtung der Blätter des Baumes "weitergemeldet" wird. Hierzu dient die zweite Anweisung. Schließlich wird durch die dritte Anweisung garantiert, daß ein Knoten aus dem Fehlerzustand herauskommen kann, indem er sich einen neuen Elternknoten aus der Menge seiner Nachbarn aussucht, der nicht in einem Fehlerzustand ist.

Dieser Algorithmus ist interessant, da er die Grundidee aller Spannbaum-Konstruktionsalgorithmen in Reinform enthält: Durch das ständige Überprüfen und die ständige Korrektur der lokalen Bedingung $l = l_p + 1$ wird erreicht, daß ein legaler Zustand relativ schnell erreicht wird, wenn die jeweiligen Verweise in den Variablen p bereits einen aufspannenden

```
process root
  const l init 0
  begin

  end
```

Abbildung 4.6: Der lokale Algorithmus des Wurzelknotens bei der selbststabilisierenden Spannbaumkonstruktion.

process *node*
const n **init** {* number of nodes in network *}
neighbours **init** $\{n_1, n_2, \ldots, n_k\}$
var $l \in \{1, 2, \ldots, n\}$
$p \in neigbours$
begin
{* adjust node level: *}
$l \neq n \wedge l \neq l_p + 1 \wedge l_p \neq n \quad \longrightarrow \quad l := l_p + 1$
{* propagate errors: *}
$[\!] \; l \neq n \wedge l_p = n \quad \longrightarrow \quad l := n$
{* choose new parent m: *}
$[\!] \; m \in neighbours \wedge l = n \wedge l_m < n - 1 \quad \longrightarrow \quad l := l_m + 1;\; p := m$
end

Abbildung 4.7: Der lokale Algorithmus der Nicht-Wurzelknoten bei der selbststabilisierenden Spannbaumkonstruktion.

Baum bilden. Falls dies aber nicht der Fall ist, zerfällt der Graph in eine Menge von Teilbäumen, die — solange der Wurzelknoten *root* nicht dazugehört — einen Zyklus enthalten müssen. Doch innerhalb eines solchen Zyklus' schaukeln sich die Werte von l gegenseitig hoch, so daß irgendwann ein Fehlerzustand $l = n$ erreicht wird. Nach einer gewissen Zeit gehen alle Knoten innerhalb des Teilbaumes in den Fehlerzustand über und durch die dritte Anweisung verschmilzt dieser Teilbaum dann durch die Wahl einer anderen Verweisstruktur mit einem anderen Teilbaum, der (noch) nicht fehlerhaft ist. Irgendwann führt dies dazu, daß sich alle Teilbäume mit der Wurzel verschmelzen. Der vollständige Korrektheitsbeweis und ein ausführliches Beispiel kann im Originalartikel [39] nachvollzogen werden.

Ein selbststabilisierendes Kommunikationsprotokoll. In Computernetzwerken werden Daten zwischen benachbarten Rechnern in Form von genau festgelegten Bitsequenzen (Paketen) über das physikalische Kommunikationsmedium (Kupferdraht, Lichtwellenleiter etc.) übertragen. Die Regeln, nach denen dies geschieht, nennt man ein *Kommunikationsproto-*

koll. Zu den Aufgaben derartiger Protokolle gehören beispielsweise die Sicherstellung, daß die Pakete in der richtigen Reihenfolge und unverändert beim Empfänger eintreffen, oder daß die zur Verfügung stehende Bandbreite des Mediums möglichst effizient ausgenutzt wird.

Eines der einfachsten Protokolle zur Datensicherung, Flußsteuerung und Reihenfolgegarantie über unzuverlässige Kommunikationsleitungen ist das *stop and wait*-Protokoll [134, S. 143ff]. Die Idee des Protokolls ist, daß Sender und Empfänger sich bei jedem versandten Paket neu synchronisieren: Versandte Pakete werden mit einer fortlaufenden Sequenznummer versehen und vom Empfänger jeweils explizit unter Angabe der Sequenznummer durch Rücknachrichten (*acknowledgements*) bestätigt. Problematisch wird es für das Protokoll nur, wenn die lokalen Zustände von Sender und Empfänger durch vorübergehende Störeinflüsse (elektromagnetische Strahlung, Leistungsspitzen in der Spannungsversorgung etc.) korrumpiert worden sind. In diesem Fall kann es möglich sein, daß der Empfänger auf eine Sequenznummer wartet, die der Sender nie losschicken wird. Der Empfänger wird also nie eine positive Bestätigung zurückschicken. Hier ist die Situation gegeben, daß beide Stationen erst durch explizites Rücksetzen (von außen) wieder in einen synchronisierten Zustand gelangen können. In diesen Fällen kann Selbststabilisierung helfen.

Kommunikationsprotokolle sind bereits relativ früh bezüglich ihrer Selbststabilisierungseigenschaft untersucht worden, mit dem Ergebnis, daß sie diese Eigenschaft im allgemeinen nicht aufwiesen. Ich möchte hier eine einfache selbststabilisierende Variante des *stop and wait*-Protokolls vorstellen, wie es in den Veröffentlichungen von Gouda und Multari [79, S. 451f], Afek und Brown [2, S. 33], sowie Costello und Varghese [43, S. 39] vollständig oder in Teilen jeweils neu auftaucht.

In dem Szenario gibt es zwei Prozesse, einen Sender (*sender*) und einen Empfänger (*receiver*), die über eine unzuverlässige Nachrichtenleitung zuverlässig kommunizieren sollen. Das Attribut "zuverlässig" meint hier: unverfälscht, ohne Verdoppelung oder Verlust von Paketen und in der richtigen Reihenfolge. Der Sender hat Zugriff auf eine unendliche Folge $data[1, 2, \ldots]$ von Datenelementen vom Typ $Data$. Der Empfänger liefert mittels einer internen Aktion $deliver(d)$ ein Datenelement d auf an seine Anwendungsschicht. Die Korrektheitsforderung des Protokolls in einer fehlerfreien Umgebung ist, daß die Ausgabe des Empfängers immer ein Präfix von $data$ ist (*safety*), sowie daß jedes Datenelement des Senders irgendwann in der Ausgabe des Empfängers erscheint (*liveness*). Wenn man im Zusammenhang mit Selbststabilisierung Fehlereinflüsse mit einkalkuliert, schwächt sich die Korrektheitsbedingung etwas ab: In diesem Falle müssen *safety* und *liveness* zwar auch gelten, aber erst ab einem gewissen Zeitpunkt nachdem die Störeinflüsse aufgehört haben [2, S. 28]. Zusätzlich muß dann nur noch nachgewiesen werden, daß das Protokoll aus einem beliebigen Zustand in einen Zustand hineinkommt, "in dem es korrekt weitergeht." Afek und Brown schreiben hierzu:

> The last operational interval [in which there are no more faults aside from message loss] consists of two periods, a convergence period, and an infinitely long stable period, called the *final interval.* In the convergence period the protocol automatically moves the system from the arbitrary global state at which it began to a legal global state which guarantees it's correct operation thereafter [...]. When the system is subject to transient errors [...] it is required to behave correctly only during the final interval [...]" [2, S. 28]

Die Nachrichtenkanäle können also beliebige Fehler in das System injizieren: Nachrichten können verlorengehen, sich überholen oder verdoppeln. Wichtig ist nur, daß das System sich selbst in einen legalen Zustand bringt, wenn diese Fehler einmal aufhören. Vorausgesetzt wird Fairneß auf Prozessen und Kommunikationskanälen [2, S. 30].

4.11 Algorithmus (selbststabilisierendes *stop and wait*-Protokoll) Die lokalen Algorithmen von Sender und Empfänger sind in Abbildung 4.8, beziehungsweise Abbildung 4.9 wiedergegeben. Die Anweisung **timeout**, die als Wache in der ersten Anweisung des Senders auftaucht, ermöglicht ein zeitüberwachtes Senden. Die Wache wird wahr, sobald ein interner Zähler bezogen auf die lokale Uhr des Senders einen gewissen (festen) Wert (den *timeout*-Wert, oder einfach *timeout*) überschritten hat. Der Zähler wird zurückgesetzt, wenn eine der drei bewachten Anweisungen ausgeführt wird.

Die Variablen *ns* (für *number to send*) und *na* (für *number already acknowledged*) stehen für die Ordnungsnummern der Pakete, die als nächstes gesendet, beziehungsweise zuletzt empfangen worden sind. Ein normaler Ablauf beginnt damit, daß senderseitig der *timeout* abläuft und das erste Nachrichtenpaket *data*[1] zusammen mit der Sequenznummer 1 an den Empfänger gesendet wird. Dort angekommen wird die Nachricht entgegengenommen und an die Ausgabe weitergeleitet. Zudem wird durch eine zusätzliche Nachricht der Erhalt des Datenelementes bestätigt. Der Sender wird nach Erhalt der Bestätigung seinen eigenen Zähler *ns* inkrementieren und das nächste Datum über die Leitung senden und so weiter.

Um anzudeuten, daß das Kommunikationsprotokoll den geforderten Korrektheitsanforderungen entspricht, will ich lediglich ein kleines Beispiel präsentieren. Abbildung 4.10 visualisiert einen möglichen Ablauf des Protokolls. Der (beliebige) Startzustand ist durch die Werte $ns = 25$ und $na = 37$ charakterisiert. Der Sender schickt nach Abwarten der *timeout*-Zeit das Datenelement mit der Sequenznummer 25 über die Leitung. Der Empfänger erhält sie, setzt seine Variable *na* auf die soeben empfangene Nummer, liefert das Datenelement aus und schickt eine Bestätigung zurück. Sobald der Empfänger seinen eigenen Zähler auf den gesendeten Wert von *ns* setzt, ist ein legaler Zustand erreicht und das Protokoll arbeitet von nun an korrekt. Das gleiche passiert, wenn die Werte von *ns*

process *Sender*
  const $data[1\ldots]$ {* data to send *}
  var $ns \in \mathbb{N}$ **init** 1 {* # of next item to send *}
   $i \in \mathbb{N}$
  begin

$$
\begin{array}{lll}
 & \textbf{timeout} & \longrightarrow \quad \textbf{out}(ns, data[ns]) \\
[\!] & \textbf{in}(i) \wedge ns = i & \longrightarrow \quad ns := ns + 1;\ \textbf{out}(ns, data[ns]) \\
[\!] & \textbf{in}(i) \wedge ns \neq i & \longrightarrow \quad \textbf{skip}
\end{array}
$$

  end

Abbildung 4.8: Lokaler Algorithmus des Sende-Prozesses im selbststabilisierenden Kommunikationsprotokoll.

process *Receiver*
  var $na \in \mathbb{N}$ **init** 0 {* # of last acknowledged item *}
   $j \in \mathbb{N}$
   $d \in Data$
  begin

$$
\begin{array}{lll}
 & \textbf{in}(j, d) \wedge na \neq j & \longrightarrow \quad na := j;\ deliver(d);\ \textbf{out}(na) \\
[\!] & \textbf{in}(j, d) \wedge na = j & \longrightarrow \quad \textbf{out}(na)
\end{array}
$$

  end

Abbildung 4.9: Lokaler Algorithmus des Empfangsprozesses im selbststabilisierenden Kommunikationsprotokoll.

und na anfangs vertauscht gewesen wären.

Ein Sonderfall tritt ein, wenn beide Zählerwerte anfangs gleich sind. In diesem Falle würde es eine Hin- und eine Rücknachricht benötigen, um in einen legalen Zustand zu kommen, da erst die nächste Nachricht korrekt ausgeliefert werden würde. Im Maximalfall benötigt das Protokoll also jeweils zwei durchgekommene Nachrichten, um wieder in einen synchronisierten Zustand zu gelangen. Da durch die Fairneß-Anforderung Nachrichten auch irgendwann einmal durchkommen, solange sie nur oft genug gesendet werden, ist Selbststabilisierung garantiert.

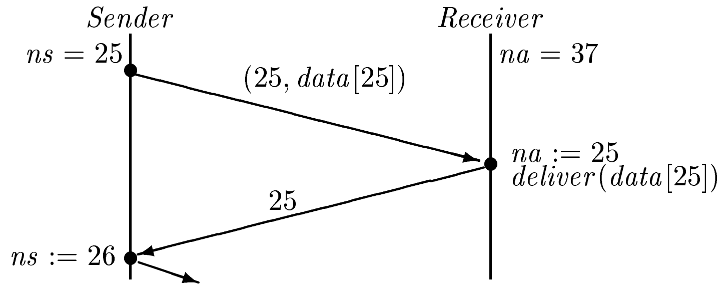

Abbildung 4.10: Beispielhafter Ablauf des selbststabilisierenden *stop and wait*-Protokolls.

Man kann das eben vorgestellte selbststabilisierende Kommunikationsprotokoll als Baustein für weitaus komplexere Protokolle verwenden (zum Beispiel für das *sliding window*-Protokoll [79, S. 452] [2, S. 28ff] oder für einen zuverlässigen *broadcast* [43, S. 39]). Dieser Ansatz eröffnet das Potential der Selbststabilisierung darum auch für die verbreiteten *message passing*-Modelle.[1]

Dijkstras Ring ohne gemeinsamen Speicher. Anknüpfend an das eben vorgestellte selbststabilisierende Kommunikationsprotokoll möchte ich nun demonstrieren, wie man Dijkstras Algorithmus zum wechselseitigen Ausschluß auf einem Ring auch in *message passing*-Umgebungen realisieren kann.

Die Idee dabei ist, das obige Kommunikationsprotokoll als einen degenerierten Ring mit nur zwei Prozessen anzusehen. Der Sender übernimmt dabei den Part des ausgezeichneten *bottom*-Prozesses. Man kann nun weitere Empfänger in das System hinzunehmen und sie ringförmig verschalten (wie in Abbildung 4.2 auf Seite 72). Die N Empfänger senden ihre Bestätigungen nun nicht mehr direkt an den Sender zurück sondern jeweils an den nächsten Empfänger, bis eine einmal initiierte Nachricht wieder beim Sender ankommt. Die lokalen Algorithmen von Sendepro-

[1] Beispielsweise benutzen Awerbuch, Patt-Shamir und Varghese genau dieses Protokoll, um einen real-implementierbaren Nachrichtenkanal zu konstruieren, dessen Kapazität auf genau eine Nachricht beschränkt ist (*unit capacity data link*=UCDL) [17, S. 271]. Ein UCDL kann dafür verwendet werden, um selbststabilisierende Ende-zu-Ende-Kommunikation in Weitverkehrsnetzen zu ermöglichen [17, S. 270].

zeß und den Empfangsprozessen sind in den Abbildungen 4.11 und 4.12 dargestellt.

process *Sender*
  var $ns \in \mathbb{N}$ **init** 1 {* # of next item to send *}
  $i \in \mathbb{N}$
  begin
    timeout $\longrightarrow$ **out**(ns)
  $\|$ **in**$(i) \wedge ns = i$ $\longrightarrow$ CS; $ns := ns + 1$; **out**(ns)
  $\|$ **in**$(i) \wedge ns \neq i$ $\longrightarrow$ **skip**
  end

Abbildung 4.11: Lokaler Algorithmus des Senders in Dijkstras Ring in *message passing*-Umgebungen.

process *Receiver*
  var $na \in \mathbb{N}$ **init** 0 {* # of last acknowledged item *}
  $j \in \mathbb{N}$
  begin
    in$(j) \wedge na \neq j$ $\longrightarrow$ CS; $na := j$; **out**(na)
  $\|$ **in**$(j) \wedge na = j$ $\longrightarrow$ **out**(na)
  end

Abbildung 4.12: Lokaler Algorithmus eines Empfängers in Dijkstras Ring in *message passing*-Umgebungen.

Die Bezeichnung *CS* steht wieder für den kritischen Abschnitt, in dem von dem exklusiven Zugriffsrecht Gebrauch gemacht werden kann. Es ist interessant, die Parallelen zu Dijkstras originalem Algorithmus zu beobachten: Erstens wird der Sender nur dann seinen kritischen Abschnitt betreten, wenn er eine Nachricht erhält, die die gleiche Sequenznummer hat wie er selbst. Die anderen Prozesse tun dies genau dann, wenn die erhaltene Sequenznummer ungleich der eigenen ist. Wenn dies geschieht, gleichen sie ihre eigene Variable *na* dem neuen Wert an. Der Beweis der Korrektheit kann sich diese Analogien zunutze machen, muß aber beachten, daß der Sender keine neuen Nachrichten schickt, solange er sich in seinem eigenen kritischen Abschnitt befindet. Auch gibt es Konfigurationen des Systems ohne ein *token*, jedoch wird ein neues *token* erzeugt, sobald das Kommando **timeout** des Sendeprozesses wahr wird. Die Originalversion dieses Protokolls und der komplette Korrektheitsbeweis kann man dem Originalartikel von Afek und Brown [2, S. 33] entnehmen. Eine weitere (unabhängige) Darstellung findet man im Artikel von Costello und Varghese [43, S. 36f]. Dort heißt die eingesetzte Methode *counter flushing*.

4.3 Das Potential der Selbststabilisierung

Nach der formalen Definition und einigen Beispielen für selbststabilisierende Algorithmen soll es in diesem Abschnitt um das Potential dieser Methode gehen. Zunächst wird diese Frage von der negativen Seite her beantwortet und betrachtet, welche Eigenschaften generell mit Selbststabilisierung unverträglich sind oder diese behindern. Anschließend wird der Begriff der Selbststabilisierung "aufgeweicht" und zwischen verschiedenen Graden dieser Eigenschaft differenziert. Das Zielpotential dieser Eigenschaft liegt auf dem Gebiet der Fehlertoleranz. Dies wird abschließend genauer untersucht und es wird dargestellt, zu welchen Problemen es bereits selbststabilisierende Lösungen gibt und zu welchen nicht.

4.3.1 Unmöglichkeitsresultate

Wenn man eine so elegante Eigenschaft wie Selbststabilisierung betrachtet, erwartet man geradezu, daß sich bestimmte Probleme unter bestimmten Voraussetzungen nicht mittels dieser Technik lösen lassen. In Fällen, in denen man dafür einen formalen Nachweis erbringen kann, spricht man von *Unmöglichkeitsresultaten* (*impossibility results*). Generell sind Unmöglichkeitsresultate sehr schwierig zu erreichen, da man eine Aussage über *alle möglichen Programme für ein bestimmtes Problem* machen muß. Das Potential der Selbststabilisierung ist aber nur einschätzbar, wenn man sich gleich mit den existierenden Resultaten auf diesem Gebiet vertraut macht. Darum will ich die existierenden Resultate hier kurz vorstellen — zwar ohne Beweis, aber mit einigen Erläuterungen, welche die besprochenen Resultate plausibler machen sollen.

Terminierung. In einer ausführlichen Untersuchung selbststabilisierender Kommunikationsprotokolle [79] haben Gouda und Multari drei wesentliche Unmöglichkeitsresultate aufgestellt. Das erste davon betrifft die Terminierung selbststabilisierender Algorithmen, die auf Nachrichtenaustausch in Netzwerken mit FIFO-Kanälen basieren. Zunächst muß jedoch definiert werden, was unter Terminierung zu verstehen ist.

4.12 Definition (ordnungsgemäße Terminierung) Eine Konfiguration c eines verteilten Algorithmus ist *ordnungsgemäß terminierend* (*properly terminating*), gdw. keine Nachrichten mehr unterwegs sind und die Werte der Variablen der einzelnen Prozesse von sich aus garantieren, daß keine keine bewachte Anweisung möglich ist. Ein verteilter Algorithmus ist ordnungsgemäß terminierend für ein Prädikat P, gdw. jeder Zustand aus P ordnungsgemäß terminierend ist.

Diese Definition der Terminierung mag etwas stark erscheinen. Normalerweise würde man lediglich fordern, daß keine Nachrichten mehr unterwegs sind und keine bewachte Anweisung möglich ist. Letzteres könnte beispielsweise auch eintreten, weil keine Nachricht für eine Wache der Form $\mathbf{in}(x)$ anliegt. Mit der stärkeren Definition 4.12 ist es jedoch für jeden Prozeß einzeln möglich, die Terminierung selbst zu erkennen.

4.13 Satz (Unmöglichkeit der ordnungsgemäßen Terminierung in *message passing*-Umgebungen [79, Theorem 2]) Ein Algorithmus, der sich selbst zu einem Prädikat P stabilisiert, kann nicht für P ordnungsgemäß terminieren.

Der formale Beweis des Satzes [79, S. 450] beruht auf der Tatsache, daß man aus jeder Konfiguration c, in der das System ordnungsgemäß für P terminierend ist, einen Zustand c' konstruieren kann, in dem alle lokalen Variablen die gleichen Werte besitzen, aber noch eine Nachricht unterwegs ist. Wegen dieser Nachricht kann c' selbst kein Zustand aus P sein; doch aus c' gibt es kein Entrinnen, da die Werte der lokalen Variablen ein Weiterschreiten des Algorithmus verhindern. Dies bedeutet, daß sich das System auch nicht zu P stabilisieren kann. Selbststabilisierende Algorithmen dürfen also nicht ordnungsgemäß terminieren.

Unendlich viele legale Zustände. Das zweite Resultat betrifft die Anzahl der Zustände, die sinnvolle verteilte Algorithmen haben müssen, um selbststabilisierend zu sein. Sinnvolle verteilte Algorithmen sind solche, die nie aufhören "zu arbeiten." Dies wird mit der Definition von Aktivität formalisiert.

4.14 Definition (aktive verteilte Algorithmen) Ein verteilter Algorithmus ist *aktiv* für ein Prädikat P (P-*live*), gdw. jede Ausführung des Algorithmus, die in einem Zustand aus P startet, unendlich ist, und in jeder solchen Ausführung mindestens ein Prozeß unendlich viele Nachrichten versendet und keine **timeout**-Aktion unendlich oft ausgeführt wird.

Die eben vorgestellte Definition soll erreichen, daß man triviale Lösungen ausschließt, in denen beispielsweise alle Zustände sicher sind oder im Endeffekt nichts wirklich getan wird. (Sie ist eine Spezialisierung des Gegenteils der ordnungsgemäßen Terminierung.) Der folgende Satz besagt, daß die Anzahl der legalen Zustände eines aktiven Algorithmus unendlich sein muß.

4.15 Satz (Anzahl der sicheren Zustände [79, Theorem 4]) Falls A ein verteilter Algorithmus ist, der sich selbst zu einem Prädikat P stabilisiert und für dasselbe Prädikat auch aktiv ist, dann ist die Anzahl der Zustände in P unendlich.

Grundlage dieses Satzes ist die Feststellung, daß ein einzelner Prozeß nie wissen kann, wieviele Nachrichten noch "da draußen" unterwegs sind. Für einen Algorithmus mit nur endlich vielen sicheren Zuständen in P kann man immer Konfigurationen konstruieren, die lokal wie welche aus P aussehen, aber trotzdem nicht zu P gehören. (Das liegt an der unbegrenzten Kapazität der Kanäle, über die Nachrichten ausgetauscht werden.) Ein Algorithmus, der sich selbst zu P stabilisieren soll, kann sich so gewissermaßen "aus Versehen" nicht zu P stabilisieren. Gouda nennt dies das Problem der *look-alike configurations* [78, S. 704].

Minimaler Befehlssatz. Das dritte Resultat von Gouda und Multari betrifft den Befehlssatz, der einem Entwickler selbststabilisierender Algorithmen zur Verfügung stehen muß. Es besagt, daß man neben bewachten Anweisungen und Sende- und Empfangskommandos mindestens auch noch die **timeout**-Anweisung benötigt. Das Ergebnis wurde zunächst nur für Algorithmen bewiesen, die zwar unendlich viele sichere Zustände haben, deren Nachrichtendurchsatz allerdings beschränkt ist.

4.16 Definition (Durchsatzbeschränkung auf Kanälen) Der Durchsatz eines Algorithmus ist *beschränkt* für ein Prädikat P (*is P-restrained*), gdw. in jedem Zustand von P und für jeden Kanal die Anzahl der Nachrichten in diesem Kanal kleiner als eine feste Zahl K ist.

Die Durchsatzbeschränkung ist eine Eigenschaft, die oft wünschenswert, aber doch eher nur von theoretischer Bedeutung ist. Notwendig wird sie für den folgenden Satz.

4.17 Satz (Notwendigkeit von **timeout**-Anweisungen [79, Theorem 6]) Ein Algorithmus, der sich selbst zu einem Prädikat P stabilisiert, für P aktiv und dessen Durchsatz für P beschränkt ist, enthält mindestens eine **timeout**-Anweisung in einem seiner lokalen Algorithmen.

Beim Beweis dieses Satzes nimmt man an, daß der Algorithmus keinerlei **timeout**-Anweisungen enthält und konstruiert einen Widerspruch: Man unterscheidet zwischen Wachen der Form $\langle guard \rangle$ (den sogenannten **out**-Wachen) und Wachen der Form $\langle guard \rangle \wedge \mathbf{in}(x)$ und betrachtet zwei Fälle:

1. In einem beliebigen Zustand ist mindestens eine **out**-Wache eines Prozesses p wahr. In diesem Fall könnte p andauernd **out**-Anweisungen ausführen und so die Anzahl der Nachrichten auf einem Kanal jede Schranke überschreiten lassen. Dies steht im Widerspruch zur Durchsatzbeschränkung des Algorithmus.

2. In einem beliebigen Zustand c ist keine **out**-Wache wahr. In diesem Fall konstruiert man sich daraus einen Zustand c', in dem alle Kanäle leer sind. Ohne aktivierte Anweisungen können aber keine Nachrichten verschickt werden, was im Widerspruch zur Aktivitätsforderung des Algorithmus steht.

Der zweite Fall reflektiert einen Zustand, in dem sich das System in einer "Ruhelage" befindet. Hier benötigt man eine Zeitüberwachung, um den Algorithmus am Leben zu erhalten. Dies ist beispielsweise der Fall, wenn der initiale Zustand einem Prozeß suggeriert, daß noch die Antwort auf eine Nachricht aussteht, die er nie abgeschickt hat [126, S. 63]. Eine umgangssprachliche Erklärung des ersten Falles ist mir nicht ohne weiteres möglich. Hier wirkt die Durchsatzbeschränkung des Algorithmus, durch die der Beweis erst möglich wird, etwas künstlich — ein Indiz dafür, daß die Definition nur dem Beweis dient.

Symmetrie. Systeme, in denen alle Prozesse identisch sind, nennt man *symmetrisch* [136, S. 305]. Symmetrie ist nicht nur in Verbindung mit Selbststabilisierung ein Problem in verteilten Systemen: Es gibt viele Unmöglichkeitsresultate auf diesem Gebiet und auch eine ganze Reihe von Algorithmen, die (in probabilistischer Art und Weise) den einzelnen Prozessen nachträglich eine eindeutige Identität verleihen [136, Kapitel 9]. Dies wird als das *Brechen der Symmetrie* bezeichnet.

Ein Resultat, welches indirekt aus Dijkstras Originalartikel [49] folgt, ist die Unmöglichkeit einer selbststabilisierenden symmetrischen Lösung des wechselseitigen Ausschlusses auf einem beliebigen Ring von Prozessoren [126, S. 52]. Burns und Pachl [32] zeigen, daß es eine symmetrische Lösung nur dann gibt, wenn die Anzahl der Prozesse im Ring eine Primzahl ist. Generell muß es in Systemen, deren Prozesse kooperieren und sich gegenseitig synchronisieren immer eine Form der Symmetriebrechung geben. Schneider teilt diese Formen ein in zwei Klassen:

1. Symmetriebrechung durch Zustand (*asymmetric by state*), wobei die einzelnen Prozesse in garantiert verschiedenen initialen Zuständen die Berechnung beginnen, und
2. Symmetriebrechung durch Identität (*asymmetric by identity*), wobei entweder gleiche Programme durch eine lokale (nicht-flüchtige) Identität parametrisiert sind, oder unterschiedliche Algorithmen in den Prozessen ablaufen.

Eine wichtige Feststellung ist, daß die Symmetriebrechung durch Zustand keine praktikable Möglichkeit für selbststabilisierende Algorithmen darstellt, da sich das System aus einem beliebigen, also auch symmetrischen, Zustand wieder "aufrappeln" können muß. Zur Brechung der Symmetrie muß also letztere Form gewählt werden, was allerdings keine eigentliche Einschränkung in der Praxis ist.

Kommunikationsanforderungen. Um Selbststabilisierung zu garantieren, muß man gewisse minimale Kommunikationsmöglichkeiten voraussetzen. Dies soll an einem einfachen Beispiel verdeutlicht werden.

4.18 Beispiel (Isolation [126, S. 64]) Ein asynchrones *message passing*-System besteht aus drei Prozessen, deren Kommunikationsgraph ein Baum ist. Ein Prozeß fungiert als Wurzel und hat jeweils einen unidirektionalen Nachrichtenkanal zu den beiden anderen Knoten, die weder mit der Wurzel noch untereinander Nachrichten austauschen können. Jeder Prozeß hat einen lokalen Zähler. Die Wurzel kann diesen Zähler spontan inkrementieren und schickt anschließend eine Nachricht zu einem ihrer Kind-Knoten, der bei Erhalt dieser Nachricht ebenfalls seinen lokalen Zähler hochzählt. Ein Zustand wird als "sicher" definiert, wenn der Stand des Zählers in der Wurzel gleich der Summe der Zähler in den Kindknoten plus der Anzahl der Nachrichten ist, die noch unterwegs sind. Ist das System ein Mal in einem illegalen Zustand, so besteht keinerlei Möglichkeit, wieder daraus zu entkommen, da die Knoten nicht genügend Möglichkeiten haben, Nachrichten auszutauschen.

Das Szenario, welches in dem Beispiel aufgezeigt wird, nennt man Isolation (*isolation* [78, S. 704]) und gehört zu den noch wenig untersuchten Problemen im Zusammenhang mit Selbststabilisierung. Man geht jedoch davon aus, daß ein stark zusammenhängender Kommunikationsgraph in der Regel ausreichend ist. (Ein gerichteter Graph ist stark zusammenhängend, wenn es einen Pfad von jedem Knoten zu jedem anderen Knoten gibt.)

Instabilität bei Simulation. In ihrem Artikel "The Instability of Self-Stabilization" [78] weisen Gouda, Howell und Rosier darauf hin, daß die Eigenschaft der Selbststabilisierung sehr fragil ist, wenn es darum geht, Algorithmen in verschiedenen Maschinenmodellen gegenseitig zu simulieren. Konkret betrachten sie eine große Bandbreite von Modellen (von zellularen Automaten über Turingmaschinen bis hin zu Petri-Netzen) und zeigen, daß es für eine große Anzahl von Maschinenmodellen keine Simulation durch ein anderes Modell gibt, welche die Selbststabilisierungseigenschaft erhält oder gar erzwingt.

Auswirkungen der Resultate. Unmöglichkeitsresultaten haftet immer ein Hauch der Praxisferne an: Für die Praxis bringen sie meist nur unwesentliche Einschränkungen mit sich. Das liegt daran, daß alle vorgestellten Sätze in direkter Weise von ihren Voraussetzungen abhängen, insbesondere beispielsweise von den starken Definitionen für die ordnungsgemäße Terminierung oder Aktivität. Schwächt man die Voraussetzungen in kleinen Teilen ab, werden vorher unmögliche Probleme plötzlich möglich.

Das Paradebeispiel ist die Forderung nach unendlich vielen legalen Zuständen (Satz 4.15): Unendlich viele legale Zustände sind in der Regel nur durch unbeschränkte Zähler innerhalb der Prozesse zu realisieren, ein Mittel, welches es in der Praxis nicht gibt. Zur Lösung dieses Problems benutzen Afek und Brown in ihrer Version des selbststabilisierenden *sliding window*-Protokolls [2] ein "Orakel," welches ein aperiodisches Bitmuster (zum Beispiel die Binärziffern einer irrationalen Zahl) erzeugt. Somit benötigt man zwar keine unbeschränkten Zähler mehr (und kann damit auch Nachrichten endlicher Größe versenden), doch benötigt man damit theoretisch einen Prozeßautomaten mit unendlich vielen Zuständen. Verzichtet man jedoch auf die Voraussetzung, daß die Kanäle unendliche Kapazität haben können oder Nachrichten beliebig lange unterwegs sein können (was vor allem in lokalen Netzwerken realistisch ist), werden plötzlich Realisierungen mit beschränkten Zählern möglich. Beispiele sind die

Realisierungen von Costello und Varghese [43] und Spinelli [131].

Ebenso verhält es sich mit der Terminierung: Ordnungsgemäße Terminierung ist eine Forderung, die in verteilten Systemen weniger wichtig ist als etwa in Einprozessorsystemen. Dies liegt daran, daß Lösungen verteilter Algorithmen eher qualitativer als quantitativer Natur sind [126, S. 63]. Wo sequentielle Algorithmen ein Ergebnis berechnen und dann stoppen, geht es verteilten Algorithmen eher darum, insgesamt eine Fortschritt wie wechselseitigen Ausschluß zu erzielen. Das Unmöglichkeitsresultat schließt beispielsweise nicht aus, daß sich selbststabilisierende Algorithmen "zur Ruhe begeben," um dann, wenn es nötig ist, wieder aus der Versenkung herauszukommen und aktiv zu werden (wie beispielsweise der Algorithmus zur verteilten Spannbaumkonstruktion aus Abschnitt 4.2.5).

Die Auswirkungen der Resultate, die die Simulation einer Modellklasse durch eine andere betreffen, haben offensichtlich Auswirkungen auf den Bau von Compilern: Wenn man einen selbststabilisierenden Algorithmus in einer formalen Notation wie den bewachten Anweisungen entworfen hat, möchte man natürlich, daß ein Compiler dafür ausführbaren Maschinencode herstellt, der ebenfalls die Selbststabilisierungseigenschaft hat. Noch besser wäre es natürlich, wenn ein Compiler automatische diese Eigenschaft dem entworfenen Algorithmus beigeben würde. Die Resultate von Gouda *et al.* [78] zeigen, daß man hierbei sehr vorsichtig vorgehen muß. Auf diese und verwandte Fragestellungen werde ich in Abschnitt 4.4 über die Methodologie der Selbststabilisierung nochmals zu sprechen kommen.

4.3.2 Selbststabilisierung und Fehlertoleranz

In der Informatik ist das Gebiet der Fehlertoleranz immer eine Spielwiese für Genies und scheinbare Magier gewesen. Es gibt zahllose Definitionen und verschiedene englischsprachige Varianten des Begriffes "Fehler" (vergleiche dazu Fußnote 5 auf Seite 28), sowie unterschiedliche Verständnisse der einzelnen Fehlerklassen, die in Abschnitt 2.3.7 bereits abgesprochen wurden. Außerdem hat man sich auf dem Gebiet der Fehlertoleranz bisher sehr stark mit der Betrachtung und Behandlung einzelner Fehlersymptome und -ursachen beschäftigt; eine scheinbare Zerfaserung des Gebietes war die Folge [9, S. 1915]. Im folgenden möchte ich eine formale Definition von Fehlertoleranz vorstellen, die auf Selbststabilisierung beruht. Durch die anschließende Diskussion wird es einfacher möglich sein, die Begriffe Selbststabilisierung und Fehlertoleranz gegeneinander abzugrenzen.

Selbststabilisierung als Teil von Fehlertoleranz. Eine einheitliche Definition von Fehlertoleranz geht auf die Dissertation von Anish Arora [14] zurück, die in Form eines Artikels in den *IEEE Transactions on Software Engineering* 1993 in einer abgespeckten Version als Artikel erschien [9]. Sie beruht auf der Modellierung von Fehlern als unerwünschte aber durchaus mögliche Aktionen auf einem erweiterten Zustandsraum (vergleiche Abschnitt 2.3.7).

4.19 Definition (Fehlertoleranz [9, S. 1017]) Sei F eine Menge von Fehleraktionen und P ein abgeschlossenes Prädikat. Ein System ist F-fehlertolerant für P, gdw. es ein Prädikat T auf Konfigurationen gibt, für das gilt:

- T gilt, wann immer P gilt.
- T ist abgeschlossen gegenüber allen Aktionen, insbesondere denen aus F.
- Das System stabilisiert sich selbst zu P.

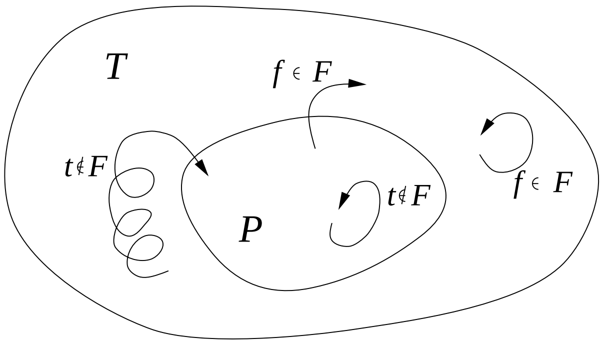

Abbildung 4.13: Schematische Visualisierung von Definition 4.19.

Die Bedeutung dieser Definition kann man sich anhand von Abbildung 4.13 klarmachen, in denen T und P durch Mengen visualisiert sind: Das Prädikat T ist eine Art Verallgemeinerung des Legalitätsprädikates P und umfaßt die Fehlerzustände, in die das System schlimmstenfalls geraten kann. (Arora nennt T darum auch oft die Menge der *tolerierten Fehlerzustände* (*fault span*) [13, S. 179].) Normalerweise, also solange keine Fehleraktionen aus F auftreten, wird sich das System immer innerhalb von P aufhalten. Im Falle eines Fehlers kann es allerdings P verlassen und in eine Art illegalen Zustand geraten, doch diese Zustände sind immer innerhalb von T. Solange jetzt weitere Fehleraktionen auftreten, wird das System wegen der Abgeschlossenheit von T auch in T verbleiben. Doch hören die Störungen irgendwann auf, wird es nach einer gewissen Zeit wieder wegen seiner Selbststabilisierungseigenschaft nach P zurückkehren und wieder ganz normal weiterarbeiten.

Wenn das Prädikat T alle möglichen Systemkonfigurationen umfaßt (als $T \equiv$ *true*), dann toleriert das System *jeden* Fehler, den man durch Fehleraktionen modellieren kann. In der Regel ist dies jedoch eine zu starke Forderung. Wenn man nur eine begrenzte Anzahl von Fehlern in die Fehlermenge F aufnimmt, dann kann T durchaus auch anders aussehen.

Praktikabilität der Definition. Trotz ihrer theoretischen Eleganz hat diese Definition einen schwerwiegenden Nachteil. Sie geht davon aus, daß die Fehleraktionen allmählich aufhören aufzutreten, daß es sich also um vorübergehende Fehler (*transient failures*) handelt.[2] Wie ist es also um die komplementäre Fehlerklasse, die der permanenten Fehler (*permanent failures*) bestellt?

In einem ersten Schritt könnte man argumentieren, daß jeder Fehler eigentlich nur vorübergehender Natur ist, da man jeden Fehler irgendwie reparieren kann. Hierauf gibt es jedoch zwei berechtigte Einwände: Erstens kann es in der Praxis vorkommen, daß hartnäckige Fehler, wie ein Systemabsturz, möglicherweise erst nach ein oder zwei Tagen repariert werden. Innerhalb dieser Zeit kann sich das Gesamtsystem nicht selbst stabilisieren und darum auch kein korrektes Verhalten zeigen. Noch viel schwerwiegender ist allerdings der zweite Einwand: Fehler in Raumsonden [122], Raumstationen [120, 130] oder in Satelliten können nicht oder nur unter nicht vertretbarem Aufwand repariert werden. In derartigen Fällen ist es aber unbedingt notwendig, daß das Gesamtsystem auch weiterhin seine Systemspezifikation erfüllt, und sei es nur mit einer verminderten Leistung. Manche Autoren betrachten Selbststabilisierung und Fehlertoleranz aus diesem Grunde als zwei verschiedene "Paradigmen" [5, S. 174]. Anagnostou und Hadzilacos schreiben dazu:

> "In self-stabilization one takes the view that failures are *transient*, and can potentially affect *all* the processors of the system. [. . .] In the second paradigm, fault-tolerance, one takes the view that failures are *permanent* and that they affect only some *fraction* of the total number of processors." [5, S. 174f]

Diese Unterscheidung wird gestützt durch zwei Unmöglichkeitsresultate, die sie im gleichen Artikel [5] vorstellen. Sie zeigen dort, daß die Widerstandsfähigkeit gegenüber permanenten und vorübergehenden Fehlern in asynchronen Systemen zwei konkurrierende Eigenschaften sind. Zwar gibt es zu Problemen wie *leader election* oder zur Spannbaumkonstruktion Algorithmen, die entweder vorübergehende oder permanente Fehler tolerieren, aber es gibt keine, die beides zugleich können [5, S. 176]. Der Grund hierfür liegt darin, daß es in asynchronen Systemen nicht möglich ist, zwischen fehlerhaften oder nur sehr langsamen Prozessen zu unterscheiden. Ein Beispiel ist das Zählen von Knoten in einem Ring der Größe n: Ein Algorithmus wird sich im ersten Fall zu einem Zustand stabilisieren, in dem alle Prozesse die Zahl n kennen; im zweiten Fall, in dem ein Prozessor dauerhaft ausgefallen ist, werden alle Prozesse am Ende die Zahl $n-1$ kennen. Da es aus dem System heraus unmöglich ist, diese beiden Fälle zu unterscheiden, kann sich das System auch nicht selbst stabilisieren.

Eine ähnliche Unterscheidung der Fehlerklassen wie Anagnostou und Hadzilacos machen Gopal und Perry in einer zweiten Untersuchung zu

[2] Gopal und Perry bezeichnen sie deshalb als eine "triviale" Definition [74, S. 196].

diesem Thema [74]. Begrifflich manifestiert sich dies in den Worten *systemic* und *process failures*. Die erste Klasse enthält alle Fehler, die nur den lokalen Speicher eines Prozesses korrumpieren und das Programm selbst nicht verändern. Dies sind somit genau die Fehler, die man auf dem Gebiet der Selbststabilisierung von Anfang an betrachtete. Die zweite Fehlermenge, die Prozeß-Fehler, enthalten alle möglichen Verhaltensweisen, die vom intendierten Verhalten des Prozesses abweichen. Hierzu gehören die bekannten Fehlerklassen des *process crash* oder wahlloses, "byzantinisches" Verhalten [74, S. 195].[3] Auch sie betonen, daß der fundamentale Unterschied zwischen den beiden Fehlerklassen die Anzahl der Prozesse ist, die jeweils betroffen werden dürfen.[4] Sie betrachten im wesentlichen synchrone Systeme und stoßen also nicht auf die durch Anagnostou und Hadzilacos beschriebenen Probleme; allerdings bemerken sie, daß für asynchrone Systeme neue Erkenntnisse [33] in Bezug auf die Voraussetzungen von wichtigen Unmöglichkeitsresultaten [56] bei weiteren Untersuchungen in Betracht gezogen werden müssen.

Maskierende vs. nicht-maskierende Fehlertoleranz. Orthogonal zur Unterscheidung zwischen permanenten und vorübergehenden Fehlern gibt es noch ein weiteres Begriffspaar, welches im Zusammenhang mit Selbststabilisierung und Fehlertoleranz genannt werden muß: maskierende und nichtmaskierende Fehlertoleranz.

Maskierende Fehlertoleranz (*masking fault-tolerance*) bezeichnet die Eigenschaft, das Auftreten von Fehlern innerhalb des Systems abzufangen und ihre Auswirkungen nicht nach außen dringen zu lassen. Formal bedeutet dies, daß sich Systeme immer und zu jeder Zeit gemäß ihrer Spezifikation verhalten. Fehler werden so hinter einer "Maske" versteckt. Auf der anderen Seite liegt nichmaskierende Fehlertoleranz (*non-masking fault-tolerance*), wenn die Systemspezifikation zwar verletzt werden kann, aber garantiert ist, daß der vorliegende und nach außen hin sichtbare Fehler nur vorübergehend ist, sich das System also nach begrenzter Zeit wieder "richtig" verhält [12, S. 424]. Nicht-maskierende Fehlertoleranz hat immer dann ihre Berechtigung, wenn maskierende Fehlertoleranz unmöglich [56], nur zu einem unverhältnismäßig hohen Preis, oder einfach unnotwendigerweise zu erreichen ist. Als Beispiel für letzteres führen Arora und Gouda einen telefonischen Rückrufservice an, bei dem das Herstellen einer Verbindung auch nicht auf Anhieb klappen muß [10, S. 1032]. Natürlich muß die Zeit, in der die Spezifikation verletzt bleibt, in akzeptablen Grenzen

[3] Als einer der ersten untersuchte Leslie Lamport dieses Gebiet. Von ihm stammt eine weitere, dritte Zweiteilung der Fehlermenge in *unannounced death* und *malfunctioning* (also ein *memory crash*) [103, S. 332]. Die Uneinheitlichkeit der Einteilungen sind ein weiteres Indiz für die eingangs angesprochene Zerfaserung des Gebietes.

[4] "Although both types of failure may result in corrupted process states, much previous work has established that tolerating malicious process failures requires that at most a fraction (typically, one-third) of the processes in the system be able to fail. In contrast, the object of tolerating systemic failures is to be resilient to the corruption of the state of *every* process in the system." [74, S. 196f]

liegen.

Tel unterscheidet auf dieser Ebene zwei verschiedene Herangehensweisen. Die erste nennt er *optimistisch* (*optimistic*) in dem Sinne, daß Prozesse zwar zeitweise inkonsistent zu ihrer Spezifikation arbeiten, daß sie aber nach einer gewissen Zeit wieder beginnen normal zu arbeiten [136, S. 459]. Der pessimistische Ansatz besagt, daß Prozesse prinzipiell jeder Information im System mißtrauen sollten und daß sie darum jeden ihrer Schritte sehr vorsichtig (in Abstimmung mit allen anderen Prozessen) tun sollten, um ihre Spezifikation nicht zu verletzen. Tel nennt Algorithmen mit letzterem Ansatz *robust* und ordnet die Selbststabilisierung in den Rahmen des ersten, optimistischen Ansatzes ein [136, S. 459]. Doch Selbststabilisierung muß nicht bedeuten, daß man automatisch nicht-maskierende Fehlertoleranz erhält.

In ihrem Artikel "Designing masking fault-tolerance via nonmasking fault-tolerance" [13] stellen Arora und Kulkarni eine Methode vor, wie man mittels Selbststabilisierung ein fehler-intolerantes Programm widerstandsfähig gegenüber vorübergehenden Fehlern machen kann, ohne daß es jemals dabei seine Spezifikation verletzt. Sie definieren maskierende Fehlertoleranz gegenüber einer Menge F von Fehleraktionen dadurch, daß in Definition 4.19 auf Seite 95 die Menge T der tolerierten Fehlerzustände gleich der Menge P der legalen Zustände ist. Dadurch ist jeder Fehlerzustand automatisch ein "sicherer" Zustand.

Ihre Transformation läuft im wesentlichen in zwei Phasen ab:

- In der ersten Phase wird dem ursprünglichen Programm durch Hinzufügung von sogenannten *Korrektoren* (*correctors*) nicht-maskierende Fehlertoleranz hinzugefügt.

- In der zweiten Phase wird aus nicht-maskierender Fehlertoleranz plötzlich maskierende Fehlertoleranz, indem alle kritischen Aktionen, die die Spezifikation nach außen hin verletzen könnten, nicht ausgeführt werden bis wieder ein sicherer Zustand erreicht ist. Die hinzugefügten Programmkomponenten, die dies erreichen, nennen sie *Detektoren* (*detectors*).

Als Korrektoren kommen viele bestehende Methoden in Frage, beispielsweise Rücksetzmechanismen, fehlererkennende Codes oder Ausnahmebehandlungsmechanismen. Mit ihnen wird erreicht, daß ein Programm auch in Fehlersituationen die *liveness* bezüglich seiner ursprünglichen Spezifikation behält. Die dazugehörige *safety* wird durch die Detektoren garantiert, die man durch Schnappschußalgorithmen, Konsistenzprüfer oder fehlerentdeckende Codes realisieren kann [13, S. 181]. Detektoren und Korrektoren müssen selbst maskierende Fehlertoleranz besitzen.

Diese Konstruktionsmethode kann man als Abstraktion des bisherigen Ansatzes zur Fehlertoleranz sehen, die mit Ausnahmebehandlungsmechanismen, Fehlererkennungsmechanismen und Rücksetzpunkten arbeiten.

Eine weitergehende Behandlung dieser Frage würde jedoch den Rahmen dieser Arbeit sprengen.

Beobachtung der Selbststabilisierung. Im Verfahren zur Herstellung maskierender Fehlertoleranz gehen die Autoren davon aus, daß die Detektoren eine Verletzung der *safety* des ursprünglichen Algorithmus erkennen können. Zu diesem Thema schreiben Katz und Perry:

> "Another interesting limitation regarding self-stabilization ist that, in general, no process can ever *know* [...] that the system is in a legal state even though the eventuality of a legal state is guaranteed." [94, S. 20]

Analog schreibt Tel:

> "It is not possible to observe from within the system that a legitimate configuration has been reached; hence the processes are never aware of when their behavior has become reliable." [136, S. 461]

Der Begriff des "Wissens" in verteilten Systemen ist sehr heikel und bedarf darum einer Definition. Katz und Perry verweisen hier auf Halpern und Fargin, die in ihrem Artikel "Modelling knowledge and action in distributed systems" [81] den Wissensbegriff für einen einzelnen Prozeß formal fassen. Demnach beruht das Wissen eines Prozesses allein auf seinem lokalen Zustand.

4.20 Definition (Wissen eines Prozesses [81, S. 160]) Sei c_p der lokale Zustand eines Prozesses p in einem System S. Betrachtet man den globalen Zustand c von S, so gibt es viele andere Konfigurationen c', die c_p nicht widersprechen. Ein Prozeß *weiß* eine Tatsache t zu einem bestimmten Zeitpunkt, wenn t wahr ist in allen Konfigurationen c', in denen der lokale Zustand von p gleich c_p ist.

Die Tatsache, die ein Prozeß im Zusammenhang dieser Arbeit "wissen" soll, ist das Erreichen eines sicheren Zustandes. Um dies zu wissen, muß der lokale Zustand des Prozesses unmißverständlich auf das Vorliegen des legalen Zustandes hindeuten. Doch hier ergeben sich Probleme. Katz und Perry schreiben:

> "Suppose there were some local state of process i that indicated a legal global state. Then, except for trivial cases, there are many illegal initial global states that are consistent with this local state of i, contradicting the validity of i's indicator." [94, S. 20]

Man benötigt also eine andere Definition von "Wissen," damit Algorithmen feststellen können, ob das System in einem legalen Zustand ist oder nicht. Mögliche Definitionen verwenden Lin und Simon [106] sowie Arora und Kulkarni [13]. In ihrem Verständnis genügt es, daß ein Prozeß nach einer gewissen Zeit einen legalen Systemzustand "beobachtet," also genügend Informationen lokal zusammengetragen hat und diese auf eine legale Konfiguration hindeutet. Diese Definition scheint praktikabel genug, um später im Abschnitt 4.4 über die Methodologie damit arbeiten zu können. In Zusammenhang mit der ursprünglichen Definition des Wissens und der Feststellung des "Nicht-Wissen-Könnens" gibt es aber eine sehr wichtige Konsequenz: Da Beobachtungen immer trügerisch sein können, dürfen selbststabilisierende Algorithmen ihr "Wissen" über einen legalen Systemzustand nicht dazu benutzen, um in einen anderen, möglicherweise effizienteren Modus zu schalten, in dem die Selbststabilisierungseigenschaft abgeschaltet ist [94, S. 20].

4.3.3 Existierende Algorithmen

Im Laufe der Zeit hat sich eine verblüffende Fülle selbststabilisierender Algorithmen in der Forschungsliteratur angehäuft, die nur noch schwer zu überschauen ist. Schneider [126] sowie Flatebo, Datta und Ghosh [58] haben verdienstvolle Übersichtsartikel veröffentlicht, die den Stand bis etwa in das Jahr 1993 abdecken. Seitdem sind zahlreiche Algorithmen hinzugekommen, unter anderem für Tiefensuche [41], Ringe mit mehr als einem *token* [59], das Finden kürzester Wege in Graphen [29, 138], Uhrensynchronisation [34, 54] und weitere Algorithmen zur Konstruktion aufspannender Bäume [6, 70, 92]. Die wesentliche Feststellung für diese Arbeit ist, daß nach den mir zur Verfügung stehenden Informationen schwachkonsistente Replikation noch nicht explizit versucht worden ist.

4.3.4 Die Kosten der Selbststabilisierung

Nach Mohemed Gouda ist das gesamte Gebiet der Komplexität der Selbststabilisierung noch größtenteils unerforscht [77, S. 15]. Eine erste Untersuchung von Chang, Gonnet und Rotem [36], die Dijkstras ursprüngliche Algorithmen [49] analysieren, kommt zu dem Schluß, daß:

> "Self-stabilization in distributed systems appears to be expensive." [36, S. 316]

Die Kosten können sich in ganz unterschiedlichen Einheiten manifestieren. Ein wesentliches Merkmal selbststabilisierender Algorithmen ist die Zeit, die sie maximal benötigen, um sich selbst in einen legalen Zustand zu bringen. Dies wird oft mit *Konvergenzzeit* bezeichnet. Auf die dahinterliegenden Zusammenhänge gehen Gouda und Evangelist [76] ein. Ghosch, Gupta, Herman und Pemmaraju [69] untersuchen beispielsweise, wie stark sich unterschiedliche Störungen auf die Konvergenzzeit einzelner Algorithmen auswirken. Die meisten Veröffentlichungen belassen es jedoch mit oberflächlichen Analysen der vorgestellten Algorithmen. Ich will dies hier aus Zeit- und Platzgründen genauso handhaben.

4.4 Versuch einer Methodologie der Selbststabilisierung

In den vergangenen Abschnitten wurde deutlich, daß Selbststabilisierung oft eine wünschenswerte Eigenschaft ist: Selbststabilisierende Algorithmen können verschiedenartige Fehler in einheitlicher Weise tolerieren, solange diese Fehler irgendwann repariert werden. Außerdem birgt diese Algorithmenklasse das Potential in sich, ein Paradigma auch für maskierende Fehlertoleranz zu werden. Allerdings genügt es nicht, diese Eigenschaft bloß zu fordern: Man muß sie beim Algorithmenentwurf berücksichtigen und anschließend nachweisen, damit sie (gerade in sicherheitskritischen Bereichen) von Nutzen ist.

In diesem Abschnitt geht es darum, eine Methodologie der Selbststabilisierung zu beschreiben. Dahinter steckt eine Ansammlung möglichst praktischer Methoden mit den Zielen,

- neue selbststabilisierende Algorithmen für eine bestimmte Aufgabe zu entwerfen, oder
- bekannte (nicht-selbststabilisierende) Algorithmen in möglichst mechanischer Art und Weise um die Selbstabilisierungseigenschaft zu erweitern.

Zu erwarten ist, daß eine so "gute" Eigenschaft wie die der Selbststabilisierung nicht "kostenlos" zu haben ist. Die vielen bereits veröffentlichten Algorithmen auf diesem Gebiet zeigen, daß ein paar bekannte Problemstellungen in verteilten Systemen besser zur Selbststabilisierung passen als andere. Es ist darum zu vermuten, daß sich Algorithmen zu deren Lösung leichter mit Selbststabilisierung ausstatten lassen und daß sich diese auch effizienter selbst stabilisieren als andere. Ich werde darum diesen Abschnitt beginnen mit allgemeinen Methoden, mit denen man praktisch jeden Algorithmus selbststabilisierend machen kann, und dann immer spezieller werden. Zum Abschluß dieses Kapitels werde ich dann diese unterschiedlichen Herangehendsweisen gegenüberstellen und bewerten.

4.4.1 Algorithmenkomposition

Viele Probleme, die in der Praxis mittels verteilter Algorithmen gelöst werden sollen, sind so komplex, daß es ungeheuer schwierig ist, Lösungsansätze "aus einem Guß" direkt und konkret zu entwickeln. In solchen

Fällen behilft man sich oft mit dem aus der sequentiellen Programmierung bekannten Mittel der schrittweisen Verfeinerung [141]: Das "große" Problem wird in mehrere Teilprobleme zerlegt, die selbst wieder so lange zerlegt werden, bis ein einfacher Algorithmus zu deren Lösung angegeben werden kann. Anschließend werden die so erstellten "kleinen" Algorithmen zusammengesetzt und durch die so definierte Hintereinanderausführung entsteht im Zusammenspiel ein Algorithmus zur Lösung des "großen" Problems. Eine solche Zusammensetzung wird oft *Algorithmen-Komposition* (*algorithm/protocol composition*) genannt.

Algorithmen-Kompositionen laufen naturgemäß in einzelnen "Phasen" ab, die aufeinander aufbauen. Um eine geforderte Nachbedingung Q zu erreichen, benutzt man einen ersten Algorithmus, der aus den gegebenen Eingabedaten eine "Zwischenbedingung" P herstellt. Anschließend startet man einen zweiten Algorithmus, der als Vorbedingung P hat und aus P die Nachbedingung Q herstellt. Beispiele für derartige Algorithmenkompositionen sind beispielsweise die Wegewahl in großen Netzwerken (*routing*) oder die Wahl eines "Chefs" in Ringen von Prozessoren (*leader election*) [136, S. 474]:

- Für die effiziente Wegewahl in Netzwerken benötigt man in der Regel sogenannte *routing-Tabellen* in den einzelnen Vermittlungsknoten. Die erste Phase des Algorithmus besteht also darin, in den einzelnen Knoten diese Tabellen aufzubauen. Sobald dies bis zu einem gewissen Grade geschehen ist, können die einzelnen Datenpakete im Netzwerk zwischen beliebigen Rechnern effizient vermittelt werden.

- In Netzwerken mit Ring-Topologie benötigt man oft einen sogenannten *Ring-Monitor*, also eine ausgezeichnete Station, die gewisse Aufgaben zentral verwaltet. Die Wahl eines solchen Monitors wird mittels Algorithmen für *leader election* geleistet. Sie bildet gewissermaßen also nur eine erste Vorstufe für die anschließende Ausführung eines zentralisierten Algorithmus auf dem Ring.

Ein kritischer Punkt bei der Komposition verteilter Algorithmen ist die Übergabe des Kontrollflusses an die nächste Phase. Normalerweise geschieht dies durch die überlagerte Ausführung eines Algorithmus zur Erkennung der verteilten Terminierung (*termination detection*). Wie im vorhergehenden Abschnitt 4.3.1 gezeigt wurde, ist dies im Falle selbststabilisierender Algorithmen nicht so einfach möglich, da sie nicht (ordnungsgemäß) terminieren dürfen. Ist es darum überhaupt möglich, einzelne selbststabilisierende Algorithmen zu einem größeren Ganzen zusammenzusetzen?

Glücklicherweise ist die Lage nicht aussichtslos, sondern viel einfacher als im herkömmlichen Fall: Man läßt die einzelnen selbststabilisierenden Teilalgorithmen einfach alle zur gleichen Zeit "loslaufen." Im traditionellen Fall würde dies zu einem heillosen Durcheinander führen: Ein nachgeschalteter Algorithmus darf erst gestartet werden, wenn seine Vorbedingung P auch tatsächlich gilt. Gilt P noch nicht, wird er vermutlich fehlerhaft ablaufen. Sind die Algorithmen jedoch alle selbststabilisierend, wird der erste Algorithmus irgendwann (aber garantiert) seine Nachbedingung P erfüllen. Ab diesem Zeitpunkt wird auch der zweite Algorithmus, der für seinen korrekten Ablauf die Vorbedingung P benötigt und für den die "Fehlerquelle" $\neg P$ nun praktisch ausgeschaltet ist, sich selbst auf seine eigene Nachbedingung Q stabilisieren und so weiter.

Damit dies so einfach funktioniert, müssen bestimmte Voraussetzungen erfüllt sein. Diese wurden unabhängig voneinander durch Dolev, Israeli und Moran 1993 [53, S. 15] und Herman [85] beschrieben. Dolev *et al.* nennen ihr Konzept *fair protocol combination*, Herman *collateral-composition rule*. (Die Informationen über Hermans Dissertation [85], die mir leider nicht vorlag, entstammen dem Buch von Tel [136, S. 475f].) Die Ergebnisse werden durch die folgende Definition und den darauffolgenden Satz zusammengefaßt.

4.21 Definition (Komposition von Algorithmen [136, S. 475][53, S. 15]) Seien $A_1 = (C_1, \delta_1)$ und $A_2 = (C_2, \delta_2)$ zwei selbststabilisierende Algorithmen mit der Eigenschaft, daß keine Variable, auf die A_2 schreibend zugreift in A_1 benutzt wird. Die Komposition der Algorithmen A_1 und A_2 ist der Algorithmus (C, δ), der wie folgt definiert ist:

- $C = C_1 \times C_2$
- $((c,d),(e,f)) \in \delta \Leftrightarrow (c,e) \in \delta_1 \vee (d,f) \in \delta_2$

Die Komposition zweier Algorithmen ergibt also einen neuen Algorithmus, dessen Zustandraum die Variablen beider Teilalgorithmen und dessen Prozeßkörper die Aktionen beider Einzelalgorithmen enthält. Die Einschränkung, daß A_1 nicht auf die Variablen von A_2 zugreifen darf, wird in folgendem Satz benötigt.

4.22 Satz (Komposition selbststabilisierender Algorithmen [136, S. 476][53, S. 15]) Seien A_1 und A_2 zwei selbststabilisierende Algorithmen. Wenn sich A_1 selbst zu einem Prädikat P_1 stabilisiert und A_2 sich selbst zu eine Prädikat P_2 stabilisiert falls P_1 gilt, dann stabilisiert sich unter starker Fairneß die Komposition von A_1 und A_2 zu P_2.

Beweis Gegeben sei eine beliebige Konfiguration c des Automaten, der aus der Algorithmenkomposition von A_1 und A_2 hervorgeht. Wegen der starken Fairneß und der Tatsache, daß sich A_1 selbst zu P_1 stabilisiert, enthält jede Ausführung, die in c startet, nach endlich vielen Schritten eine Konfiguration $c_1 \in P_1$ (Konvergenzbedingung von A_1).

Wegen der Abgeschlossenheit von A_1 und der Tatsache, daß A_2 keine Variablenwerte von A_1 verändert, wird P_1 nie falsifiziert. Da nun P_1 gilt, starke Fairneß vorausgesetzt ist, und sich A_2 selbst aus P_1 zu P_2 stabilisiert, wird jede Ausführung, die in c_1 beginnt, nach endlich vielen

Schritten eine Konfiguration $c_2 \in P_2$ erreichen (Konvergenzbedingung von A_2).

Insgesamt wird also aus einer beliebigen Konfiguration c nach endlich vielen Schritten immer eine Konfiguration $c_2 \in P_2$ erreicht werden. Also stabilisiert sich die Komposition von A_1 und A_2 selbst zu P_2. □

Bei der Komposition selbststabilisierender Algorithmen ist also wesentlich darauf zu achten, daß die nachfolgenden Teilalgorithmen keine Variablen der "vorgeschalteten" Algorithmen verändern und daß die Ausführung fair sein muß gegenüber allen Beteiligten. In *message passing*-Modellen muß man darum auch darauf achten, daß sich die Nachrichten der einzelnen Teilalgorithmen nicht gegenseitig behindern und von den einzelnen Algorithmen unterschieden werden können.

Man kann die Stabilisierungseigenschaft einer Algorithmenkomposition unter diesen Voraussetzungen auch einfach durch die Angabe einer Terminierungsfunktion nachweisen, vorausgesetzt man kennt die Terminierungsfunktionen f_1 und f_2 der beiden Teilalgorithmen A_1 und A_2 [136, S. 477]. Bezeichne $\gamma^{(i)}$ die Teilkonfiguration, die nur aus den Variablen von A_i besteht.[5] Dann ist die Funktion $f(\gamma) = (f_1(\gamma^{(1)}), f_2(\gamma^{(2)}))$ die Terminierungsfunktion der Komposition von A_1 und A_2. Dies wird dadurch deutlich, daß mit jedem Schritt von A_1 der Wert von f_1 erniedrigt wird, und mit jedem Schritt von A_2 genauso f_2 sinkt, ohne den Wert von f_1 zu verändern. Mit jedem Schritt wird also f lexikographisch kleiner und muß irgendwann terminieren.

Eine abgeleitete Beweistechnik. Wie Satz 4.22 zeigt, kann die Korrektheit einer selbststabilisierenden Algorithmenkomposition aus der Korrektheit der einzelnen Teile (unter gewissen Voraussetzungen) gefolgert werden. Gouda und Multari haben diese Idee abgewandelt und auf eine allgemeine Beweistechnik für Selbststabilisierung ausgedehnt. Sie nennen diese Technik *Konvergenzleiter* (*convergence stair*) [79, S. 451] und formalisieren sie mittels folgendem Satz (dessen Beweis eng mit dem von Satz 4.22 verbunden ist und deshalb nicht aufgeführt wird).

4.23 Satz (Beweis der Selbststabilisierung mittels einer Konvergenzleiter [79, S. 451]) Ein verteilter Algorithmus A stabilisiert sich selbst zu einem globalen Prädikat P, gdw. es eine endliche Folge von globalen Prädikaten $(P_1, P_2, \ldots, P_n)$ gibt, die folgenden drei Bedingungen genügen:

1. $P_1 = \mathit{true}$ und $P_n = P$.
2. Jedes Prädikat P_i ist bei normaler Ausführung von A abgeschlossen.
3. Jede Ausführung, die mit einem Zustand in P_i startet, hat einen Zustand in P_{i+1} (für $i < n$).

[5] Formal ist dies die Projektion einer Konfiguration γ auf die Variablenmenge von A_i.

Die drei Bedingungen werden *boundary*, *closure* und *convergence* genannt. Die Folge $(P_1, \ldots, P_n)$ heißt *Konvergenzleiter* (*convergence stair*).

Es ist interessant, diese Beweistechnik mit der Methode der Algorithmenkomposition zu vergleichen. Bei einem Beweis mittels Konvergenzleiter handelt es sich quasi um eine "Beweiskomposition": Ein "großer" Beweis kann so auf die Zusammensetzung vieler einzelner einfacher Teilbeweise zurückgeführt werden. Diese Technik ist natürlich in keiner Weise neuartig, doch ist die formale Fassung in Zusammenhang mit der Selbststabilisierung auf jeden Fall zu würdigen.

Muster-Komposition. Eine sehr schöne Komposition von vier Algorithmen präsentieren Arora und Gouda in ihrem Artikel "Distributed Reset" [10]. Mittels selbststabilisierender Algorithmen zur Herstellung eines aufspannenden Baumes und zur Verbreitung von Daten auf diesem Baum entwerfen sie einen fehlertoleranten "Rücksetzungs-Mechanismus" (*reset subsystem*), mit dem jeder Prozeß das Gesamtsystem in einem wohldefinierten Zustand zurücksetzen kann.[6] Ein solcher Mechanismus kann ein weiterer Baustein sein auf dem Weg, die Selbststabilisierungeigenschaft automatisch herkömmlichen Algorithmen hinzuzufügen.

4.4.2 Automatisierte Selbststabilisierung

Um selbststabilisierende Algorithmen für bestimmte Probleme zu bekommen, wäre es natürlich schön, wenn man auf den reichen Schatz herkömmlicher, nicht-fehlertoleranter Algorithmen zurückgreifen könnte und diese automatisch, etwa durch einen Compiler, um die Selbststabilisierungseigenschaft erweitern könnte. Konzeptuell ist dies eine Trennung von tatsächlichen Algorithmus und der Selbststabilisierungseigenschaft. Die Motivation dabei ist, daß die Komplexität der urspünglichen Algorithmen reduziert wird und daß aus diesem strukturierten Ansatz wiederverwendbare Module entstehen [94, S. 18].

Die Grundidee hinter den bisher vorgeschlagenen Verfahren ist, eine selbststabilisierende Basisschicht unter den originalen Algorithmus zu legen, die im Normalfall diesen nicht behindert, im Fehlerfall jedoch den Algorithmus in einen definierten Ausgangszustand zurücksetzt.

Selbststabilisierende Erweiterungen. Mit diesem Ziel veröffentlichten Katz und Perry 1993 einen "innovativen" [17, S. 269] Artikel namens "Self-stabilizing extensions for message-passing systems" [94]. Dort definieren sie zunächst, was eine *selbststabilisierende Erweiterung* $\tilde{A}$ (*self-stabilizing extension)* eines Algorithmus A ist, und zeigen anschließend, wie man solche Erweiterungen halbautomatisch erzeugen kann.

[6] Interessanterweise referenzieren Arora und Gouda weder Dolev *et al.* noch Herman.

4.24 Definition (selbststabilisierende Erweiterung eines Algorithmus [94, S. 19]) Ein Algorithmus $\tilde{A}$ ist eine *Erweiterung* (*extension*) eines Algorithmus A, gdw. die Projektion aller möglichen Abläufe von $\tilde{A}$ auf die Menge von Variablen und Nachrichten von A genau die Menge aller möglichen Abläufe von A ist.[7]

Ein Algorithmus $\tilde{A}$ ist eine *selbststabilisierende Erweiterung* (*self-stabilizing extension*) eines Algorithmus A, gdw. $\tilde{A}$ eine Erweiterung von A ist und $\tilde{A}$ sich selbst zur Spezifikation von A stabilisiert.

Informal gesprochen ist $\tilde{A}$ eine Erweiterung von A, wenn sich das Verhalten des Algorithmus nach außen hin nicht verändert.

Zur halbautomatischen Erzeugung selbststabilisierender Erweiterungen benötigt man ein System für das *message-passing*-Modell sowie ein entscheidbares Prädikat auf Konfigurationen, welches die "legalen" Zustände des Gesamtsystems beschreibt. Katz und Perry nennen die Basisschicht, die unterhalb des normalen Algorithmus eingefügt wird, einen *Aufseher* (*supervisor*). Grundbestandteile des Aufsehers sind

1. ein Schnappschußalgorithmus, um den Zustand des Gesamtsystems zu beobachten,
2. ein Rücksetzmechanismus, und
3. ein ausgewählter Beobachtungsprozeß (*evaluator*).

Konkret ist der Rücksetzmechanismus in den Schnappschußalgorithmus integriert worden, für den eine selbststabilisierende Variante eines der ersten Schnappschußalgorithmen von Chandy und Lamport [35] benutzt wird.

Der *evaluator* verschafft sich periodisch einzelne Schnappschüsse des Gesamtsystems und überprüft, ob das Legalitäts-Prädikat gültig ist. Falls es nicht gilt, werden in der folgenden Iteration des Schnappschusses die Zustände der einzelnen Prozesse in einen wohldefinierten Ausgangszustand zurückversetzt und quasi eingeforen. Bei der nächsten Iteration werden die Prozesse "aufgetaut" und das System beginnt wieder normal zu arbeiten.

Probleme des Verfahrens. Das Verfahren von Katz und Perry mag in seiner Einfachheit bestechen, hat aber einige inhärente Schwächen. Zum einen ist die Annahme, daß es ein entscheidbares Prädikat P zur Bestimmung legaler Zustände gibt, oft nicht zutreffend. Definiert man P als die Menge aller Konfigurationen, die sich von einer Menge von Startzuständen erreichen läßt, so ist P im allgemeinen Fall unentscheidbar [126, S. 62]. In vielen speziellen Fällen kann es deshalb durchaus sein, daß

[7] In der Definition wird ein *Stottern* (*stuttering*) des Ablaufes zugelassen; das bedeutet in der Projektion wird eine endliche Folge gleicher Zustände wie ein Zustand behandelt.

der Aufwand, das Legalitätsprädikat zu entscheiden, die Komplexität des Gesamtalgorithmus um eine Größenordnung übersteigt.

Ein zweiter Kritikpunkt ist nicht so sehr ein Versäumnis der Autoren sondern hängt mit den inhärenten Schwierigkeiten zusammen, überhaupt die Gültigkeit eines Prädikates in verteilten Systemen zu überprüfen [21, 127]. Der Wahrheitswert eines Prädikates ist nämlich nicht eine Frage der Berechnung sondern immer bezogen auf einen konkreten Ablauf und, wie in Kapitel 2 bereits angedeutet, gibt es in einer verteilten Berechnung viele Abläufe, die alle "gleich wahr" sind, d.h. dasselbe Resultat der Berechnung liefern. Da das Fehlen einer globalen Zeit vorausgesetzt wird, sind alle diese Abläufe auch "gleich gültig" und eine Antwort auf die Frage nach der Gültigkeit eines Prädikates ist im allgemeinen abhängig vom Beobachter [127, S. 163]. Dies bedeutet im Zusammenhang mit dem Algorithmus von Katz und Perry, daß der Beobachter auf der einen Seite illegale Zustände übersehen, und auf der anderen Seite illegale Zustände fälschlicherweise annehmen kann.

Stabile Prädikate und Selbststabilisierung. Allgemein geht es bei der Selbststabilisierung also immer darum, die mögliche Gültigkeit von $\neg P$ im Verlauf einer verteilten Berechnung zu entdecken und das System in einen Zustand zu überführen, in dem P gilt. Nur Prädikate, die bei normaler Ausführung nicht falsifiziert werden können, wenn sie einmal gelten, können bisher leicht durch einen Schnappschuß erkannt werden [127, S. 164]. Man nennt diese Art interessanterweise *stabile Prädikate*. Wenn $\neg P$ also ein stabiles Prädikat ist, wird das Verfahren von Katz und Perry dies auch erkennen und kann einen globalen *reset* ausführen. Doch selbst, wenn P ein stabiles Prädikat ist (was wegen der Abgeschlossenheit auch gegeben sein sollte), muß $\neg P$ nicht unbedingt auch stabil sein.

4.4.3 Lokale Überprüfbarkeit

Im allgemeinen ist die Frage nach der Gültigkeit eines Prädikates also unsinnig, solange sie sich nicht auf eine Menge konkreter Ausführungen bezieht. Cooper und Marzullo [42] haben zur Vereinfachung zwei Prädikatseigenschaften eingeführt, die in diesem Zusammenhang nützlich sind.

4.25 Definition (*possibly* und *definitely* [42]) Sei P ein Prädikat auf den Konfigurationen eines verteilten Systems. Für eine verteilte Berechnung B gilt $\mathit{possibly}(P)$, gdw. es eine Ausführung $A = (c_1, c_2, \dots)$ von B gibt, in der P für mindestens eine der darin vorkommenden Konfigurationen c_i gilt. Für eine verteilte Berechnung B gilt $\mathit{definitely}(P)$, gdw. für jede Ausführung $A = (c_1, c_2, \dots)$ von B das Prädikat P für mindestens eine darin vorkommende Konfiguration c_i gilt.

Falls also $\mathit{possibly}(P)$ gilt, ist die Beobachtung von P möglich. Sicher erkennen kann man P nur, falls $\mathit{definitely}(P)$ gilt.

Allgemein hat man im Zusammenhang der Selbststabilisierung also das Problem, $possibly(\neg P)$ zu entdecken. Hierzu gibt es bereits eine interessante Menge von Verfahren, die entweder zentral [63, 64] oder verteilt [61] arbeiten. Dabei machen sich die Autoren oft die Tatsache zunutze, daß Prozesse in manchen Fällen allein aufgrund lokaler Information bereits einen illegalen Zustand erkennen können. Ein Beispiel ist der wechselseitige Ausschluß: Sagen wir, ein Knoten x "weiß" (etwa durch Inspektion einer gemeinsamen Variablen), daß sein Nachbar y gerade das *token* hat. Wenn x nun aber das *token* von einem anderen Nachbarn z erhält, dann ist klar, daß mehr als ein *token* im System ist. Prozeß x kann daher folgern, daß die globale Invariante verletzt ist und ein Rücksetzen des Systems initiieren. Im Gegensatz zu den allgemeinen Verfahren, die man als "globales Prüfen mit globaler Korrektur" bezeichnen könnte, kann man hier von "lokalem Prüfen mit globaler Korrektur" sprechen.[8]

Die Klasse von Prädikaten, für welche diese Eigenschaft bereits untersucht wurden, nennen Schwarz und Mattern "locally decomposable" [127, S. 166]. Garg und Waldecker definieren hierfür den Begriff des lokalen Prädikates.

4.26 Definition (lokales Prädikat (*local predicate*) [63, S. 256]) Ein Prädikat P_i heißt *lokales Prädikat*, gdw. es nur Bedingungen enthält, die sich auf einen einzigen Prozeß im System beziehen.

Kann man ein Prädikat als Konjunktion von lokalen Prädikaten formulieren, ist die Entdeckung von $\neg P$ einfach lokal möglich. Betrachte hierfür ein solches Prädikat:

$$P \equiv P_1 \wedge P_2 \wedge \ldots \wedge P_n$$

Damit P wahr wird, müssen alle Teilprädikate P_i gelten. Gilt nur eines der P_i nicht, ist das Gegenteil von P (also $\neg P$) wahr. Die Entdeckung von $\neg P$ ist hier also equivalent zur Entdeckung von $\neg P_i$ für eines der P_i. Wenn alle P_i lokale Prädikate sind, ist dies einfach und effizient durchführbar. In diesen Fällen spricht man von lokaler Überprüfbarkeit (*local detectability* [127, S. 165] oder *local checkability* [17, 18]).

In Systemen mit gemeinsamen Speicher, in denen der Zustand eines Nachbarn quasi Teil des eigenen Zustandes ist, kann man die Legalitätsprädikate vieler Anwendungen gut als Konjunktion lokaler Prädikate formulieren und somit lokal überprüfen. In Systemen, die auf dem Versand von Nachrichten basieren, spielen die Zustände der Kanäle oft auch eine wichtige Rolle. (Die vereinfachte Annahme von Schwarz und Mattern [127, S. 162], daß der Zustand der Kanäle immer durch den lokalen Zustand eines Prozesses reflektiert wird, möchte ich nicht treffen.) Wegen ihrer besonderen Eigenschaften (mögliches Puffern von Nachrichten, ungewisse Verzögerungszeiten) lassen sich Kanäle jedoch nur "widerwil-

[8] Im Englischen benutzt man hierfür die Bezeichnungen *global checking and global correction*, bzw. *local checking and global correction* [17, S. 269].

lig" überprüfen. Garg, Chase, Mitchell und Kilgore stellen ein Verfahren [62] vor, welches eine bestimmte Klasse von Prädikaten auf Kanälen überprüfen kann (die sogenannten *monotonen Kanalprädikate*). Eine andere Möglichkeit ist jedoch, die Eigenschaften eines Kanals derartig einzuschränken, daß er leichter überprüfbar wird. Diesen Weg wählen Awerbuch, Patt-Schamir und Varghese. Sie erweitern den Begriff der lokalen Überprüfbarkeit auf Kanäle mit der folgenden Definition:

4.27 Definition (Lokale Überprüfbarkeit [17, S. 271]) Für eine Kante $e = (u, v)$ des Kommunikationsgraphen ist das *e-Kantensystem* (*e-subsystem*) das 4-Tupel bestehend aus dem Zustand der beiden Knoten u und v, sowie aus den Zuständen der beiden unidirektionalen Kommunikationsleitungen von u nach v bzw. von v nach u.

Für eine beliebige Konfiguration c bezeichne $[c]_e$ die Projektion von c auf das e-Kantensystem.

Ein Prädikat p ist ein *Kantenprädikat* für die Kante e (*link-predicate for e*), falls für zwei beliebige Konfigurationen c und c'

$$[c]_e = [c']_e \Longrightarrow p(c) = p(c')$$

gilt.

Ein Prädikat P ist *lokal überprüfbar*, falls es als Konjunktion von Kantenprädikaten geschrieben werden kann.

Erläuterung der Definition. Die Motivation hinter dieser Definition ist die, daß es vermutlich einfacher (und "billiger") ist, wenn jeder Prozeß einen Überblick über seine momentane engere Umgebung hält, als daß ein einzelner Prozeß die ganze Arbeit alleine machen muß. Die Umgebung schließt im Falle von *message passing*-Modellen auch die Kommunikationskanäle ein.Die Kantenprädikate sind Prädikate auf einem solchen kleinen "Teilzustand" der Konfiguration bestehend aus zwei Knoten und den Kommunikationskanälen zwischen ihnen. Die Definitionsbedingung sagt aus, daß solange sich das Teilsystem nicht ändert, auch das zugehörige Kantenprädikat gleich bleibt. Das bedeutet, daß ein solches Prädikat nicht "von außen" (durch globalen Einfluß) falsifiziert werden kann. (Kantenprädikate dürfen also nur Bezüge auf lokale Variablen der beiden beteiligten Nachbarprozesse enthalten, sowie auf den Zustand des zwischen ihnen liegenden Kanals.) Jeder beobachtende Prozeß kann durch die Überprüfung seiner Kantenprädikate zwar nie feststellen, ob insgesamt ein legaler Zustand besteht, er kann jedoch aus der Verletzung eines seiner Prädikate sofort auf das Bestehen eines "illegalen" Gesamtzustandes schließen.

In der Überprüfung eines Kantenprädikates tritt wieder das "Beobachtungsproblem" zutage. Varghese setzt in seiner Dissertation voraus, daß die einzelnen Kantenprädikate einer Spezifikation stabil sind [140, S. 101], damit sie durch einen Schnappschuß des Kantensystems überprüft werden können. In einem früheren Artikel zu diesem Thema wird diese Voraussetzung nicht gemacht. Es wird lediglich angenommen, daß die Kanäle aus den bereits erwähnten *UCDL*s bestehen (*UCDL*s sind Kanäle, deren Kapazität auf genau eine Nachricht beschränkt ist; vergleiche Fußnote 1 auf Seite 87). In solchen Fällen hat man bei Empfang einer Nachricht einen genauen Überblick über den Kanal.

Die der lokalen Überprüfbarkeit im Zusammenhang mit der Selbststabilisierung wurde erstmals 1990 vorgeschlagen von Afek, Kutten und Yung [3] (und findet ihre Fortsetzung in den Artikeln von Awerbuch, Patt-Shamir, Varghese und Dolev [17, 18], Arora, Gouda und Varghese [12] und der Dissertation von Varghese [140]). Sie ist leider in der Vergangenheit erstaunlich oft getrennt von den bereits oben angesprochenen Bemühungen verfolgt worden, globale Prädikate effizient zu erkennen.

4.4.4 Lokale Korrigierbarkeit

Die natürliche Fortsetzung der Methode des lokalen Prüfens verbirgt sich hinter den folgenden Fragen: Warum muß man einen illegalen Zustand immer global korrigieren? Wenn das globale Prädikat P die Konjunktion einer Menge von lokal überprüfbaren Prädikaten ist, müßte es doch genügen, wenn jeder Konten selbst "sein" Prädikat erfüllt? Eine Antwort auf diese Frage ist nicht einfach und muß ein diffiziles Wechselspiel zwischen lokaler Korrektur und dem dadurch verursachten Einfluß auf die Gültigkeit anderer Teilprädikate bedenken. Daß dies unter bestimmten Voraussetzungen tatsächlich gelingt, soll in diesem Abschnitt beschrieben werden.

Stabile Kantenprädikate. Eine ins Auge springende hinreichende Bedingung dafür, daß sich garantiert ein legaler Zustand einstellt, ist, wenn alle Kantenprädikate stabile Prädikate sind. So kann jedes Kantensystem autonom das lokale Prädikat p herstellen und hat damit seine Aufgabe erledigt. Da p stabil ist, kann es nicht mehr falsifiziert werden. Wenn alle Kantensysteme ihre Korrekturaktionen abgeschlossen haben, ist das System wieder in einem legalen Zustand [17, S. 271].

Man kann diese Beobachtung durch die Definition der *abhängigen Stabilität* verallgemeinern. Informal ist ein Prädikat p_1 stabil in Abhängigkeit von einem Prädikat p_2, falls p_1 stabil ist wenn p_2 stabil ist.

4.28 Definition (abhängige Stabilität (*dependent stability*) [17, S. 271]) Seien p_1 und p_2 Kantenprädikate. Dann ist p_1 stabil in Abhängigkeit von p_2, gdw. aus der Gültigkeit von $p_1 \wedge p_2$ in einer Konfiguration c die Gültigkeit von p_1 in jeder Folgekonfiguration von c gefolgert werden kann.

Offensichtlich wird ein System von Teilbedingungen zum Gesamtprädikat $P = \bigwedge p_i$ konvergieren, wenn der zwischen den einzelnen Kantenprädikaten p_i so definierte Abhängigkeitsgraph keine Zyklen enthält [17, S. 271].

Konvergenzanweisungen. Die Standarddefinition der Selbststabilisierung von Schneider (Definition 4.4) enthielt die zwei Bedingungen der Abgeschlossenheit und der Konvergenz. Beide bezogen sich auf ein globales Legalitätsprädikat P, welches zu erfüllen war. Beim Entwurf selbststabilisierender Algorithmen legt die Unterscheidung von Abgeschlossenheit und Konvergenz eine getrennte Formulierung von normalen Programmanweisungen ("Abgeschlossenheits-Anweisungen", *closure actions*) einerseits, und Konvergenzanweisungen (*convergence actions*) andererseits nahe [12, S. 425]. Während normale Programmanweisungen nur in Zuständen sinnvoll erscheinen, in denen P gilt, sollten Konvergenzanweisungen nur aktiv sein, wenn $\neg P$ gilt.

Hat man nun für ein bestimmtes Problem einen normalen Algorithmus konstruiert (also eine Menge von normalen Programmanweisungen, welche die Abgeschlossenheit von P garantieren), dann kann man nach folgendem Schema eine Menge von dazugehörigen Konvergenzaktionen entwerfen:

1. Zerlege P in eine Menge von Teilprädikaten P_i, die alle jeweils einzeln geprüft werden können und deren Konjunktion P ergibt.

2. Entwerfe für jedes Teilprädikat P_i eine Konvergenzaktion der Form:

$$\neg P_i \quad \longrightarrow \quad \text{“erfülle } P_i\text{”}$$

Die Zerlegung von P in Teilprädikate P_i ist eine Verallgemeinerung der lokalen Überprüfbarkeit aus Definition 4.27. Arora, Gouda und Varghese nennen die einzelnen Teilprädikate *Teilbedingungen* (*constraints*) und die Methode *constraint satisfaction* [12]. (Man beachte, daß die einzelnen Konvergenzaktionen trivialerweise nur aktiviert sind, wenn $\neg P$ gilt.) Es ist jedoch klar, daß Konvergenzaktionen dieser Art nicht notwendigerweise zur Konvergenz des Gesamtsystems führen, da die Erfüllung einer Teilbedingung P_i die Verletzung einer anderen Teilbedingung P_j zur Folge haben kann. Falls die Herstellung von P_j wiederum P_i verletzt ($i \neq j$), wird das System nie zu P konvergieren. Man benötigt also Methoden zur Validierung der Konvergenz (*convergence validation*) [12, S. 426].

Validierung der Konvergenz. Die Validierung der Konvergenz ist keine einfache Aufgabe, da sich — wie eben beschrieben — Konvergenzaktionen gegenseitig beeinflussen können und normale Programmaktionen möglicherweise gleichzeitig mit Konvergenzaktionen ausgeführt werden (schließlich ist nur $\neg P$ für die einzelnen Knoten erkennbar).

Das Problem der gegenseitigen Verletzung der Teilbedingungen bestand in einer zyklischen Abhängigkeit. Informal ist eine Teilbedingung P_i abhängig von P_j, wenn die Herstellung von P_j die Bedingung P_i verletzt. Wie bereits bei der Behandlung der abhängigen Stabilität besprochen wird ein System von Teilbedingungen zum Gesamtprädikat P konvergieren, wenn der so definierte Abhängigkeitsgraph keine Zyklen enthält.

Es kann jedoch manchmal schwierig sein, die gegenseitige Abhängikeit von Teilbedingungen zu überprüfen. Arora, Gouda und Varghese verallgemeinern deshalb diese Idee und liefern durch ihre Definition eines *Abhängigkeitsgraphen der Teilbedingungen* (*constraint graph*) [12, S. 426] ein hinreichendes Kriterium für Konvergenz.

4.29 Definition (Abhängigkeitsgraph der Teilbedingungen [12, S. 426f]) Sei K eine Menge von Konvergenzaktionen eines verteilten Algorithmus und P ein globales Legalitätsprädikat.

Der Abhängigkeitsgraph der Teilbedingungen (*constraint graph*) ist der gerichtete Graph, der wie folgt aus K entsteht:

- Jeder Knoten des Graphen ist bezeichnet mit einer Menge von Variablen, die in K benutzt werden. Eine Variable darf nur in der Bezeichnung eines Knotens auftauchen.
- Für jede Konvergenzaktion $k \in K$ gibt es eine Kante im Graphen. Wenn eine Konvergenzaktion k Variablen aus einer Menge u liest und Variablen aus einer Menge v schreibt, dann besteht die dazugehörige Kante zwischen den Knoten u und v.

Die Kanten im Abhängigkeitsgraphen spiegeln direkt die Teilbedingungen wieder, da es eine eindeutige Zuordnung (Bijektion) zwischen den Teilbedingungen und den Konvergenzaktionen gibt.

4.30 Beispiel Ein Programm habe drei ganzzahlige Variablen x, y und z und das Legalitätsprädikat P bestehe aus der Konjunktion der Terme $x \neq y$ und $x \leq z$. Die in einem ersten Schritt entworfenen Konvergenzaktionen könnten so lauten:

$$\begin{array}{lcl} x = y & \longrightarrow & x := y + 1 \\ x > z & \longrightarrow & x := z \end{array}$$

Es ist klar, daß die Ausführung einer von beiden Aktionen die Vorbedingung der jeweils anderen wahr machen kann. Der entsprechende Abhängigkeitsgraph ist in Abbildung 4.14(a) abgebildet.

In einem zweiten Schritt wird eine andere Menge von Konvergenzaktionen entworfen:

$$\begin{array}{lcl} x = y & \longrightarrow & y := x + 1 \\ x > z & \longrightarrow & z := x \end{array}$$

Die wechselseitige Abhängigkeit ist hier nicht mehr gegeben. Der dazugehörige Abhängigkeitsgraph ist in Abbildung 4.14(b) dargestellt.

Arora, Gouda und Varghese zeigen, daß Konvergenz garantiert ist, wenn der Abhängigkeitsgraph ein gerichteter Baum (*out-tree*)[9] ist [12,

[9] Ein gerichteter Baum ist ein schwach zusammenhängender, gerichteter Graph, bei dem ein Knoten (die Wurzel) keine eingehende Kante hat und alle anderen Knoten eine einzige inzidente Kante haben [12, S. 427].

Theorem 1]. Demnach konvergiert das System aus Abbildung 4.14(b) während das aus Teil (a) nicht notwendigerweise konvergiert. Die Autoren demonstrieren die Praktikabilität ihrer Definition durch die Formulierung eines selbststabilisierenden Wellenalgorithmus (*diffusing computation*) [12, S. 427f]. Außerdem verfeinern sie die notwendigen Konvergenzbedingungen und zeigen, daß auch Programme mit zyklischen Abhängigkeitsgraphen unter bestimmten zusätzlichen Voraussetzungen konvergieren [12, Theoreme 2 und 3]. In den Fällen, in denen der Nachweis der Konvergenz möglich ist, spricht Varghese von lokaler Korrigierbarkeit (*local correctability*) [140, S. 104].

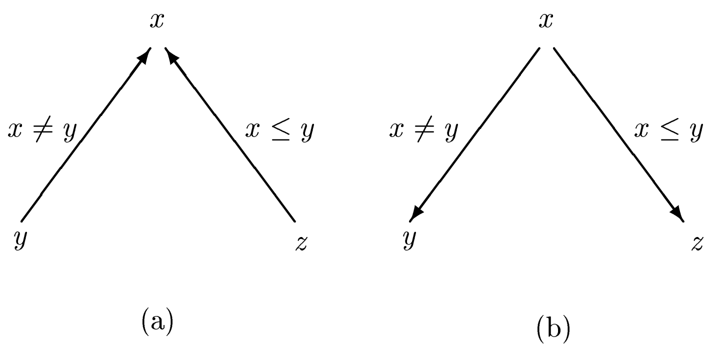

Abbildung 4.14: Abhängigkeitsgraphen aus Beispiel 4.30.

Diskussion des Verfahrens. Auf den ersten Blick erscheint die Methode des Abhängigkeitsgraphen eine neue Möglichkeit zu sein, selbststabilisierende Algorithmen in systematischer Weise zu entwerfen. Auf den zweiten Blick ist dies jedoch nur eine Neuformulierung und Verfeinerung des Kompositionstheorems 4.22 aus Abschnitt 4.4.1. Die Teilalgorithmen, die hier zusammengesetzt werden, sind die Konvergenzaktionen, die sich trivialerweise (das heißt in konstanter Zeit) selbst zu ihrer Teilbedingung P_i stabilisieren. Wenn zyklenfreie abhängige Stabilität vorausgesetzt wird (Definition 4.28), wird P_i nicht weiter falsifiziert, wenn alle anderen Teilbedingungen gelten, von denen P_i abhängt. Konsequenterweise gilt am Ende die Konjunktion aller P_i.

4.4.5 Eine Methodologie der Selbststabilisierung?

In den vergangenen Abschnitten wurden vier Aspekte des Entwurfes von selbststabilisierenden Algorithmen näher betrachtet, die jetzt einander gegenübergestellt werden sollen. Dabei fällt auf, daß sich die Algorithmenkomposition ein Entwurfsprinzip ist, welches sich von den anderen drei Begriffen (automatisierte Selbststabilisierung, lokale Überprüfbarkeit und lokale Korrigierbarkeit) durch seine Universalität abhebt. Dies läßt es vertretbar erscheinen, daß die letzten drei Begriffe zunächst getrennt von der Kompositionsmethode betrachtet werden sollen.

Entwurfsmethodik. Als ein wichtiges Ergebnis dieses Abschnittes ist eine Vorgehensweise beim Entwurf selbststabilisierender Algorithmen festzuhalten:

1. Betrachte das Legalitätsprädikat des Problems.
2. Ist es in irgend einer Form lokal überprüfbar, wende die Methode der lokalen Überprüfbarkeit an.
3. Ist das so spezifizierte Problem auch noch lokal korrigierbar, wende die Methode der lokalen Korrektur an.
4. Ansonsten begnüge dich mit den Methoden der automatisierten Selbststabilisierung oder der lokalen Überprüfung mit globaler Korrektur.

Es ist wichtig zu bedenken, daß die Möglichkeit, etwas lokal zu überprüfen bzw. es lokal zu korrigieren, nur von der Formulierbarkeit *des Problems* in Form von Kantenprädikaten abhängt. Ist ein Problem in dieser Art auszudrücken, kann automatisch lokale Überprüfung stattfinden. Eine genauere Analyse (unter Zuhilfenahme der hinreichenden Kriterien aus Abschnitt 4.4.4) muß dann klären, ob auch lokale Korrektur möglich ist. Das Problem bei dieser Methode liegt darin, daß die Kriterien für eine positive Beantwortung der Fragen Existenznachweise sind. Ich habe keine einfache Möglichkeit gefunden, die Grundlage eines Kriteriums für eine negative Aussage (“Problem A ist nicht lokal überprüfbar/korrigierbar!”) bilden könnte. Es bleibt dem Informatiker also nichts anderes übrig, als nach geeigneten Lösungen zu suchen.

Vermutung einer Hierarchie. Die Präsentation der verschiedenen Methoden zur Erlangung von Selbststabilisierung in den vergangenen Abschnitten suggeriert eine Hierarchie der Verfahren, die nun näher untersucht werden soll. Die Methoden der globalen Überprüfung und globalen Korrektur waren teuer, dafür aber immer anwendbar. Im Gegensatz dazu ist das Verfahren der lokalen Überprüfung und lokalen Korrektur viel effizienter, jedoch nicht immer anwendbar. Dazwischen liegt die Methode der lokalen Überprüfung mit globaler Korrektur.

Um formal zu zeigen, daß hier tatsächlich eine Hierarchie der Problemstellungen (und damit auch der Verfahren) besteht, müssen drei Punkte nachgewiesen werden:

- Jedes lokal überprüfbare Problem ist global überprüfbar.
- Jedes lokal korrigierbare Problem ist global korrigierbar.
- Jedes lokal korrigierbare Problem ist lokal überprüfbar.

Während der letzte Punkt die Kombinationsmöglichkeiten der Attribute global/lokal und überprüfbar/korrigierbar von vier auf drei reduziert (global überprüfbar und lokal korrigierbar ist nämlich dann immer lokal überprüfbar und lokal korrigierbar), beschreiben die ersten beiden Punkte eine echte Inklusionsbeziehung. Dies wird durch Abbildung 4.15 verdeutlicht.

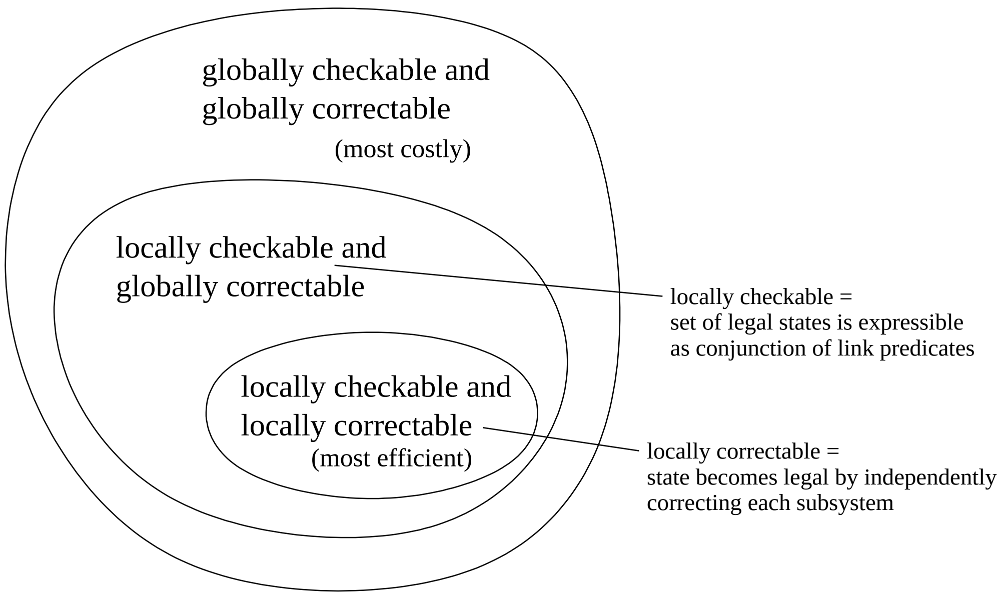

Abbildung 4.15: Vermutete Hierarchie der Problemstellungen und Verfahren zur Selbststabilisierung. Die Darstellung folgt den originalen englischen Ausdrucksweisen nach Varghese [140].

Beweisansätze. Daß ein lokal überprüfbares Prädikat P auch global überprüfbar ist, kann man recht einfach zeigen. Dazu benutzt man einfach einen der bereits existierenden Algorithmen, um $possibly(\neg P)$ zu entdecken [61, 63–65]. Um zu zeigen, daß lokal korrigierbare Prädikate auch global korrigierbar sind, benutzt man einfach einen globalen Rücksetzmechanismus [10, 94].

Der Nachweis, daß ein lokal korrigierbares Prädikat auch lokal überprüfbar ist, setzt eine geeignete Formalisierung des Begriffes der "lokalen Korrigierbarkeit" voraus, die in dieser Arbeit nicht geleistet wurde. Bei bereits existierenden Begriffsbildungen ist entweder die Forderung nach lokaler Überprüfbarkeit Teil der Definitionen von lokaler Korrigierbarkeit [140, S. 104] oder es fehlen geeignete Formalisierungen gänzlich [12]. Hier ist es notwendig, das ausgedrückte Konzept noch weiter zu untersuchen.

Echte Teilmengen. Daß die einzelnen Mengen nicht zusammenfallen, wird durch die Existenz einzelner Probleme gezeigt, die nur in die jeweilige Obermenge hineinfallen. Aufgrund der eingangs genannten Probleme ist dies jedoch schwierig. Varghese nennt den wechselseitigen Ausschluß als Beispiel für ein nicht lokal überprüfbares Problem [140, S. 27]. Sein Nachweis ist jedoch informal und durch Ansätze, wie sie im folgenden Kapitel aufgezeigt werden, teilweise widerlegbar.

Auf der anderen Seite nennt Varghese [140] (und andere [18, S. 327]) auch Beispiele für Probleme, die zwar lokal überprüfbar sind, jedoch nicht lokal korrigierbar. Dies sei immer dann der Fall, wenn es darum geht, sich auf irgendetwas zu einigen [140, S. 28] (als Beispiel bringt er das Drei-Farben-Färbungsproblem in einem Ring). Hierbei ist jedoch zu kritisieren, daß die Unmöglichkeit der Korrektur vermutlich eher durch die Symmetrie des Ringes hervorgerufen wird und nicht durch das Problem an sich. Für den formalen Nachweis einer Hierarchie muß also noch viel Arbeit geleistet werden.

Berücksichtigte Ideen. In diesem Abschnitt habe ich eine Reihe von Methoden vorgestellt, mit denen man selbststabilisierende Algorithmen entwerfen oder bereits bestehende Algorithmen mit einem "Selbststabilisierungsmodul" versehen kann. Dabei wurden die meiner Meinung nach wesentlichen Beiträge aus der Forschungsliteratur im Zusammenhang dargestellt. Es gibt noch zahlreihe weitere Ideen, die hier nicht berücksichtigt wurden, einerseits, weil ich sie im Bezug auf das Thema dieser Arbeit für unwesentlich hielt (beispielsweise Dolev und Hermans Konzept der *superstabilization* [51] oder Ghoshs *fault-containment* [69, 71]), und andererseits, weil ich nicht sah, wie sie in das formale Rahmenwerk dieses Abschnitts hineingearbeitet hätten werden können (beispielsweise Tels Methode der "minimalen Pfade" [136, S. 480ff]). Weitere Arbeiten können die Untersuchung dieser Teilprobleme und die Integration in das hier begonnene Rahmenwerk umfassen.

4.5 Ausblick

In diesem Kapitel ging es um Selbststabilisierung als eine allgemeine Methode zur Realisierung von Fehlertoleranz. Ich habe versucht, das noch relativ junge Forschungsgebiet in den Grundzügen zu umreißen und die für diese Arbeit wesentlichen Aspekte herauszuarbeiten. Diese umfassen:

- die Formalisierung: Selbststabilisierung ist Abgeschlossenheit und Konvergenz eines Legalitätsprädikates.

- die Beweisverpflichtungen: Selbststabilisierung verlangt einen Nachweis der *safety* und der *liveness*.

- das Potential: Selbststabilisierung ist eine einheitliche und elegante Methode zur Tolerierung von vorübergehenden Fehlern und hat damit direkte Anwendung in dynamischen Umgebungen.

- die Methodologie: Selbststabilisierung kann beim Algorithmenentwurf in strukturierter Weise berücksichtigt und in zum Teil automatisierter Form herkömmlichen Algorithmen hinzugefügt werden.

Die Darstellung dieses Kapitels ist jedoch in keiner Weise umfassend — viele Einzelheiten der genannten Aspekte verdienen eine nähere Betrachtung —, doch eine ganzheitliche Vorstellung des Forschungsgebietes war auch nicht das Ziel. Das wesentliche Ergebnis der vergangenen Abschnitte ist ein allgemeines Verständnis und ein besseres Gefühl für das Phänomen der Selbststabilisierung, welches in den Grundzügen einer strukturierten Methodologie mündete. Nun können diese Erkenntnisse bei der Entwicklung von selbststabilisierenden Replikationsalgorithmen angewandt werden.

5

Selbststabilisierende Replikationsalgorithmen

Inhaltsübersicht

Kurzfassung

In Kapitel 3 wurde das Themengebiet der Replikation vorgestellt. Dort habe ich auch ein paar gängige Verfahren präsentiert, mit deren Hilfe man replizierte Dienste (wie Telefonverzeichnisse oder Versandhauskataloge) in Computernetzwerken realisieren kann. Alle diese Verfahren hatten Nachteile, wenn es darum ging, temporäre Rechner- oder Netzausfälle zu tolerieren. In Kapitel 4 hingegen wurde das Konzept der Selbststabilisierung vorgestellt, welches geeignet ist, gerade diese Eigenschaften der Algorithmen zu verbessern und sie resistenter gegen eine viel größere Menge von möglichen Fehlern zu machen. In diesem Kapitel werden darum beide Gebiete miteinander verschmolzen: Das Ziel, welches am Ende erreicht wird, ist der Entwurf und die Realisierung selbststabilisierender Replikationsalgorithmen.

5.1 Einführung

In diesem Kapitel geht es darum, die Anwendung (schwach-konsistente Replikation) mittels einer Methode (Selbststabilisierung) fehlertolerant zu gestalten. Hierfür werde ich zwei Wege beschreiten:

1. Ausgehend von einer Formalisierung des Replikationsproblems werde ich systematisch die in Kapitel 4 erarbeitete Methodologie einsetzen, um selbststabilisierende (und damit fehlertolerante) Lösungen für schwach konsistente Replikation zu finden.

2. Ausgehend von den in Kapitel 3 vorgestellten Replikationsalgorithmen werde ich untersuchen, ob die Methode der Selbststabilisierung in irgendeiner Form eingesetzt werden kann, um deren Fehlertoleranz-Eigenschaften zu verbessern.

Dabei wird es interessant sein zu beobachten, welche Algorithmen durch welche Transformationen entstehen und wie sie sich gegenüber den herkömmlichen Replikationsalgorithmen verhalten.

5.2 Replikation mit einer Primärkopie

In diesem Abschnitt möchte ich ausgehend von einer Formalisierung des Replikationsproblems durch Einsatz der in Kapitel 4 erarbeiteten Methodologie einen ersten fehlertoleranten Algorithmus zu dessen Lösung erarbeiten.

5.2.1 Formalisierung von Replikation

Es gibt unterschiedliche Möglichkeiten, das Replikationsproblem formal zu spezifizieren. Der erste Ansatz stellt den eigentlichen Kern der Aufgabe in den Mittelpunkt, also die Herstellung wechselseitiger Konsistenz zwischen den Replikaten.

5.1 Definition (Konsistenzproblem) Gegeben sei ein System mit n unabhängigen Prozessen p_1 bis p_n. Jeder einzelne Prozeß p_i enthält eine lokale Variable r_i vom Typ *Data*. Das System ist in einem (wechselseitig) konsistenten Zustand, gdw. das folgende Prädikat P gilt:

$$P \equiv \forall i, j \in \{1, 2, \dots, n\} : r_i = r_j$$

Das *Konsistenzproblem* ist die Aufgabe, eine Konfiguration, in der $\neg P$ gilt, in eine Folgekonfiguration zu überführen, in der P gilt.

Glücklicherweise läßt sich das Konsistenzproblem wie gesehen durch ein einfaches Legalitätsprädikat (also eine *safety*-Eigenschaft) ausdrücken, so daß keine Probleme bei der Anwendung der Methodik aus Kapitel 4 auftreten. Ein zu entwickelnder Algorithmus sollte sich also selbst zu P stabilisieren. Implizit wird hier zunächst das gebräuchliche *shared memory*-Modell vorausgesetzt, in dem ein Prozeß direkten Zugriff auf die Replikationsvariablen seiner Nachbarn hat. Eine Übertragung in das *message passing*-Modell, wird anschließend geleistet. (Im *message passing*-Modell muß man auch noch die Zustände der Nachrichtenkanäle in P beschreiben.) Die Kommunikationstopologie sei aber in jedem Fall ein verbundener Graph mit bidirektionalen Kommunikationskanälen.

5.2 Bemerkung (Starke Konsistenz und maskierende Fehlertoleranz) Bei der Betrachtung von Definition 5.1 ist man versucht, sie nur auf schwache Konsistenz zu beziehen. Dabei schließt die Formulierung des Konsistenzproblems auch starke Konsistenz mit ein. Auch in verteilten Datenbanken mit strengen Konsistenzanforderungen gibt es globale Konfigurationen, in denen die Konsistenzbedingung verletzt ist. Im Unterschied zu Systemen mit schwacher Konsistenz dringt diese Tatsache aufgrund der Sperr- und Zugriffsmechanismen nicht nach außen. Stark-konsistente Replikationsverfahren entsprechen also einer Umsetzung des Konsistenzproblems mittels maskierender Fehlertoleranz. Da es hier nur um schwache Konsistenz geht, will ich mögliche weitere Wege in Richtung starker Konsistenz [13] nicht weiterverfolgen.

5.2.2 Ein erster Algorithmus

Ist P lokal überprüfbar? Zunächst muß die Frage gestellt werden, ob das Prädikat P lokal überprüfbar ist, ob man es also als Konjunktion von Kantenprädikaten ausdrücken kann. Zur Beantwortung dieser Frage kann man folgende Betrachtung anstellen: Der Allquantor (zusammen mit der Einschränkung auf den Indexmengen) läßt sich umformen in eine große Konjunktion, die alle Kombinationen der Indizes beinhaltet, also:

$$r_1 = r_1 \wedge r_1 = r_2 \wedge \ldots \wedge r_1 = r_n \wedge r_2 = r_1 \wedge \ldots \wedge r_n = r_n$$

Aufgrund der Reflexivität und Transitivität der Gleichheit genügt eine Kombination, deren transitive Hülle alle Kombinationen beinhaltet, also zum Beispiel:

$$r_1 = r_2 \wedge r_2 = r_3 \wedge r_3 = r_4 \wedge \ldots \wedge r_{n-1} = r_n$$

Damit dieser Term als Konjunktion von Kantenprädikaten ausgedrückt werden kann, muß man nur jeweils eine Kombination von Indizes nehmen, die in der Nachbarschaftsrelation enthalten ist. Wenn also beispielsweise *neighbours* die Menge von Paaren ist, zwischen denen eine Kommunikationsleitung besteht, dann ist P formulierbar als:

$$P \equiv (p_i, p_j) \in \mathit{neighbours} \Rightarrow r_i = r_j$$

Die einzelnen Gleichheiten $r_i = r_j$ bleiben erhalten, solange sich in den Prozessen p_i und p_j, sowie bei der Kommunikation zwischen den beiden nichts ändert. Das bedeutet, daß die einzelnen Gleichheiten Kantenprädikate für die Kante zwischen den Prozessen p_i und p_j sind. Da P also ausdrückbar als Konjunktion von Kantenprädikaten ist, ist P im Sinne von Definition 4.27 auf Seite 109 lokal überprüfbar.

Ist P lokal korrigierbar? Nach der Feststellung lokaler Überprüfbarkeit, muß untersucht werden, ob P auch lokal korrigierbar ist. Diese Eigenschaft ist für die Effizienz eines erzeugten Algorithmus sehr vorteilhaft.

Ein lokal überprüfbares Prädikat ist lokal korrigierbar, falls seine Gültigkeit durch die Korrektur seiner einzelnen Kantenprädikate hergestellt werden kann. Sagen wir also, ein Prozeß p_i erkennt, daß eines seiner Kantenprädikate verletzt ist, daß also $r_i \neq r_j$ für ein $(i, j) \in \mathit{neighbours}$. Eine Möglichkeit für Prozeß p_i, diese Bedingung wieder zu erfüllen, ist, die Anweisung

$$r_i \neq r_j \quad \longrightarrow \quad r_i := r_j$$

in seinen Anweisungsteil mitaufzunehmen. Es verbleibt die Validierung der Konvergenz.

5.3 Beispiel (Validierung der Konvergenz in einem einfachen System) In diesem Beispiel soll ein einfaches System bestehend aus drei Knoten und zwei Kanten betrachtet werden, die in Form einer linearen Kette miteinander verbunden sind (vergleiche Abbildung 5.1 links). Ein Replikationsalgorithmus muß demnach beispielsweise das Prädikat $P \equiv r_1 = r_2 \wedge r_1 = r_3$ erfüllen. Gemäß dem Vorschlag, daß jeder Knoten periodisch den Wert seiner Nachbarreplikate lese und seinen Wert bei Bedarf diesen angleiche, ergeben sich vier mögliche Konvergenzaktionen:

(a) auf Knoten 1: $r_1 \neq r_2 \rightarrow r_1 := r_2$

(b) auf Knoten 1: $r_1 \neq r_3 \rightarrow r_1 := r_3$

(c) auf Knoten 3: $r_3 \neq r_1 \rightarrow r_3 := r_1$

(d) auf Knoten 2: $r_2 \neq r_1 \rightarrow r_2 := r_1$

Gemäß der Methode von Arora, Gouda und Varghese, die auf Seite 112 vorgestellt wurde, habe ich in Abbildung 5.1 den Abhängigkeitsgraphen (Definition 4.29) auf der rechten Seite in der Abbildung aufgeführt. Die Pfeile sind dabei mit den Buchstaben derjenigen Konvergenzaktionen bezeichnet, denen sie entsprechen. Nach dem Kriterium von Arora, Gouda und Varghese von Seite 112 ist es das erklärte Ziel, einen (auswärts) gerichteten Baum als Graphen zu haben, denn dann ist die Konvergenz gesichert. Wenn man also die Konvergenzaktion (a) wegläßt, kann man entweder (b) oder (c) weglassen. Entsprechend könnte man auch die Konvergenzaktion (b) entfernen, müßte dann aber auch die Aktion (a) wegnehmen. Man darf aber nicht (c) und (d) gleichzeitig entfernen (es ist klar, daß dann die Konvergenz nicht gewährleistet ist).

$$P \equiv r_1 = r_2 \wedge r_1 = r_3$$

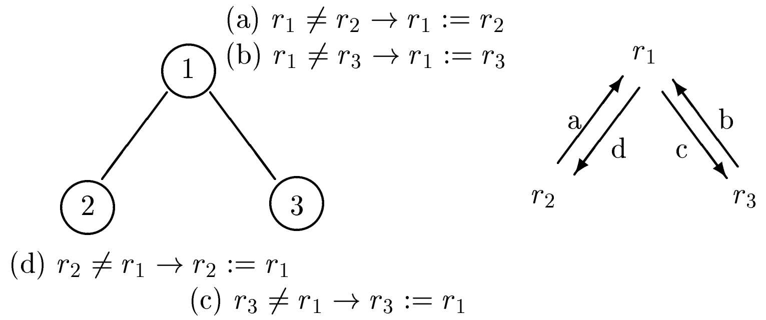

Abbildung 5.1: Schematische Visualisierung der Analogien zwischen der Kommunikationstopologie mit Konvergenzaktionen (links) zum Abhängigkeitsgraphen (rechts) in Beispiel 5.3.

Die Entscheidung, nach welcher Seite hin ein Kantenprädikat korrigiert werden muß, ist also nicht so einfach. Es gibt immer auch Kombinationen von Konvergenzanweisungen, welche die Voraussetzungen des Kriteriums von Arora, Gouda und Varghese nicht erfüllen, und diese gilt es zu meiden. Wie aber im Anschluß an Definition 4.29 auf Seite 112 bereits gesagt wurde, besteht eine eindeutige Zuordnung zwischen Konvergenzaktionen und Kanten im Abhängigkeitsgraph. In Falle der Replikation besteht aber auch eine eindeutige Zuordnung zwischen den Knoten des Abhängigkeitsgraphen und den Knoten des Kommunikationsgraphen. Dies resultiert aus der Tatsache, daß jeder dieser Knoten genau eine "Replikationsvariable" r_i enthält und jede Konvergenzaktion genau eine solche Variable schreibt. Wenn eine Korrektur des Kantenprädikates also immer in einer Richtung stattfindet, und wenn diese "Richtungen" zusammen im Kommunikationsgraphen einen gerichteten Baum ergeben, dann stabilisiert sich das System zum Legalitätsprädikat P (in der Sprache von Arora, Gouda und Varghese wäre P dann lokal korrigierbar. Insgesamt führen diese Beobachtungen zu einem ersten einfachen selbstabilisierenden Replikationsalgroithmus.

5.4 Algorithmus (erster Replikationsalgorithmus) Gegeben ist ein System mit n Prozessen, die durch eine konstante Verweisstruktur in Form eines Baumes verbunden sind. Dies wird erreicht durch die Speicherung der Identität des jeweiligen Elternknotens von Knoten i in einer lokalen Konstante p. Die Wurzel des Baumes ist durch die Bedingung $p = i = 0$ identifiziert. Alle Prozesse besitzen eine lokale Replikationsvariable r_i vom Typ *Data*.

Jeder Knoten überprüft periodisch, ob seine eigene Replikationsvariable denselben Wert besitzt, wie die Replikationsvariable seines Elternknotens. Falls beiden einen unterschiedlichen Wert besitzen, setzt der Knoten seinen Wert auf den Wert des Elternknotens.

Eine Realisierung des Algorithmus für den Fall lokal gemeinsamen Speichers ist in Abbildung 5.2 dargestellt.

process *replication node i*
  const i **init** {* own node id *}
   p **init** {* id of parent node *}
  var $r \in Data$ {* replication data *}
  begin
    $i \neq 0 \wedge r \neq r_p \quad \longrightarrow \quad r := r_p$
    {* "fault" actions: *}
  ⟦ $i = 0 \wedge$ {* update wanted *} $\longrightarrow$ $r :=$ {* new value *}
  end

Abbildung 5.2: Erster Replikationsalgorithmus zur Lösung des Konsistenzproblems.

Bemerkungen zu Algorithmus 5.4. Der eben vorgestellte Algorithmus ist auf den ersten Blick trivial, doch lohnt es sich trotzdem, nochmals seine Eigenschaften zusammenzufassen:

- Algorithmus 5.4 stabilisiert sich selbst zu wechselseitiger Konsistenz. Dabei toleriert er jede Form von vorübergehenden Fehlern (Knoten- und Verbindungsausfall, Korrumpierung des lokalen Speichers der Prozesse).
- Als "Fehler" im eigentlichen Sinne wird auch eine Änderung des Wertes der Variable r in der Wurzel aufgefaßt. Die Konsistenz wird also erst erreicht, wenn auch Änderungen dieses Wertes aufhören.
- Die schwache Konsistenz ist quasi ein "Nebenprodukt" der Selbststabilisierung: Die Zustände, in denen $\neg P$ gilt, sind unerwünscht, aber solange vertretbar, wie die Konvergenzzeit in einem für die Anwendung erträglichen Rahmen bleibt.
- Der "dominante" Wert, der nach der Selbststabilisierung in allen Knoten gespeichert ist, ist der Wert der Wurzel des in den Variablen p implizit gespeicherten Baumes.

Die Eigenschaft der Selbststabilisierung hat hier die Konsequenz, daß man innerhalb der Knoten keinen stabilen Speicher annehmen muß. Allenfalls könnte (und sollte) man in der Wurzel des Knotens einen solchen sicheren Speicher annehmen, damit auch nach einem Absturz des Wurzelprozesses oder nach einer vorübergehenden Perturbierung seines Zustandes nicht "falsche" Werte im Netzwerk verteilt werden. Die Herstellung von Konsistenz ist aber auch ohne stabilen Speicher garantiert. (Allerdings kann man über den Sinn dieser Form von Konsistenz streiten.)

Die Feststellung eines "dominanten" Wertes innerhalb der Wurzel zeigt, daß man hier im Prinzip einen *broadcast* des Wurzelknotens innerhalb des Netzwerkes realisiert hat und bekräftigt die enge Verbindung zwischen Replikations- und Gruppenkommunikationsmechanismen. Wenn also der Wurzelknoten eine Nachricht an alle Prozesse senden will, setzt er seine eigene Replikationsvariable auf einen entsprechenden Wert und wartet. Nach einer bestimmten Zeit werden alle anderen Knoten garantiert diesen Wert annehmen und die Nachricht damit erhalten. Eine Anwendung des Algorithmus wäre also immer dann sinnvoll, wenn es eine ausgezeichnete Primärstation (*primary copy*) gibt. Beispielsweise ist dies bei der Verteilung von Software oder der Bereitstellung "zentraler" Auskunftsdienste gegeben.

Bewertung und Analogien. Die Fehlertoleranzeigenschaften von Algorithmus 5.4 und seine Einfachheit sind die positiven Punkte bei seiner Bewertung. Wegen der eingesetzten Methodik toleriert er auf der anderen Seite aber nur vorübergehende Fehler; daß dabei die Änderung des "dominanten" Replikationswertes der Wurzel auch als "Fehler" gilt, ist unschön. Auch ist die Forderung nach einer konstanten Kommunikationstopologie, die zudem noch einen gerichteten Baum darstellen soll, eigentlich zu stark. Dynamische Änderungen dieser Topologie sind auf den ersten Blick nicht einfach möglich. Dadurch, daß der Algorithmus einfach nur Konsistenz herstellt, ohne auf den bei allen Knoten hergestellten Wert zu achten, löst er das "Korrektheitsproblem" (also zu beachten, daß nur "richtige" Werte auf den Knoten verteilt werden) nicht. Dies kann zwar durch die Forderung nach stabilem Speicher in der Wurzel gelöst werden, würde aber wieder in zusätzlichen (stärkeren) Voraussetzungen für den Algorithmus münden.

Der Algorithmus hat deutliche Ähnlichkeiten zum *TSAE*-Protokoll von Golding (vergleiche Abschnitt 3.6.2). Das periodische Überprüfen und die Angleichung des eigenen Replikationswertes entsprechen den dort verwendeten *anti-entropy sessions*. In *TSAE* werden neue Informationen auch immer implizit auf einem Baum verbreitet, der durch die Kanten beschrieben ist, auf dem ein Knoten die Nachricht erstmals erhielt. Jedoch ist dieser Baum nicht von vornherein festgeschrieben. In *TSAE* kann zudem jeder Knoten senden, in Algorithmus 5.4 nur eine Station (nämlich die Wurzel). Dies und der von vornherein feste Kommunikationsbaum ermöglichen es, auch ohne Zeitstempel auszukommen: Ein "anderer" Wert ist immer auch ein "neurer" Wert; so kann man die Replikationsvariable r quasi als Empfangsvektor mit einem Element bezeichnen, also die "Sicht" jedes einzelnen Prozesses vom Zustand der Wurzel.

Das Finden weiterer Analogien zu *TSAE* (etwa was den Bei- oder den

Austritt von Knoten im Baum anbelangt) ist wegen der unterschiedlichen Modelle, die den Algorithmen zugrundeliegen, schwierig. Auch sehe ich keine Analogien zum *HPP*-Protokoll, es sei denn, man möchte die der Kommunikation zugrundeliegende feste Baumstruktur als eine solche ansehen.

5.2.3 Weitere Eigenschaften des einfachen Algorithmus

Aus der Konstruktion des Algorithmus kann man ableiten, daß ein durch die Wurzel einmal geschriebener Wert nach endlicher Zeit alle Knoten im Netzwerk erreicht. Voraussetzung dafür ist jedoch, daß die "Schreibfrequenz" der Wurzel nicht allzu hoch ist. Ein neuer Wert darf erst geschrieben werden, wenn alle seine Kinderknoten im Baum vom alten Wert Kenntnis erlangt haben. Die Tatsache, daß alle Knoten auch alle Werte mitbekommen, wird durch folgendes Lemma nachgewiesen, dessen Beweis aus Platzgründen in den Anhang (Abschnitt A.1) verlegt wurde.

5.5 Lemma Gegeben sei ein System mit n Knoten, auf dem Algorithmus 5.4 abläuft. Ein Knoten (auch die Wurzel) ändere in Algorithmus 5.4 erst dann ihren Wert r, wenn alle Kinderknoten im Baum den vorherigen Wert übernommen haben. Die Wurzel schreibe niemals zweimal hintereinander denselben Wert nach r_0. Ein Schreibereignis e_1 auf Knoten p_1 *entspricht* einem Schreibereignis e_2 auf Knoten p_2, gdw. mit e_1 und e_2 auf den jeweiligen Knoten dieselben Werte geschrieben werden. Unter der Annahme schwacher Fairneß gibt es nach spätestens n Schreiboperationen der Wurzel für jede weitere solche Schreiboperation eine ihr entsprechende Schreiboperation auf der Replikationsvariable in allen Knoten und umgekehrt.

Faßt man die Schreiboperation *nicht* als "Fehler" auf und nimmt man die Voraussetzungen aus Lemma 5.5 hinzu, ist das Prädikat, zu dem sich der Algorithmus selbst stabilisiert, etwas komplexer und durch das folgende Lemma beschrieben. Es besagt, daß sich die "neuen" Datenwerte der Wurzel "in Wellen" entlang des Baumes ausbreiten und enthält eine Definition des Begriffes der "aktiven Konsistenz." Der Beweis ist in Anhang A.2 nachzulesen.

5.6 Lemma (aktive Konsistenz) Sei $F = (f_1, f_2, f_3, \dots)$ eine Folge von Schreiboperationen der Wurzel und $W = (w_1, w_2, w_3, \dots)$ die Folge der durch F nach r_0 geschriebenen unterschiedlichen Datenwerte. Ein Knoten (auch die Wurzel) ändere erst dann ihren Wert r, wenn alle Kinderknoten im Baum den vorherigen Wert übernommen haben. Sei *num* eine Funktion, die auf einem beliebigen Knoten i einem in r_i gespeicherten Datenwert die Nummer der Schreiboperation zuordnet, die ihn ursprünglich nach r_0 schrieb ($num(w) = 0$ gdw. w ein Wert ist, der durch einen Fehler entstand). Schreiboperationen sollen nicht als vorübergehende Fehler gelten.

Unter der Annahme schwacher Fairneß stabilisiert sich der einfache Replikationsalgorithmus nach den ersten n Schreiboperationen zum Prädikat:

$$
\begin{aligned}
P \equiv \forall i \in \mathit{Nodes} : i \neq 0 \Rightarrow \\
&(r_i = r_{p_i} \wedge \mathit{num}(r_i) = \mathit{num}(r_{p_i})) \vee \\
&(r_i \neq r_{p_i} \wedge \mathit{num}(r_{p_i}) = \mathit{num}(r_i) + 1)
\end{aligned}
$$

Die Zustände, in denen P gilt, heißen *aktiv-konsistente Zustände*.

Aktiv-konsistente Zustände sind sozusagen Zwischenzustände auf dem Weg zur Konsistenz. Dieses Lemma und der Begriff der aktiven Konsistenz wird später noch Verwendung finden. Lemma 5.5 läßt sich mit Hilfe der aktiven Konsistenz noch spezialisieren. Der Beweis hat große Ähnlichkeiten zum Beweis von Lemma 5.5 und wird deshalb nicht geführt.

5.7 Lemma Die Voraussetzungen seien dieselben wie in Lemma 5.5. Das System befinde sich außerdem in einem aktiv-konsistenten Zustand. Dann findet zu jedem Schreibereignis der Wurzel garantiert auf jedem Knoten ein entsprechendes Schreibereignis statt.

Lemma 5.7 garantiert, daß unter bestimmten Voraussetzungen alle Knoten alle Werte aus der Wurzel auch mitbekommen.

5.2.4 Dynamische Kommunikationstopologie

Eine mögliche Erweiterung von Algorithmus 5.4 besteht darin, die Kommunikationstopologie dynamisch zu gestalten. Ziel dabei ist es, daß der Algorithmus im Fehlerfall automatisch die Kommunikationstopologie restrukturiert und wieder in einen korrekten (Spannbaum-) Zustand überführt. Hier bietet es sich an, die Methode der Algorithmenkomposition zu verwenden.

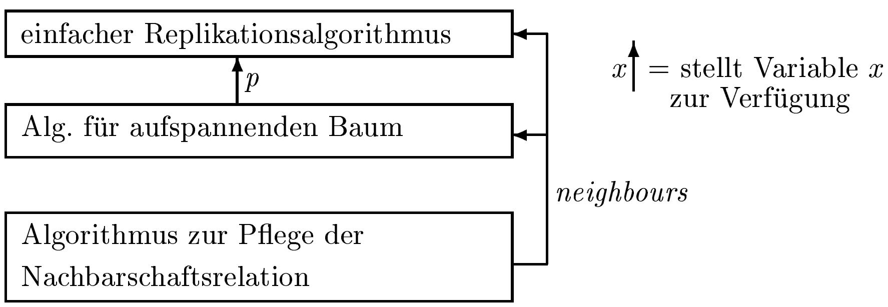

Abbildung 5.3: Die verschiedenen Schichten in Algorithmus 5.8 und die Lesebeziehungen unter den Variablen.

Die Berechnung eines aufspannenden Baumes kann effizient durch einen selbststabilisierenden Standardalgorithmus gelöst werden, wie beispielsweise der Algorithmus 4.10 von Chen, Yu und Huang [39] aus dem Abschnitt 4.2.5. Bei Einsatz dieses Algorithmus wäre der Elternknoten p nun eine Variable und würde sich bei einer Umstrukturierung oder einem Fehler automatisch ändern. Aufbauend auf diesem Wert kann dann der oben vorgestellte Replikationsalgorithmus arbeiten. Die logische Anordnung beider Algorithmen ist in Abbildung 5.3 dargestellt. Dort ist (in Anlehnung an einen Algorithmus von Arora und Gouda [10]) noch eine zusätzliche Basisschicht eingefügt worden, die die Nachbarschaftsrelation (repräsentiert durch die Menge *neighbours*) auf dem neusten Stand hält. Diese Schicht hat die Aufgabe, durch periodische Kommunikation mit den Nachbarknoten festzustellen, ob diese noch "am Leben" sind und entsprechend die Menge *neighbours* zu aktualisieren.[1]

5.8 Algorithmus (Replikationsalgorithmus für eine dynamische Kommunikationstopologie) Gegeben ist ein System mit n Prozessen. Diese Prozesse prüfen regelmäßig, ob ihre Nachbarn im physikalischen Netzwerk noch antworten und speichern die Ergebnisse dieser Operationen in einer Menge *neighbours*, welche die jeweils aktuelle Menge der antwortenden Nachbarn enthält.

Aufbauend auf dieser Basisschicht arbeitet der Algorithmus 4.10 zur Herstellung eines aufspannenden Baumes von Seite 83. Darin ist die Menge *neigbours* nun eine Variable.[2] Darauf aufbauend arbeitet der einfache Replikationsalgorithmus 5.4, dessen Bezeichner p nun keine Konstante sondern eine Variable ist.

Aus der schichtweisen Komposition der drei vorgenannten Algorithmen entsteht ein selbststabilisierender Replikationsalgorithmus für dynamische Kommunikationstopologien.

Korrektheit. In Abbildung 5.3 sind die durch die Schichten jeweils zur Verfügung gestellten Variablen in Form von Pfeilen eingezeichnet. Es ist einfach zu sehen, daß keine Variable einer unterliegenden Schicht von einer höheren Schicht geschrieben wird. Gemäß Satz 4.22 von Seite 103 über die Komposition selbststabilisierender Algorithmen ist der zusammengesetzte Algorithmus ebenfalls selbststabilisierend und konvergiert unter der Annahme starker Fairneß zum Legalitätsprädikat der obersten Schicht, also zu wechselseitiger Konsistenz.

Bemerkungen zu Algorithmus 5.8. Der Algorithmus 4.10 verlangt einen ausgewählten Wurzelprozeß *root*, auf dem ein anderer Algorithmus

[1] Eine solche Schicht muß nicht unbedingt in einem erhöhten Nachrichtenaufwand münden, wenn man die "Lebenszeichen" an ohnehin verschickte Nachrichten anhängt (Methode des *piggybacking*).

[2] Zusätzlich müßte die Bedingung $p = root$ durch $p = 0$ ersetzt werden, wenn man die in Algorithmus 4.10 noch in zwei Abbildungen dargestellten Algorithmen für die Wurzel und die anderen Knoten in einer Darstellung vereinigen wollte.

abläuft als auf den restlichen Knoten. Dies wurde bei der Besprechung von Algorithmus 5.4 als Nachteil angesehen, da bei längerfristigem Ausfall dieses ausgewählten Knotens das gesamte System blockiert wäre. Bei der Anwendung innerhalb von Algorithmus 5.8 ist dieser Nachteil nicht mehr so schwerwiegend, da bereits ein "ausgewählter" Knoten, die ausgezeichnete Primärstation, existiert, bei der eine besondere Pfege durch einen Dienstanbieter angenommen werden kann. Es muß nur sichergestellt werden, daß die ausgewählten Knoten beider Algorithmen auch ein und derselbe reale Knoten im Netzwerk sind.

Der zusammengesetzte Algorithmus ist in den Abbildungen 5.4 und 5.5 dargestellt. Die drei Schichten wurden hierbei in eine einheitliche Darstellung überführt und in einem einzigen Prozeßkörper aufgezeichnet. Kommentare sollen die logische Unterscheidung zwischen den Schichten hervorheben.

process *replication root*
  const l **init** 0
  var $r \in \mathit{Data}$ {* replication data *}
  begin
    {* update wanted *} $\longrightarrow$ $r :=$ {* new value *}
  end

Abbildung 5.4: Lokaler Algorithmus des Wurzelknotens aus Algorithmus 5.8.

5.2.5 Übertragung in das Nachrichtenmodell

Eine Übertragung von Algorithmus 5.8 in das Nachrichtenmodell ist vor allem deswegen wünschenswert, da hierdurch der Algorithmus auf vielen realen Architekturen direkt eingesetzt werden kann. Insbesondere ist dies ein wesentlicher Schritt hin zur Simulation der verschiedenen Algorithmen, wie sie in Kapitel 6 noch besprochen werden sollen. Bei der semi-manuellen Transformation, wie sie hier durchgeführt wird, können sich natürlich wieder neue Fehler einschleichen, doch kann man sich dabei ein paar Prinzipien zunutze machen, die einem dem Übergang erleichtern.

Zunächst muß der Algorithmus in eine Form überführt werden, in denen die Kommunikation über die in Kapitel 2 angesprochenen Kommunikationsregister erfolgt. Dies kann dadurch erreicht werden, daß man für jede Variable, die von einem Nachbarn gelesen wird, eine eigene lokale "Kommunikationsvariable" bereitstellt, die bei Änderung des lokalen Wertes jeweils aktualisiert werden muß [10, S. 1031]. Der nächste Schritt ist die Überführung dieser Variablen auf den Nachbarknoten selbst und die Aktualisierung ihres Wertes durch das Senden einer Nachricht.

process *replication node i*
const n **init** {* number of nodes in network *}
i **init** {* own node id *}
var *neighbours* {* set of neighbour nodes *}
$l \in \{1, 2, \ldots, n\}$ {* level of node in tree *}
$p \in neigbours$ {* parent in tree *}
$r \in Data$ {* replication data *}
begin
{* spanning tree algorithm: *}
$l \neq n \wedge l \neq l_p + 1 \wedge l_p \neq n \longrightarrow l := l_p + 1$
$[\!]$ $l \neq n \wedge l_p = n \longrightarrow l := n$
$[\!]$ $m \in neighbours \wedge l = n \wedge l_m < n - 1 \longrightarrow l := l_m + 1;\ p := m$
{* replication algorithm: *}
$[\!]$ $r \neq r_p \longrightarrow r := r_p$
end

Abbildung 5.5: Lokaler Algorithmus eines Nicht-Wurzelknotens aus Algorithmus 5.8.

Schicht 1: Pflege der Nachbarschaftsbeziehung. Die Pflege der Nachbarschaftsbeziehung setzt voraus, daß ein Knoten seine "physikalischen" Nachbarn kennt, also alle Knoten, mit denen er direkt und ohne Umwege kommunizieren kann. Diese Menge kann man als konstant ansehen. Sie soll mit N bezeichnet werden. Wenn ein Knoten eine Nachricht von einem solchen Nachbarn erhält, so nimmt er dessen Identität in die Menge *neighbours* der als vorhanden angesehenen Nachbarn auf und schickt an diesen eine Nachricht zurück, daß auch er selbst "am Leben" ist. Hat ein Prozeß nach einer gewissen Zeit keine Nachricht von seinem Nachbarn j bekommen, so nimmt er bis auf weiteres an, daß dieser nicht mehr am Leben ist und entnimmt ihn aus der Menge *neighbours*.

Der lokale Algorithmus der ersten Schicht ist in Abbildung 5.6 dargestellt. Seine Korrektheit folgt aus dem Beweis des folgenden Satzes, der in Anhang A.3 nachzulesen ist.

5.9 Satz (Korrektheit des Algorithmus zur Pflege der Nachbarschaftsbeziehung) Ein Knoten n ist ein *physikalischer Nachbar* von Prozeß p, gdw. es einen bidirektionalen Kommunikationskanal zwischen p und n gibt. Ein physikalischer Nachbar n ist ein *toter Nachbar* (*down-neighbour*) von Prozeß p, gdw. der Nachrichtenkanal zwischen n und p nicht mehr funktioniert. Ein nicht-toter Nachbar von p wird als *lebender Nachbar* (*up-neighbour*) bezeichnet.

Die Kommunikationskanäle seien derart, daß sie insgesamt nur eine einzige Nachricht puffern können, wenn sie funktionieren. Wenn sie nicht funktionieren, gehen alle darauf verschickten Nachrichten verloren. Funktionierende Kanäle haben eine endliche Verzögerungszeit. Die Anweisung $\mathbf{timeout}_j$ in Prozeß p sei genau dann "wahr", wenn keine Nachricht mehr zwischen p und j unterwegs ist.

Unter all diesen Voraussetzungen stabilisiert sich der Algorithmus aus Abbildung 5.6 zum Prädikat:

$$
\begin{aligned}
P \equiv \forall p \in \mathit{Nodes} : \mathit{neighbours}_p = \{&\text{alle lebenden Nachbarn von } p\} \wedge \\
&\text{"zwischen zwei lebenden Nachbarn ist genau eine Nachricht unterwegs"} \wedge \\
&\text{"zwischen einem lebenden und einem toten Nachbarn} \\
&\text{ist maximal eine Nachricht unterwegs"}
\end{aligned}
$$

Zu beachten ist, daß ein toter Nachbar durch das Auf und Ab des Kanals wieder lebendig werden kann und umgekehrt. Diese Tatsache allein ist im Sinne der Selbstabilisierungseigenschaft des Algorithmus kein Fehler, da er sich auch bei anhaltend toten Nachbarn selbst stabilisiert. (Man könnte höchstens den Übergang von "lebendig" nach "tot" oder umgekehrt als vorübergehenden Fehler bezeichnen.) Als einen Fehler würde man das Vorhandensein von toten Nachbarn dann auffassen, wenn gefordert wäre, daß die Menge *neighbours* immer gleich der Menge N der physikalischen Nachbarn sein muß.

Die Voraussetzungen, daß auf jeder Kante des Kommunikationsgraphen maximal eine Nachricht unterwegs sein kann, ist durch die bereits in Fußnote 1 auf Seite 87 erwähnten *UCDL*s implementierbar [17]. Der Algorithmus wird jedoch auch funktionieren, wenn man voraussetzt, daß nur "wenige" Nachrichten gleichzeitig unterwegs sein können. Um dies zu sehen, kann man für die Kommunikation zwischen zwei Knoten das Bild einer Partie Tennis heranziehen: Die Knoten entsprechen den Tennisspielern und eine Nachricht einem Ball. Ist kein Ball im Spiel (d.h. keine Nachricht unterwegs) schlägt einer von beiden (oder beide gleichzeitig) durch die Anweisung mit der Wache **timeout** auf. Wird eine Nachricht empfangen, so wird sie einfach "zurückgeschlagen." Bei einem Anfangs leeren Kanal werden so maximal zwei Nachrichten unterwegs sein (wenn beide "Spieler" ungefähr gleichzeitig aufschlagen). Der einzige Zustand, der nicht entreten darf, besteht aus unendlich vielen Bällen, die im Spiel sind. Dies kann wegen der Forderung nach Selbststabilisierung durchaus ein Startzustand sein (es soll ja jeder sein dürfen). Darum muß man (wie bei vielen Problemen in diesem Zusammenhang [52]) eine Schranke auf der maximalen Anzahl von Nachrichten in einem Kanal annehmen.

Der Algorithmus reagiert im übrigen sehr eigenwillig, wenn eine der Korrektheitsbedingungen verletzt wird: Wenn beispielsweise ein Kanal Nachrichten nur in einer Richtung durchläßt, wird sich der Algorithmus nicht selbst stabilisieren. Die geforderten Eigenschaften der $\mathbf{timeout}_j$-Anweisung sind ebenfalls schwierig zu erfüllen und nur durch einen "aus-

process *maintain neighbours at node i*
  const $N = \{n_1, n_2, \ldots, n_k\}$ {* physical neighbours *}
  var *neighbours* **init** $\emptyset$ {* up-neighbours *}
  $j \in N$
  begin
    $j \in N \wedge \textbf{timeout}_j \longrightarrow neighbours := neighbours \setminus \{j\};\ \textbf{out}(j, i)$
  $[\!]$ $\textbf{in}(i, j) \longrightarrow neighbours := neighbours \cup \{j\};\ \textbf{out}(j, i)$
  end

Abbildung 5.6: Pflege der Nachbarschaftsbeziehungen im Nachrichtenmodell. Die Nachricht (r, s) ist eine leere Nachricht von Knoten s an Knoten r.

reichend großen" *timeout*-Wert implementierbar.

Schicht 2: Konstruktion des aufspannenden Baumes. Zur Konstruktion des aufspannenden Baumes wird der Algorithmus 4.10 von Chen, Yu und Huang ins Nachrichtenmodell überführt. Als erstes müssen hierzu auf jedem Knoten zusätzliche Variablen $\tilde{l}_j$ eingeführt werden, welche die momentane "Sicht" der Kommunikationsvariablen der Nachbarn enthalten. Anschließend muß sichergestellt werden, daß bei jeder Änderung der eigenen Werte die eigenen Nachbarn eine Kopie zur Aktualisierung ihrer "Sicht" erhalten. Dabei ist die Anweisung **shout**(l) eine abkürzende Schreibweise für die Tatsache, daß ein Prozeß i den Wert seiner lokalen Variable l an alle (lebenden) Nachbarn schickt [136, S. 492], also für:

  forall $j \in neighbours$ **do** $\textbf{out}(j, i, l)$

Der erste und zweite Parameter des **out**-Kommandos bezeichnet dabei wieder den Empfängerknoten, bzw. den Sendeknoten.

Der lokale Algorithmus eines Nicht-Wurzelknotens zur Spannbaumkonstruktion im Nachrichtenmodell ist in Abbildung 5.7 dargestellt. Man beachte, daß die Anweisung **timeout** → **shout**(l) hinzugefügt wurde, um einerseits die Anfangssymmetrie zu brechen und andererseits mit verlorenen Nachrichten umzugehen. Die Wurzel selbst hat die Nummer 0 und ist durch konstante Werte $l_0 = 0$ sowie $p_0 = 0$ gekennzeichnet. Sie enthält lediglich die beiden Zeilen:

  begin
    $\textbf{timeout} \longrightarrow \textbf{shout}(0)$
  $[\!]$ $\textbf{in}(0, j, k) \longrightarrow \textbf{shout}(0)$
  end

Die Korrektheit des Algorithmus aus Abbildung 5.7 folgt aus dem folgenden Satz. Der Beweis ist wieder in den Anhang (Abschnitt A.4) verlegt worden.

process *maintain spanning tree at node i*
  const n **init** {* total number of nodes *}
  i **init** {* own node id *}
  var *neighbours* {* set of neighbour node ids *}
  $l \in \{1, \ldots, n\}$ {* level in tree *}
  $p \in \{1, \ldots, n\}$ {* parent in tree *}
  $\forall j \in neighbours : \tilde{l}[j] \in \{1, \ldots, n\}$ {* view of level of neighbours *}
  $k \in \mathbb{N}$
  begin
    $l \neq n \wedge l \neq \tilde{l}[p] + 1 \wedge \tilde{l}[p] \neq n \longrightarrow l := \tilde{l}[p] + 1;$ **shout**(l)
  ⫿ $l \neq n \wedge \tilde{l}[p] = n \longrightarrow l := n;$ **shout**(l)
  ⫿ $m \in neighbours \wedge l = n \wedge \tilde{l}[m] < n - 1 \longrightarrow l := \tilde{l}[m] + 1;\ p := m;$ **shout**(l)
  ⫿ **timeout** $\longrightarrow$ **shout**(l)
  ⫿ **in**$(i, j, k) \longrightarrow \tilde{l}[j] := k$
  end

Abbildung 5.7: Lokaler Algorithmus einer Nicht-Wurzel zur Berechnung eines aufspannenden Baumes im Nachrichtenmodell.

5.10 Satz (Korrektheit des Algorithmus zur Berechnung eines aufspannenden Baumes) Die Kommunikationskanäle seien derart, daß Nachrichten nur eine maximale endliche Verzögerung haben. Die Menge der lebenden Nachbarn *neighbours* sei zwar anfangs variabel doch ab einem gewissen Zeitpunkt fest. Die Anweisung **timeout** sei genau dann wahr, wenn keine Nachrichten an Knoten i von seinen Nachbarn unterwegs sind. Es gelte schwache Fairneß.

Unter diesen Voraussetzungen stabilisiert sich der Algorithmus aus Abbildung 5.7 zum folgenden Prädikat P:

$$
\begin{aligned}
P \equiv\ & \forall i \in Nodes : \\
& (i = 0 \Rightarrow (p_i = 0 \wedge l_i = 0)) \wedge \\
& (i \neq 0 \Rightarrow (\exists j \in neighbours_i : p_i = j \wedge l_j = l_i + 1)) \wedge \\
& (\forall j \in neighbours_i : \tilde{l}_j[i] = l_i) \wedge \\
& (\forall \langle i, j, k \rangle \in M : k = l_j)
\end{aligned}
$$

Die Menge M ist dabei die Menge aller Nachrichten, die unterwegs sind. Eine Nachricht von Knoten j an Knoten i mit Inhalt k wird in der Form $\langle i, j, k \rangle$ geschrieben.

Die ersten beiden konjunktiven Terme von P spiegeln die Existenz des Spannbaumes wieder. Die dritte Bedingung besagt, daß die “Sicht” eines Knotens auf seinen Nachbarn “korrekt” sein muß und der letzte Term fordert, daß nur “richtige” Nachrichten unterwegs sein dürfen.

5.11 **Bemerkung** (Implementierung des Algorithmus für Schicht 2) Die Simulation des Algorithmus ergab folgende einfache Optimierungen, die eine sprunghafte Vermehrung von Nachrichten im System (*message implosion* [105]) verhindern. Zum einen kann im lokalen Algorithmus der Wurzel die zweite Anweisung $\mathbf{shout}(0)$ mit der Wache $\mathbf{in}(0, j, l)$ durch eine "leere" Anweisung **skip** ersetzt werden. So werden Nachrichten einfach "verschluckt" und aus dem Verkehr gezogen. Die erste Zeile mit der **timeout**-Wache sorgt dafür, daß im Falle leerer Datenkanäle wieder Nachrichten per **shout** ins System gegeben werden.

Zum anderen benötigt man im Algorithmus einer Nicht-Wurzel eine Möglichkeit, um auf das Ausscheiden des Elternknotens aus der *neighbours*-Menge zu reagieren. Eine Zeile der Form

$$p \notin \mathit{neighbours} \quad \longrightarrow \quad l := n;\ \mathbf{shout}(l)$$

wäre dafür ausreichend.

Wird in beiden Fällen die **timeout**-Anweisung durch einen Zähler implementiert, so darf dieser bei Empfang einer Nachricht nicht zurückgesetzt werden, da ansonsten die Gefahr bestünde, daß einige Kanäle dauerhaft keine Nachricht enthielten.

Schicht 3: selbststabilisierende Replikation. Um den einfachen Replikationsalgorithmus ins Nachrichtenmodell zu überführen, werden dieselben Transformationsschritte vorgenommen, wie im vorhergehenden Fall des Spannbaumalgorithmus': Zusätzliche lokale Variablen müssen eingeführt werden, welche die momentane "Sicht" der Replikationsvariable des Elternknotens enthält. Da sich durch Änderungen in der Kommunikationstopologie der Elternknoten durchaus ändern kann, müssen jeweils alle Variablen r der physikalischen Nachbarn vorgehalten werden. Ansonsten sind die Änderungen analog zur Schicht, die den aufspannenden Baum konstruiert. Die Algorithmen für den Wurzelknoten und die anderen Prozesse sind in den Abbildungen 5.8 und 5.9 dargestellt.

Ein Beweis der Korrektheit orientiert sich an der Konstruktionsmethode aus Abschnitt 5.2.2 und benutzt das Kompositionstheorem 4.22. Da er keine neuen Einsichten liefert, wird er hier nicht explizit geführt.

5.12 **Bemerkung** (Implementierung des Algorithmus für Schicht 3) In Anlehnung von Bemerkung 5.11 sollte bei einer Implementierung die Wache $\mathbf{in}(0, j, k)$ die Anweisung **skip** ermöglichen (und nicht $\mathbf{shout}(0)$). Auch darf bei Implementierung der **timeout**-Wache durch einen Zähler dieser bei Empfang einer Nachricht nicht zurückgesetzt werden.

Bewertung und Analogien. Der zusammengesetzte Algorithmus, der durch die Überlagerung der drei vorgenannten Verfahren entsteht, stabilisiert sich selbst zu wechselseitiger Konsistenz. Er toleriert dabei jede Form vorübergehender Fehler und erlaubt sogar, daß einzelne Kanten des Kommunikationsnetzwerkes dauerhaft ausfallen, solange der resultierende Graph noch verbunden ist. Hierbei wird nicht die Tatsache des Nicht-Funktionierens als Fehler angesehen sondern der Wechsel zwischen den

process *replication root*
  const i **init** 0 {* root id is 0 *}
  var *neighbours* {* set of neighbours *}
  $r \in Data$ {* primary copy *}
  begin
    timeout $\longrightarrow$ **shout**(r)
  $[\!]$ **in**$(0, j, k)$ $\longrightarrow$ **shout**(r)
  $[\!]$ {* update wanted *} $\longrightarrow$ $r :=$ {* new value *}; **shout**(r)
  end

Abbildung 5.8: Lokaler Algorithmus des Wurzelknotens zur Replikation mit einer Primärkopie im Nachrichtenmodell.

process *replication node i*
  const i **init** {* own node id *}
  var *neighbours* {* set of neighbour ids *}
  $p \in \{1, \ldots, n\}$ {* parent in tree *}
  $r \in Data$ {* own replication data *}
  $\forall j \in neighbours : \tilde{r}[j] \in Data$ {* view of replication data *}
  begin
    $r \neq \tilde{r}[p]$ $\longrightarrow$ $r := \tilde{r}[p]$; **shout**(r)
  $[\!]$ **timeout** $\longrightarrow$ **shout**(r)
  $[\!]$ **in**(i, j, k) $\longrightarrow$ $\tilde{r}[j] := k$
  end

Abbildung 5.9: Lokaler Algorithmus eines Nicht-Wurzelknotens zur Replikation mit einer Primärkopie im Nachrichtenmodell.

beiden Funktionszuständen. Durch diese Konstruktionsmerkmale ist er robuster als die in Kapitel 3 vorgestellten Algorithmen wie *TSAE* oder *HPP*, obwohl er natürlich anderes leistet.

Bei einer Implementierung ist darauf zu achten, daß sich die einzelnen Schichten nicht gegenseitig behindern. Das bedeutet insbesondere, daß die Nachrichten der jeweiligen Schichten getrennt betrachtet werden müssen (andere Schichten dürfen sie nicht "sehen"). Natürlich können trotz dieser Bedingung Nachrichten unterschiedlicher Schichten gemeinsam befördert werden, wenn das unterliegende Kommunikationssystem derartige Mechanismen zur Verfügung stellt.

Bei der in diesem Abschnitt vorgestellten Realisierung im Nachrichtenmodell sind schon deutliche Ähnlichkeiten zu *TSAE* und *HPP* zu beobachten. Vor allem das Empfangen und Weiterleiten von "Replikationsnachrichten" aus den Abbildungen 5.8 und 5.9 im Baum erinnert an den entsprechenden Mechanismus aus *HPP*, obwohl dort natürlich Nachrichten von beliebigen Knoten aus losgeschickt werden können. Die Fähigkeit zur automatischen Restrukturierung der Kommunikationstopologie ist jedoch nur dem hier vorgestellten Algorithmus eigen.

5.2.6 Alternative Formalisierung von Replikation

Bei der Formalisierung von Replikation und dem daraus resultierenden Algorithmus entsteht also ein Verfahren mit einer Primärkopie. Nur der Knoten mit der Primärkopie darf den Replikationswert schreiben; alle anderen Stationen sind mehr oder weniger passiv. Wie kann man es erlauben, daß alle Prozesse schreiben dürfen? Als ersten Ansatz könnte man die Ideen aus diesem Abschnitt weiterführen und fordern, daß sich jeweils der "neuste" Wert, den ein Prozeß geschrieben hat, auf allen anderen Stationen durchsetzt.

Durch das Fehlen einer gemeinsamen Zeit unterscheidet sich der technische Zeitbegriff in verteilten Systemen vom umgangssprachlichen deutlich. In den meisten Modellen verteilter Berechnungen wird Zeit üblicherweise auf die Messung potentieller kausaler Beeinflussung reduziert [101]. In dieser "logischen Zeit" ordnet man Ereignissen des Systems Zeitstempeln zu und macht sie damit vergleichbar. Die Zuordnung geschieht über eine Abbildung $C : E \rightarrow T$ von der Menge der Ereignisse in eine Menge T (den "Zeitbereich"), auf dem eine totale Ordnung oder eine Halbordnung definiert ist [101, S. 559]. Diese Ordnung wird bezugnehmend auf Lamport als *happened before*-Relation bezeichnet [101, S. 559]. Betrachtet man zwei Ereignisse e_1 und e_2 einer Berechnung, so kann man — vereinfacht gesagt — durch Bestimmung der Zeitstempel feststellen, ob e_1 vor e_2 stattfand ($C(e_1) < C(e_2)$) oder nicht.

Diese Überlegungen führen zu einer veränderten Formulierung des Konsistenzproblems, welches im folgenden weiter untersucht werden soll.

5.13 Definition (erweitertes Konsistenzproblem) Gegeben sei ein System mit n unabhängigen Prozessen p_1 bis p_n. Jeder einzelne Prozeß enthalte zwei lokale Tupel $(r_i, C(r_i))$ und $(w_i, C(w_i))$ vom Typ $Data \times T$. Dabei sei $C : \{w_i, r_i\} \rightarrow T$ eine Funktion, welche dem Datenwert in r_i, bzw. w_i den Zeitstempel des letzten Änderungsereignisses aus einer total geordneten Menge $(T, <)$ zuordnet. Die Variable r_i wird *Replikationsvariable* und w_i wird *update-Variable* genannt.

Das System ist in einem erweitert konsistenten Zustand, gdw. das folgende Prädikat P gilt:

$$P \equiv \forall i, j \in \{1, \ldots, n\} : r_i = r_j \wedge C(r_i) \geq C(w_i)$$

Das *erweiterte Konsistenzproblem* ist die Aufgabe, eine Konfiguration, in der $\neg P$ gilt, in eine Folgekonfiguration zu überführen, in der P gilt.

Bemerkungen zu Definition 5.13. Die Replikationsvariable r_i ist die momentan von Prozeß p_i bereitgehaltene Version des Replikationswertes.

Die *update*-Variable w_i ist eine Hilfskonstruktion: In der Spezifikation muß ausgedrückt werden, daß am Ende r_i "neuer" ist als alle Werte, die zuletzt von beliebigen Prozessen geschrieben wurden. Die Variable w_i wird nur dazu benutzt, um schreibend auf r_i einzuwirken und enthält immer diesen letzten Schreibwert. Bei konkreten Realisierungen kommt man zwar auch ohne diesen zusätzlichen Speicherplatz aus, doch ließe sich das Problem nicht mehr in dieser Form spezifizieren, und die Spezifikation ist ja hier der Ausgangspunkt für den Algorithmenentwurf.

Will ein Prozeß p_i die Replikationsvariable aktualisieren, muß er den Wert von w_i ändern. Falls $C(w_i) > C(r_i)$, ist der Wert von w_i "neuer" als die momentan vorgehaltene Version des Replikationswertes und das System ist nicht mehr in einem erweitert konsistenten Zustand. Ein Algorithmus muß nun dafür sorgen, daß sich derjenige Wert von w_i mit dem höchsten Zeitstempel im gesamten System ausbreitet und alle Replikationsvariablen r_i besetzt (vergleiche hierzu auch Beispiel 5.14 und Abbildung 5.10 auf Seite 140).

5.2.7 Umsetzung des erweiterten Konsistenzproblems

Durch die Ähnlichkeiten zwischen Definitionen 5.13 und 5.1 ist es unmittelbar einsichtig, daß man auch das erweiterte Konsistenzproblem in selbststabilisierender Art und Weise lösen kann. Man benutzt den einfachen Replikationsalgorithmus 5.4 von Seite 124 und fordert, daß sich die Replikationsvariablen immer in Richtung des neusten Wertes hin korrigieren, etwa in der folgenden Art und Weise:

$$m \in \mathit{neighbours} \wedge r_m \neq r \wedge C(r_m) > C(r) \quad \longrightarrow \quad (r, C(r)) := (r_m, C(r_m))$$

Ein Prozeß i, der den Replikationswert schreibend verändern will, muß dann lediglich w_i und r_i auf diesen Wert setzen und mit dem jeweiligen Zeitstempel versehen.

Korrektheit. Eine wesentliche Voraussetzung für die Korrektheit des eben angesprochenen Verfahrens ist, daß es einen eindeutig "neusten" Wert im System gibt. Gäbe es mehrere unterschiedliche Werte r_i mit demselben "neusten" Zeitstempel, dann könnte es zu unendlichen Folgen von Aktualisierungsaktionen kommen. Die Konvergenz des Systems ist dann nicht mehr garantiert und damit Selbststabilisierung nicht nachweisbar. Eine eindeutige totale Ordnung auf der Menge T ist also eine notwendige Voraussetzung.

Verhalten bei Partitionen. Der Algorithmus weist ein interessantes Verhalten bei Netzpartitionierungen auf. Da es nun keinen dominanten Wert einer ausgezeichneten Station mehr gibt, werden sich Teilnetze, die durch den Ausfall von Verbindungsleitungen oder -knoten entstehen, eigenständig weiterentwickeln. Zwar ist das Gesamtnetz dann nicht mehr in einem erweitert konsistenten Zustand, doch sind es die Teilnetze schon. Bei der Realisierung mit einer Primärstation arbeitet nur die Partition korrekt, welche diesen ausgezeichneten Knoten enthält. Alle anderen sind blockiert.

Probleme des Verfahrens. Es ist offensichtlich, daß bei seltenen Änderungen der Replikationsvariable alle Prozesse auch die neuen Werte mitbekommen. Das Schreiben darf dann aber erst dann stattfinden, wenn sich das System wieder in einem erweitert konsistenten Zustand befindet. Wenn hingegen mehrere Prozesse wie im folgenden Beispiel oftmals und praktisch gleichzeitig schreibend auf die Replikationsvariable einwirken, dann kann es sein, daß einzelne Prozesse nichts von den zeitweiligen Änderungen mitkriegen und nur den jeweils aktuellen Wert sehen.

5.14 Beispiel Gegeben sei ein System mit drei Prozessen p_1, p_2 und p_3, die in Form einer linearen Kette angeordnet sind und auf denen der oben skizzierte Algorithmus zur Lösung des erweiterten Konsistenzproblems abläuft. Zeitstempel seien durch natürliche Zahlen implementiert. Das System sei in einem erweitert konsistenten Zustand, d.h. es gelte $(r, C(r)) = (a, 5)$ auf allen Knoten und $C(w)$ sei überall kleiner als 5.

In einem ersten Schritt ändert p_1 seine Werte $(w_1, C(w_1))$ und $(r_1, C(r_1))$ auf $(b, 7)$. Gleichzeitig (oder kurze Zeit später) ändert Prozeß p_2 die seinigen auf $(c, 10)$. Nun hat p_2 den neusten Wert im System und durch den Algorithmus werden p_1 und p_3 diesen Wert in ihre Replikationsvariable übernehmen. Die vorherige Änderung durch p_1 bleibt für Prozeß p_3 unsichtbar.

Mit diesem Verfahren kann man also keinen zuverlässigen *broadcast* realisieren und somit ist die Modifikation des einfachen Algorithmus von einem auf mehrere "Sender" in gewisser Weise mißglückt. Abschnitt 5.3 wird sich weiter mit diesem Thema beschäftigen.

Implementierungsaspekte der Zeitstempel. Weiter oben wurde bereits angesprochen, daß eine totale Ordnung notwendig für das korrekte Funktionieren des Algorithmus ist. Zur Implementierung einer totalen Ordnung auf den Ereignissen des Systems gibt es mehrere Ansätze. Das einfachste Ordnungskriterium besteht in einer Durchnumerierung der Ereignisse pro Prozeß und der zusätzlichen Identität des Knotens als letztes ausschlaggebendes Ordnungsmerkmal (den aus dem Tennis bekannten *tie breaker*). Bei einer solchen Realisierung werden aber immer Prozesse bevorzugt, "auf denen viel passiert" und deren Ereignisse darum meist höhere Zeitstempel haben als alle anderen. Eine bessere Lösung ist darum, die logische Lamport-Zeit [101] zu verwenden, die dieselbe Technik benutzt aber zusätzlich noch die Kausalrelation beachtet. Aber auch hier wird letztendlich die Prozeßidentität als *tie breaker* benötigt, was dazu führt, daß einige Prozesse (je nachdem mit hoher oder niedriger Ordnungsnum-

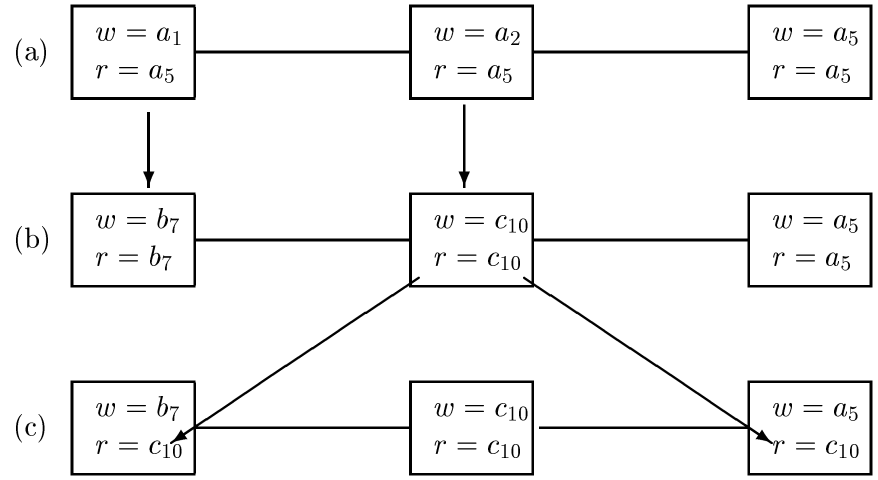

Abbildung 5.10: Ablauf des Algorithmus zur Lösung des erweiterten Konsistenzproblems aus Beispiel 5.14.

mer) durch ihre Identität bevorzugt werden. Wenn i der Knoten mit der "neusten" Ordnungsnummer ist, werden sich im Zweifelsfall immer Nachrichten von i im System verteilen. Diese Tendenz geht bereits wieder in Richtung einer Primärkopie, welches ja bei diesem Ansatz vermieden werden sollte.

Ein "gerechterer" Ansatz ist darum, eine Zeit zu verwenden, die zwar Kausalität respektiert, aber nicht von den lokalen Prozeßidentitäten abhängig ist. Als allgemeinste Form kommt hierfür die Vektorzeit [112, 127] in Frage. Doch der Zeitbereich, aus denen die Zeitstempel genommen werden, ist der n-dimensionale Vektorraum und auf diesem besteht nur eine Halbordnung, die auch nicht ohne zusätzliche Entscheidungskriterien (wie die Prozeßidentität oder probabilistische Verfahren) jeweils eindeutig zu einer totalen Ordnung umgemünzt werden kann. Es bleibt also die Frage, wie man die "unvergleichbaren" Fälle behandelt.

In der Literatur gibt es einen Ansatz, den man als "optimistisch" bezeichnet [25, S. 8]: Das System vertraut darauf, daß nicht allzu viele Schreiboperationen gleichzeitig stattfinden und stellt Konfliktauflösungsmechanismen zur Verfügung, die in solchen (hoffentlich seltenen) Fällen die Konsistenz wieder herstellen. Dies ist in verteilten Datenbanken vor allem dann notwendig, wenn eine verteilte Datenbank durch eine Partitionierung insgesamt in einen inkonsistenten (d.h. nicht mehr 1-Kopieserialisierbaren) Zustand gekommen ist und die Partitionen wieder zusammengeführt werden müssen.

Man kann natürlich bei der Verwendung von Vektorzeit auch versuchen, durch eine geschickte Verzahnung von Lese- und Schreibzugriffen möglichst immer eine Kausalkette zwischen den einzelnen Schreibereignissen herzustellen, beispielsweise indem man ein *token* zirkulieren läßt. Interessant sind hier vor allem die verteilten Lösungen wie beispielsweise die “Informationsstrukturen” von Beverly Sanders [124]. Eine genaue Betrachtung dieser Ansätze an dieser Stelle würde aber zu weit führen, da ich mich ansonsten zu weit vom eigentlichen Thema dieser Arbeit fortbewegen würde (welches ja nicht “Wechselseitiger Ausschluß” ist). Bei der Implementierung von Vektorzeit gibt es auch die bekannten Probleme [37, 129], mit denen man in der Praxis klarkommen muß.

5.3 Ein fehlertoleranter *broadcast*

In Abschnitt 5.2.6 wurde durch die Neuformulierung des Konsistenzproblems versucht, einen Replikationsalgorithmus zu entwickeln, bei dem alle Prozesse schreibend auf die Replikationsvariable zugreifen dürfen. Da in Abschnitt 5.2.2 zudem ein “dominanter” Wert innerhalb des Replikationsalgorithmus festgestellt wurde, hätte man beinahe erwarten können, daß sich aus der Verschmelzung beider Ansätze ein zuverlässiger *broadcast* ergeben würde. Leider wurden diese Erwartungen durch das Beispiel 5.14 enttäuscht. In diesem Abschnitt soll ein anderer Weg beschritten werden.

Spezifikation eines selbststabilisierenden zuverlässigen *broadcast*. Zunächst muß geklärt werden, was ein selbststabilisierender zuverlässiger *broadcast* ist. In Definition 3.5 wurde bereits beschrieben, welche drei Bedingungen ein Algorithmus erfüllen muß, um ein zuverlässiger *broadcast* im herkömmlichen Sinne zu sein:

1. Alle funktionierenden Prozesse müssen dieselbe Menge an *broadcast*-Nachrichten zustellen,
2. diese Nachrichten müssen alle von funktionierenden Prozessen verschickt worden sein und
3. die Nachrichten werden maximal ein Mal zugestellt.

In Verbindung mit der Selbststabilisierung möchte ich wieder annehmen, daß Kommunikationskanäle auch dauerhaft funktionsunfähig sein können. Das Vorhandensein solcher Kanäle soll darum nicht als vorübergehender Fehler angesehen werden, sondern lediglich der Wechsel von “funktionsfähig” nach “funktionsunfähig” und umgekehrt. Dies ist gerechtfertigt durch die Annahme einer bereits spezifizierten Basisschicht, die die Nachbarschaftsrelation aktuell hält. Voraussetzen muß man allerdings wieder, daß der nach Wegnahme der funktionsunfähigen Kommunikationsleitungen verbleibende Kommunikationsgraph noch verbunden ist. (Wenn man möchte, kann man die bei der Spezifikation des herkömmlichen zuverlässigen *broadcast* benutzte Bezeichnung der “korrekten Prozesse” (*correct processes*) [80, S. 100] auf genau jene Knoten beziehen, die im möglicherweise partitionierten Netzwerk noch erreichbar sind.) Für den selbststabilisierenden zuverlässigen *broadcast* genügt es darum zu fordern, daß nach dem Aufhören vorübergehender Fehler der *broadcast* nach endlicher Zeit wieder korrekt funktioniert.

Das "Problem" bei der genauen Spezifikation eines selbststabilisierenden *broadcast* liegt darin, daß man hier zu einer *liveness*-Eigenschaft hin stabilisieren will, nämlich zu genau den Ausführungen, die die korrekte Ausführung des *broadcast* noch möglich machen (vergleiche Definition 2.18 auf Seite 33). Man muß darum hier die alternative Definition der Selbstatsbilisierung von Tel (Definition 4.5 auf Seite 75) verwenden.

Idee hinter einem *broadcast*-Algorithmus. Die Idee bei diesem Ansatz ist, den oben vorgestellten Replikationsalgorithmus für dynamische Kommunikationstopologien (Algorithmus 5.8) einfach n-fach zu überlagern. Das bedeutet insbesondere:

- Jeder Prozeß ist die Wurzel eines eigenen aufspannenden Baumes.
- Jeder Prozeß besitzt darum auch n verschiedene Replikationsvariablen (eine pro Wurzel).
- Jeder Prozeß besitzt n Verweise auf (möglicherweise unterschiedliche) Elternknoten.

Die lokalen Variablen der Prozesse sind entsprechend n-dimensionale Vektoren: ein "Elternvektor" $p[1 \dots n]$ und ein "Datenvektor" $r[1 \dots n]$. Die parallele Ausführung der n Algorithmen führt zu folgendem Algorithmus:

5.15 **Algorithmus** (zuverlässiger *broadcast*) Gegeben sei ein System mit n Prozessen. Der Prozeß i habe jeweils einen Elternvektor $p_i[1 \dots n]$ und einen Datenvektor $r_i[1 \dots n]$. Ein Prozeß prüft periodisch für jede Komponente j seines Elternvektors mit $i \neq j$ nach, ob sich der gleiche Eintrag bei Prozeß $p[j]$ von dem eigenen unterscheidet. Falls ja, wird er diesen Wert lesen und ihn an die j-te Stelle seines eigenen Datenvektors schreiben. Sobald sich eine Komponente des Datenvektors geändert hat, entspricht dies einer neu eingegangenen *broadcast*-Nachricht. Diese kann dann durch den Aufruf des **deliver**-Primitives an die Anwendungsschicht ausgeliefert werden. Das **broadcast**-Primitiv wird dadurch realisiert, daß ein Knoten i den "eigenen" Eintrag $r[i]$ des Datenvektors auf die zu versendende Nachricht setzt. Der entsprechende Algorithmus ist in Abbildung 5.11 dargestellt.

Korrektheit von Algorithmus 5.15. Natürlich muß gezeigt werden, daß Algorithmus 5.15 tatsächlich ein selbststabilisierender zuverlässiger *broadcast* ist. Hierzu muß man eine Menge L von "legalen" Konfigurationen angeben, für die die beiden Bedingungen aus Definition 4.5 gelten müssen:

1. Korrektheit: Jede Ausführung, die in L beginnt, ist ein zuverlässiger *broadcast* gemäß Definition 3.5.

process *broadcast node i*
  const $p[1 \dots n]$ **init** {* parent vector *}
  var $r[1 \dots n] \in \mathit{Data}$ {* data vector *}
   $j \in \mathbb{N}$
   m {* broadcast message *}
  begin
   $j \in \{1, \dots, n\} \wedge i \neq j \wedge r[j] \neq r_{p[j]}[j] \longrightarrow r[j] := r_{p[j]}[j]$; **deliver**$(r[j])$
   {* implementation of **broadcast**(m): *}
  $[\!]$ {* m is new broadcast message *} $\longrightarrow r[i] := m$; **deliver**$(m)$
  end

Abbildung 5.11: Lokaler Algorithmus für einen selbststabilisierenden zuverlässigen *broadcast* (Algorithmus 5.15).

2. Konvergenz: Jede Ausführung enthält eine Konfiguration aus L.

Mit anderen Worten muß also zum einen gezeigt werden, daß — ausgehend von einem "legalen" Zustand aus L — die drei Bedingungen aus der Definition 3.5 von Seite 47 erfüllt sind; zum anderen benötigt man einen Nachweis darüber, daß das System aus einem beliebigen Zustand (d.h. nachdem vorübergehende Fehler vorübergegangen sind) nach endlicher Zeit garantiert in einen "legalen" Zustand übergeht. Dies ist in unserem Falle (unter gewissen Voraussetzungen) nicht schwierig, handelt es sich doch bei der Menge L gerade um diejenigen Konfigurationen des Gesamtsystems, in denen jeder der n Teilalgorithmen einen aktiv-konsistenten Zustand (vergleiche Lemma 5.6 von Seite 127) erreicht hat.

Voraussetzungen für die Korrektheit. Bevor der Satz, der die Korrektheit formal konstatiert, im einzelnen dargelegt wird, möchte ich zunächst intuitiv dessen Voraussetzungen herleiten.

Wichtig ist, daß sich die n überlagerten Algorithmen nicht gegenseitig behindern. Formal bedeutet dies, daß kein Algorithmus auf Einträge in den Vektoren schreibend zugreifen darf, die ihm "nicht gehören." Dies ist Voraussetzung für die Anwendbarkeit des Kompositionstheorems 4.22. Aus der Korrektheit des einfachen Replikationsalgorithmus' kann man dann folgern, daß sich (sobald alle Fehlerquellen versiegen) ein insgesamt aktiv-konsistenter Zustand einstellen wird.

Aus der Korrektheit der Teilalgorithmen (und ihren bereits nachgewiesenen Eigenschaften) kann man anschließend auch folgern, daß sich eine *broadcast*-Nachricht auf allen Prozessen verteilt, wenn ein Prozeß seine eigene Replikationsvariable ändert. Vorausgesetzt werden muß hier allerdings wieder, daß alle Nachbarn den entsprechenden alten Wert bereits übernommen haben, bevor man selbst eine Aktualisierung vornehmen kann.

Ein weiteres, aber vielleicht einfacher zu lösendes Problem ist der wiederholte Versand identischer *broadcast*-Nachrichten. Eine Zuweisung $r := r$ würde den Algorithmus nicht aus seiner Ruhelage erwecken, eine Nachricht ginge verloren.

5.16 Satz (Korrektheit von Algorithmus 5.15) Gegeben sei ein System mit n Prozessen, auf denen Algorithmus 5.15 abläuft. Ein Prozeß i ändere den Eintrag $r_i[k] = x$ seines Datenvektors nur dann auf einen neuen Wert y, wenn für alle Knoten $j \in \mathit{neighbours}$ mit $p_j[k] = i$ gilt: $r_j[k] = x$. Ein Knoten rufe außerdem niemals zweimal hintereinander **broadcast**(m) mit dem gleichen Wert m auf.

Unter diesen Voraussetzungen stabilisiert sich das System nach mindestens n Schreiboperationen jedes Knotens zur Eigenschaft des zuverlässigen *broadcast* aus Definition 3.5.

Der Beweis von Satz 5.16 ist in Anhang A.5 nachzulesen.

Anmerkungen zu Algorithmus 5.15. Die Einschränkung der Änderungsfrequenz in den Voraussetzungen zur Korrektheit erscheint auf den ersten Blick etwas stark und schwerlich zu implementieren. Im Wirklichkeit werden hier nur Vorbedingungen für einen erneuten Aufruf des **broadcast**-Primitives beschrieben. Diese Vorbedingungen sind im Modell mit gemeinsamen Speicher einfach abprüfbar. Im *message passing*-Modell ist dies nur durch das Versenden von Bestätigungsnachrichten (auf einer beliebigen Kommunikationsebene) erreichbar. Darum erfordert das Abprüfen dieser Bedingung im Nachrichtenmodell einen etwas größeren Implementierungsaufwand, dessen Realisierung zusätzlich auf Korrektheit überprüft werden muß.

Die Voraussetzung, daß niemals zweimal hintereinander identische Werte geschrieben werden dürfen, ist durch die Hinzufügung von Redundanz zur Nachricht implementierbar (beispielsweise durch beschränkte Sequenznummern).

Als Nebenprodukt der Einschränkungen auf der Änderungsfrequenz ist jedoch sichergestellt, daß sich zwei aufeinanderfolgende Aktualisierungen auf ihrem Ausbreitungsweg im Netzwerk nicht überholen und bei einem Knoten in der umgekehrten Reihenfolge eingehen können. Informal gesprochen ist dies dadurch garantiert, daß die Ausbreitung eines neuen Wertes pro Kanal und Knoten in *FIFO*-Reihenfolge geschieht und sich die einzelnen "Ausbreitungswellen" darum nicht überholen können.

In der Darstellung des Algorithmus in Abbildung 5.11 ist der Elternvektor als Konstante deklariert worden. Wie in den Ausführungen zum einfachen Replikationsverfahren aus Abschnitt 5.2.4 bereits beschrieben, kann man durch die "Vorschaltung" eines selbststabilisierenden Spannbaumalgorithmus' auch dynamischen Kommunikationstopologien entgegenkommen. Dies würde allerdings dann auch in einem n-dimensionalen "Stufenvektor" resultieren.

Interessant ist auch die Erkenntnis, daß die schwache Konsistenz, die vormals ein "Nebenprodukt" der Selbststabilisierung war (vergleiche Bemerkungen zu Algorithmus 5.4 auf Seite 124) nun der eigentliche "Motor" des realisierten *broadcast* ist. Da nun auf der obersten Schicht des Algorithmus nicht mehr primär zur wechselseitigen Konsistenz stabilisiert wird sondern aktiven Konsistenz und damit auch zur Spezifikation des zuverlässigen *broadcast*, sind die inkonsistenten Zustände nicht mehr "inkorrekt," sondern Zwischenzustände auf dem Weg, die spezifizierte *liveness*-Bedingung zu erfüllen.

Mögliche Erweiterungen. Nach Betrachtung der Eigenschaften des Algorithmus kann man sich über nachträgliche Erweiterungen und Optimierungen Gedanken machen. Eine mögliche Optimierung betrifft die Anzahl der Bäume und der durch sie verursachte Platzbedarf. Warum muß man n Bäume parallel aufrechterhalten, wenn möglicherweise ein einziger (wie in *HPP*) ausreicht? Eine Antwort darauf ergibt sich, wenn man betrachtet, was diese Optimierung an Speicherplatz pro Prozeß spart. Nimmt man also nur noch einen einzigen aufspannenden Baum im Netzwerk an, der durch Verweise auf die jeweiligen Nachbarknoten gegeben ist, injiziert man Symmetrie in das System. Dies wird deutlich, wenn man sich in die Situation eines Knotens versetzt, der die Änderung "seines" Replikationswertes bei einem Nachbarknoten feststellt. Dieser Knoten muß nun die Entscheidung treffen, ob er diesen Wert übernehmen soll oder nicht, d.h. er muß wissen, in welcher Richtung des Baumes die Wurzel liegt. Übernähme er einfach diesen Wert ohne weitere Prüfung, ist die Zustellung einer *broadcast*-Nachricht m nicht gewährleistet, da die "Ausbreitungswellen" von m durch alte Werte in den Blättern des Baumes schnell zurückgedrängt werden könnten. Man muß also pro Baum die "Richtung zur Wurzel" in Form einer Variable vorhalten. Dies ist aber genau der Elternvektor, der oben bereits beschrieben wurde. Man hätte dann n gleichartige Bäume überlagert, die alle eine andere Wurzel haben. Wie der Algorithmus zur Berechnung des aufspannenden Baumes zu diesem Zweck abgeändert werden muß oder kann, soll hier nicht weiter besprochen werden, da man hier höchstens die Speicherkomplexität des "Stufenvektors" reduzieren könnte. Dies wäre im Hinblick auf die gesamte Speicherkomplexität unrelevant, die weiter bei $O(n)$ bleibt. (Drastische Kostenersparnisse in der Praxis sind jedoch nicht ausgeschlossen.) Außerdem kann man fragen, ob n unterschiedliche Bäume, die möglichst verschiedene Kommunikationskanäle benutzen, nicht doch die Kommunikationstopologie besser ausnutzen und die Last besser auf die einzelnen Leitungen aufteilen.

Im Hinblick auf den letztgenannten Einwand bietet sich eine Variante des Algorithmus an, die versucht, die durch die unterschiedlichen Bäume benutzten Kanten möglichst disjunkt zu halten. In einem vollständig verbundenen Netzwerk würde ein derartiger Algorithmus zu einem n-dimensionalen Würfel (*hypecube*) führen, der wenigstens die Last auf den einzelnen Kanten minimiert.

Zum Abschluß der Besprechung der Erweiterungen muß nochmals wiederholt werden, daß man mittels der durch Hadzilacos und Toueg [80] beschriebenen (und in Kapitel 3 erwähnten) *broadcast*-Transformationen aus dem hier entwickelten "Basis-*broadcast*" alle weiteren Ausprägungen dieses Kommunikationsprimitives in strukturierter Weise herleiten kann. Die Eigenschaft der Selbststabilisierung muß jedoch im Einzelfall jeweils neu überprüft werden.

Bewertungen und Analogien. Ohne eine genaue Komplexitätsabschätzung zu machen, kann man die Speicher- und Nachrichtenkomplexität des Algorithmus kritisieren, die vermutlich in der Größenordnung $O(n)$ (Speicher) bzw. $O(n^2)$ (Nachrichten zur Herstellung einer stabilen Konfiguration) und $O(n)$ (pro Aufruf von **broadcast**) liegen. Auf der anderen Seite toleriert der Algorithmus jede Form vorübergehender Fehler und kommt mit dynamischen Änderungen in der Kommunikationstopologie zurecht.

Vergleicht man diesen Algorithmus mit *TSAE* fallen im wesentlichen zwei Punkte auf:

- Da dynamischen Änderungen als "Fehler" interpretiert werden, muß der *broadcast* in solchen Fällen laut Spezifikation (und im Gegensatz zu *TSAE*) nicht zu 100 Prozent korrekt funktionieren. Dies liegt an der Kommunikationsstruktur "Baum," die sehr fragil gegenüber Kanalausfällen ist. In *TSAE* wird die 100-prozentige Korrektheit erreicht durch die Forderung einer k-fach verbundenen Kommunikationstopologie, über die durch das Fluten Nachrichten auch im schlimmsten Fall (nur noch 1-fach verbunden) bei allen Kommunikationspartnern ankommen. Der Baum spielt im hier entwickelten Algorithmus also zwei Rollen: Zum einen ist der der Garant der Selbststabilisierung, zum anderen auch der verwundbarste Punkt des Algorithmus.

- In *TSAE* kann man (im Gegensatz zu dem hier entwickelten Algorithmus) Nachrichten in nahezu beliebiger Stückzahl und Geschwindkeit versenden. Bei dem hier entwickelten Replikationsalgorithmus ist die Einschränkung der Änderungsfrequenz eine wesentliche Voraussetzung der Korrektheit. Sie resultiert aus den im Gegensatz zu *TSAE* beschränkten "Puffergröße" der einzelnen Knoten (die Puffergröße ist 1). Zwar kann man sich auch hier vorstellen, die Anzahl der durch Knoten gepufferten Werte zu erhöhen, doch muß diese Größe zur Wahrung der Selbststabilisierungeigenschaft immer beschränkt sein. Bei beliebigen Puffergrößen wäre es nämlich nie möglich zu wissen, wann "falsche" Werte aus dem System verdrängt worden sind. Hier zeigt sich also wieder: Fehlertoleranz wird durch Leistungseinschränkungen erkauft.

- In diesem Abschnitt wurden bisher nur die Eigenschaften eines *broadcast* betrachtet. Für einen echten Vergleich mit *TSAE*, welches ja einen *multicast* realisiert, müßte man noch die Gruppenkontrollmechanismen entwickeln und näher untersuchen.

Diese Beobachtungen zeigen, daß durch die konsequente Berücksichtigung der Selbststabilisierungseigenschaft im Prinzip ein "beschränktes" (und damit selbststabilisierendes) *TSAE*-Protokoll entwickelt wurde. Die eingangs erwähnte zweite Vorgehensweise, also die nachträgliche Optimierung von *TSAE*, wäre vermutlich zum selben Ergebnis gekommen. Wenn unterschiedliche Vorgehensweisen von unterschiedlichen Startpunkten aus zu denselben Ergebnissen kommen, stecken oft tiefere Zusammenhänge hinter den verwendeten Methoden, als man annimmt.

Betrachtet man die oben besprochene Erweiterung, die nur noch einen Kommunikationsbaum vorsieht, treten offenbar auch oberflächliche Ähnlichkeiten zu *HPP* zutage. Die Unterschiede zum *HPP*-Protokoll ergeben sich jedoch wieder aus der Art der verwendeten Baumstruktur. In *HPP* ist sie "ungerichtet"; Nachrichten werden einfach an alle Nachbarn weitergeleitet. Hier braucht man zur Garantie der Selbststabilisierung immer einen gerichteten Baum. Ein Versand von Nachrichten "auf gut Glück" an alle Nachbarn im Kommunikationsbaum wie bei *HPP* wäre hier wie oben besprochen möglicherweise fatal.

5.4 Selbststabilisierung und das Plazierungsproblem bei Replikation

Bisher wurden nur Systeme betrachtet, in denen jeder Knoten eine eigene Kopie des Replikationsdatums besaß. Dies entspricht einer im voraus festgelegten Verteilung der Replikate auf bestehende Maschinen. Andere Rechner, die auf den replizierten Dienst zugreifen wollen, müssen dann über einen sogenannten Verzeichnisdienst den ihnen am nächsten stehenden (funktionierenden) Knoten ausfindig machen und ansprechen. Kennt man die Muster, die diesen Zugriffen zugrundeliegen, im voraus, dann kann man die Replikate so anordnen, daß die Ressourcen im Netzwerk relativ zu einem festzulegenden Kostenmaß optimal genutzt werden. Bei einer Anwendung, in denen Replikate oft gelesen und wenig geschrieben werden, ist beispielsweise eine breite Verteilung der Replikate sinnvoll, um die Leseoperationen möglichst lokal und somit "billig" zu gestalten [142, S. 113]. Auf der anderen Seite möchte man in einer Anwendung, in der die Replikate oft verändert werden, möglichst wenige Kopien im Netzwerk verteilen, da eine Aktualisierung je nach geforderter Konsistenzbedingung "hohe" Kosten mit sich bringt:

- Bei starker Konsistenz führt eine Schreiboperation zu einer teilweisen Blockierung des Systems bis alle oder die Mehrheit der Replikate die Schreiboperation ausgeführt haben.

- Bei schwacher Konsistenz hängt die Geschwindigkeit, mit der sich neue Werte über alle Knoten verteilen, entscheidend von der Anzahl der Replikate und der zwischen ihnen bestehenden Verbindungsstruktur ab.

Es gibt bisher erst wenige Verfahren, die eine anfangs statisch vorgegebene Verteilungsstruktur dynamisch an veränderte Zugriffsmuster anpassen und sich zu einer kostenoptimalen Verteilung stabilisieren. Der Algorithmus für *dynamic data allocation* (*DDA*) von Wolfson und Jajodia [142] ist ein solches Verfahren. In diesem Abschnitt werde ich ausgehend von den Ideen des *DDA*-Algorithmus das Problem der Plazierung von Replikaten mit der bereits in den vergangenen Abschnitten dieses Kapitels eingesetzten Methodik analysieren und einer selbststabilisierenden Lösung zuführen.

5.4.1 Kostendefinition des *DDA*-Algorithmus

Optimale Plazierung von Replikaten verlangt die Definition eines Kostenmaßes, anhand dessen man die Optimalität überprüfen kann. Traditionell ist dies die Domäne der Komplexitätstheorie, in der man den Aufwand eines Algorithmus durch eine Abschätzung seines Verbrauchs an Betriebsmitteln mißt. Als Betriebsmittel muß man im Zusammenhang verteilter Systeme neben den traditionellen Größen wie Speicher und Rechenzeit auch die zur Kommunikation notwendigen Ressourcen wie Anzahl der Nachrichten und deren Größe in Betracht ziehen [136, S. 69f]. Die Kosten sind auch abhängig vom jeweiligen Systemmodell und den darin zur Verfügung stehenden Operationen.

Im Falle des *DDA*-Algorithmus [142] definieren die Autoren dessen Kosten in Abhängigkeit von der Anzahl der vorhandenen Replikate und der Anzahl der Lese- und Schreibzugriffe einzelner Prozessoren darauf.

5.17 Definition (Kosten im *DDA*-Algorithmus [142, S. 114ff]) Betrachte ein repliziertes Datenobjekt r. Die Teilmenge R aller Prozessoren, die ein Replikat von r enthalten, heißt *r-Schema* (*r-scheme*). Zu jeder Zeit gibt es genau eine Primärkopie (*primary copy*) von r. Der Prozeß, auf dem die Primärkopie liegt, sei mit $P(r)$ bezeichnet. $P(r)$ ist der einzige Prozeß, der das r-Schema kennt. Alle anderen Prozessoren kennen $P(r)$.

Prozessoren initiieren Lese- und Schreiboperationen auf r. Eine Leseoperation durch Prozeß i besteht aus einem Zugriff auf die lokale Kopie von r, falls i zum r-Schema gehört. Falls hingegen $i \notin R$, richtet sich i an die Primärkopie $P(r)$, welche ihm den Wert von r zurücksendet. Die Zugriffskosten auf eine lokale Kopie von r seien 1. Die Kosten, um einen Wert von einem Knoten zum anderen zu schicken, seien d. Die Kosten $c_r(i)$ einer Leseoperation auf Knoten i sind daher 1, falls $i \in R$ und $1 + d$ falls $i \notin R$.[3]

[3] Eigentlich müßten die Kosten für die letztere Operation $1 + 2 \cdot d$ lauten, da zwei

Eine Schreiboperation von Knoten i, der nicht zum r-Schema gehört, erfordert das Senden einer Nachricht an $P(r)$, welcher den neuen Wert an alle Mitglieder des r-Schemas verteilt. Gehört i bereits zu R, dann braucht $P(r)$ den neuen Wert natürlich nicht an i zu verteilen. Die Kosten $c_w(i)$ einer Schreiboperation von Knoten i sind demnach $|R| \cdot (1+d)$ falls $i \notin R$ und $|R| \cdot (1+d) - d$ falls $i \in R$.

Sei $S = (s_1, s_2, \ldots, s_k)$ eine Folge von Lese- und Schreiboperationen. Das *Schreib-/Lese-Muster* (*read/write pattern*) von S ist die Menge $A = \{(R_i, W_i) | 1 \leq i \leq n\}$, wobei R_i und W_i die Anzahl der Lese-, bzw. Schreiboperationen von Knoten i aus S sind.

Die Kosten eines r-Schemas R in Bezug auf A (bezeichnet mit $cost(R, A)$) sind die Summe über die Kosten aller Lese- und Schreiboperationen aus A, also:

$$cost(R, A) = \sum_{i=1}^{n} (R_i \cdot c_r(i) + W_i \cdot c_w(i))$$

Ist das Schreib-/Lese-Muster regelmäßig geworden, verteilt der *DDA*-Algorithmus nach beschränkter Zeit die Replikate so, daß die Kostenfunktion optimal ist [142, S. 118]. Das bedeutet, daß kein anderes r-Schema niedrigere Kosten für dasselbe Schreib-/Lese-Muster verursacht.

Informal gesprochen erreicht der *DDA*-Algorithmus dies durch das folgende Vorgehen: Die einzelnen Knoten führen eine lokale Statistik darüber, wieviele Schreib- und Leseoperationen sie innerhalb der letzten Zeiteinheit initiiert haben und — im Falle von Knoten im r-Schema — wieviele Schreiboperationen anderer Knoten empfangen worden sind. Aus diesen Daten kann ein Knoten i, der ein Replikat besitzt, berechnen, wie hoch die Kostenersparnis durch das lokale Halten der Kopie ist. Falls die Kosten, um die Schreiboperationen von i lokal auszuführen, größer sind als die Kosten, um die Leseoperationen von i entfernt zu bearbeiten, löscht i sein lokales Replikat [142, S. 155].

Wie oben beschrieben ist die Primärkopie eine ausgezeichnete Station, über die alle nicht-lokalen Leseoperationen und alle Schreiboperationen laufen. Im Gegensatz zu den Einzelknoten, die nicht zum r-Schema gehören und somit kein Wissen über die globalen Schreibzugriffe auf das replizierte Datenobjekt haben, bemerkt die Primärstation bei einer starken Lesetätigkeit eines Knotens i, daß man billiger wegkommen kann, wenn man auf i eine lokale Kopie von r anlegt, auch wenn dadurch die Schreibkosten insgesamt pro Schreiboperation anwachsen. In diesem Falle wird i eine Kopie von r zugewiesen und i in das r-Schema aufgenommen.

Insgesamt sind hier drei Punkte wesentlich:

- Kosten hängen vom konkreten Replikationsalgorithmus ab. Insbesondere hängen davon die konkreten Werte für $c_w(i)$ und $c_r(i)$ ab.

Nachrichten (Hin- und Rücknachricht) versandt werden müssen, doch will ich mich hier an die Formulierung im Originalartikel halten. Die Ergebnisse unterscheiden sich auch nur um einen konstanten Faktor.

- Kosten werden jeweils für eine längere Zeitspanne berechnet (hier manifestiert in Form einer Befehlsfolge S, bzw. des entsprechenden Lese/Schreib-Musters A).
- Die Entscheidung eines Knotens, das r-Schema zu verlassen, fällt lokal aufgrund von lokaler Information des Knotens. Im Gegensatz dazu erfolgt die Entscheidung des Beitritts zentral durch die ausgezeichnete Primärstation $P(r)$.

5.4.2 Analyse der Kostenberechnung des *DDA*-Algorithmus

Warum darf ein Knoten nicht aufgrund lokaler Information dem r-Schema beitreten? Damit ein Beitritt zum r-Schema die Kosten reduzieren kann, müssen die zusätzlichen Kosten, die ein vergrößertes r-Schema mit sich bringt, durch die wegfallenden Kommunikationskosten mit $P(r)$ mehr als aufgewogen werden. Um diese Entscheidung zu treffen, müßte ein Knoten also zum einen das r-Schema kennen und zum anderen wissen, wieviele Schreiboperationen hierauf das letzte Schreib-/Lese-Muster enthielt. Beides sind aber Informationen, die nur der ausgezeichneten Primärstation zur Verfügung stehen. Man benötigt für diese Entscheidung also eine Art "globales" Wissen. Im *DDA*-Algorithmus wird die zentrale Primärstation also nicht nur dafür benötigt, um 1-Kopie-Serialisierbarkeit zu gewährleisten, sondern auch um globales Wissen zu akkumulieren und auf dieser Basis Entscheidungen zur Kostenoptimierung zu treffen.

Ist das Optimalitätskriterium lokal überprüfbar? Mit den eben gemachten Ausführungen ist es unwahrscheinlich, daß das Optimalitätskriterium des *DDA*-Algorithmus lokal überprüfbar ist. Um dies zu klären, kann man die folgenden Überlegungen anstellen: Knoten, die im *DDA*-Algorithmus zum r-Schema gehören, halten lokale Informationen über die Zahl der Schreibzugriffe aller Knoten. Im Gegensatz dazu speichert ein Knoten, der kein Replikat enthält, keinerlei lokale Statistiken und hätte auch lediglich Zugriff auf Daten über sein eigenes Schreib- und Leseverhalten. Angenommen ein Knoten träfe auch die Beitrittsentscheidung zum r-Schema aufgrund dieser lokalen Informationen (zum Beispiel wenn es deutlich mehr Lese- als Schreibzugriffe gibt). Durch den Beitritt (und das zusätzliche Replikat) entstehen zusätzliche Schreibkosten, und nun kann es sein, daß diese Schreibkosten aufgrund einer hohen Schreibtätigkeit in anderen Teilen des Systems den Vorteil des lokalen Lesens mehr als aufwiegen. Im nächsten Schritt würde derselbe Knoten also wieder das r-Schema verlassen. Da sich das Schreib-/Lese-Muster nicht verändert hat, wird das r-Schema nie konvergieren.

Bei dieser Überlegung kommt es entscheidend darauf an, daß die Aufhebung des Kostenvorteils *irgendwo* im System passiert und nicht durch die Formulierung von Kantenprädikaten, die ja nur die lokale Nachbarschaft miteinbeziehen, erfaßt werden kann. Selbst wenn man die globalen Informationen über die Schreibtätigkeit des Systems auf allen Knoten repliziert und darauf lokale Prädikate formuliert, sind diese Informationen schnell veraltet, und Entscheidungen aufgrund der so vorhandenen Informationen sind nicht immer richtig. Hier spielt wieder die Frage hinein, was ein Prozeß tatsächlich "weiß" (vergleiche Definition 4.20). Die Optimalitätsbedingung des DDA-Algorithmus ist darum wahrscheinlich nicht lokal überprüfbar.

5.4.3 Umformulierung des Optimalitätskriteriums

Das Resultat des letzten Abschnittes mag auf den ersten Blick ernüchtern, zeigt es doch, daß für die Kostenkontrolle in der Art des DDA-Algorithmus nur Verfahren mit globalem Prüfen und globaler Korrektur in Frage kommen. Auf der anderen Seite ist dieses Ergebnis allerdings nicht überraschend, da auch das *flood-d*-Protokoll die Optimalität der Verbindungsstruktur durch die Sammlung statistischer Daten auf einem zentralen Knoten und durch ihn initiierte Umkonfigurationsentscheidungen erreicht (vergleiche Abschnitt 3.6.3).

Bevor man sich aber auf die Realisierung selbststablisierender Kostenkontrollalgorithmen stürzt, stellt sich die Frage, ob es nicht noch andere Formen einer Kostendefinition gibt, die vielleicht doch lokal überprüfbar sind? Tatsächlich gibt es entsprechende Ansätze [11]: In ihnen sind die globalen "Kosten" optimal, wenn die "Last" auf allen beteiligten Prozessoren gleichmäßig verteilt ist. In der Fachliteratur spricht man hier von dem Problem des Lastausgleichs (*load balancing*) [128]. Es gilt also, das Plazierungsproblem von Replikaten in Form des *load balancing* umzuformulieren.

Optimierung durch Lastausgleich. Betrachte ein Netzwerk mit n Knoten in dem ein Dienst r durch ein beliebiges schwach-konsistentes Replikationsverfahren auf einer Teilmenge R der Knoten (das r-Schema) zur Verfügung gestellt wird. Um einen Zusammenhang zwischen "Last" und "Kosten" herzustellen, will ich die folgenden Annahmen treffen:

- Da es sich um ein schwach-konsistentes Replikationsverfahren handelt, blockieren Änderungen von r nicht das System. Die Größe von R ist also im Gegensatz zu stark-konsistenten Verfahren keine kritische Größe in Bezug auf den Durchsatz von Lesezugriffen sondern beeinflußt lediglich die Zeitspanne, bis Änderungen zu allen Replikaten durchgedrungen sind.

- Lesezugriffe stellen den Hauptteil der Last und damit der Betriebskosten einzelner Knoten dar.

- Ein Knoten hat eine gewisse optimale Auslastung gemessen in Lesezugriffen pro Zeiteinheit. Bei dieser Auslastung sind die Betriebskosten aufgrund der guten Ressourcennutzung besonders niedrig. Steigt die Auslastung stark, so steigen die Kosten überproportional stark.

Um das weitere Vorgehen zu vereinfachen, will ich vorläufig annehmen, daß die Anzahl der Replikate konstant ist und daß alle lesenden Stationen die gleiche Last erzeugen, also alle dieselbe Anzahl an Lesezugriffen pro Zeit haben. Unter diesen Annahmen sind die Bereitstellungskosten des Dienstes genau dann optimal, wenn alle Stationen, die eine Version des Dienstes tragen, etwa gleich stark ausgelastet sind und keine Station eine Überlast trägt.

5.4.4 Lastausgleich durch lokales Prüfen und lokale Korrektur

Lastausgleich in verteilten Systemen ist eine Anwendung, von der Arora und Gouda 1995 gezeigt haben, daß sie mit lokalem Prüfen und lokaler Korrektur lösbar ist [11]. Intuitiv scheint dieses Resultat eine direkte Folgerung aus dem in dieser Arbeit erbrachten Ergebnis zu sein, daß das Konsistenzproblem durch dieselbe Technik lösbar ist (vergleiche Abschnitt 5.2.2), denn auch beim Lastausgleich wird eine gewisse (wenn auch nicht exakte) Konsistenz bestimmter Werte innerhalb der Knoten gefordert.

Arora und Gouda beschreiben die Umgebung, in der ihr Algorithmus funktioniert, sehr allgemein: Auf den Knoten (hier den Replikaten) des Systems gibt es jeweils eine lokale Variable x, die eine Anzahl von ganzzahligen "Lasteinheiten" repräsentiert, die diese Station momentan trägt [11, S. 185]. Die parallel ablaufende Anwendung darf jederzeit und auf jeder Station die Last positiv und/oder negativ beeinflussen. Wenn diese Änderungen aufhören, stabilisiert sich ihr Algorithmus zu einem Zustand, in dem die Werte von x für alle Knoten sich maximal um den Wert 1 unterscheiden. Das Legalitätsprädikat lautet also:

$$P \equiv \forall u, v \in R : |x_u - x_v| \leq 1$$

Dazu werden nur lokale Prüf- und Korrekturaktionen verwendet. Voraussetzung ist jedoch, daß die den Replikationsknoten unterliegende Kommunikationstopologie ein gerichteter, azyklischer Graph ist [11, S. 191].

Der Algorithmus. Angelehnt an die Ausführungen von Arora und Gouda [11] will ich nun den Algorithmus zur Lastbalancierung im Rahmen eines Replikationsverfahrens vorstellen und dabei nur den Fall betrachten, daß die Replikate durch eine Baumstruktur mit einer ausgezeichneten Wurzel verbunden sind.

5.18 Algorithmus (Lastausgleich in Baumstrukturen nach Arora und Gouda [11]) Die Idee des Algorithmus ist, bei unterschiedlichen Lasten zunächst einen lokalen Lastausgleich vorzunehmen. Knoten u und v, die eine Verletzung des lokalen Prädikates $x_u - x_v > 1$ feststellen (o.B.d.A. sei $x_u > x_v$), tauschen untereinander einfach d Lasteinheiten aus, wobei $0 < d < x_u - x_v$. Durch diese Aktionen allein ist die Konvergenz zu P noch nicht garantiert, da allein durch Herstellung der lokalen Bedingung beispielsweise eine streng monoton fallende Folge von Werten $x_0 = 5$, $x_1 = 4$, $x_2 = 3$ usw. resultieren könnte. Man muß also noch zwei zusätzliche Bedingungen lokal überprüfen [11, S. 189]:

1. Die Werte von x auf jedem Pfad von der Wurzel zu einem Blatt dürfen nicht zunehmen.

2. Die Werte von x auf jedem Pfad von der Wurzel zu einem Blatt dürfen nur ein Mal um den Wert 1 abnehmen.

Bei einem Vorgehen gemäß Bedingung 1 wird also bei der lokalen Korrektur immer dem Knoten in Richtung der Wurzel im Zweifelsfall der höhere Wert zugewiesen. Das bedeutet, daß es keine unendlichen Folgen von Aktionen gibt, die eine Lasteinheit zwischen zwei Knoten hin- und hertauschen. Bedingung 2 stellt dann insgesamt sicher, daß am Ende P gilt.

Folgende Beobachtung führt zu einer Vorgehensweise, die Bedingung 2 sicherstellt: Falls P gilt, gibt es auf jedem Pfad von der Wurzel bis zu einem Blatt des Baumes maximal einen Knoten, bei dem sich die Last um genau 1 von der seines Elternknotens unterscheidet. Durch die Einführung einer lokalen booleschen Variable b wird lokal angezeigt, ob man selbst vor oder hinter diesem möglichen Knoten auf dem Pfad liegt. Der Wert von b in der Wurzel ist immer *true*, und ein Nachfolger wird seinen eigenen Wert genau dann auf *false* setzen, wenn seine Last sich um 1 von der seines Vorgängers unterscheidet. Für alle nachfolgenden Knoten auf dem Pfad ist nun feststellbar, ob sie vor oder hinter diesem Knoten stehen, wenn der Wert von b entsprechend im Netzwerk propagiert wird.

Stellt ein Knoten fest, daß bei ihm ein zweiter "Lastabstieg" stattfindet, dann muß er eine Lasteinheit von seinem Vorgänger übernehmen. Nun kann es aber vorkommen, daß genau diese Lasteinheit im nächsten Schritt wieder zurückgetauscht wird, um Bedingung 1 wieder herzustellen. Eine unendliche Folge von Tauschaktionen wäre die Folge. Dies kann jedoch verhindert werden, indem beide Tauschoperationen nur dann zugelassen werden, wenn eine entsprechende Aktion mit dem unmittelbaren Vorgängers der beiden betrachteten Knoten nicht möglich ist. Dies ist eine Bedingung, die einer der am Tausch beteiligten Knoten leicht überprüfen kann. Eine Visualisierung dieses Sachverhaltes ist in Abbildung 5.13 dargestellt, der Algorithmus in Abbildung 5.12. Insgesamt wird so das Legalitätsprädikat erfüllt und ein Lastausgleich hergestellt [11, S. 190].

$$
\begin{array}{l}
\textbf{process}\ \textit{load balancing node}\ i\\
\quad \textbf{const}\ i\ \textbf{init}\ \{^*\ \text{own node id}\ ^*\}\\
\quad\ \ p\ \textbf{init}\ \{^*\ \text{parent node id}\ ^*\}\\
\quad \textbf{var}\ x \in \mathbb{N}\ \{^*\ \text{own workload}\ ^*\}\\
\quad\ \ b \in \{\textit{true}, \textit{false}\}\ \{^*\ \text{“downstep” flag}\ ^*\}\\
\quad\ \ ok \in \{\textit{true}, \textit{false}\}\ \{^*\ \text{ok to swap below}\ ^*\}\\
\quad \textbf{begin}
\end{array}
$$

$$
\begin{array}{lllll}
& \{^*\ \text{balance load locally:}\ ^*\} & & \\
& x > x_p \wedge x - x_p > 1 & \longrightarrow & \textbf{swap}(i, p, \lceil \frac{x - x_p}{2} \rceil) \\
[\!] & x < x_p \wedge x_p - x > 1 & \longrightarrow & \textbf{swap}(p, i, \lfloor \frac{x_p - x}{2} \rfloor) \\
& \{^*\ \text{swap larger workload to parent:}\ ^*\} & & \\
[\!] & x = x_p + 1 \wedge ok_p & \longrightarrow & \textbf{swap}(i, p, 1) \\
& \{^*\ \text{propagate “errorneous” workload deficits:}\ ^*\} & & \\
[\!] & x + 1 = x_p \wedge \neg b_p \wedge ok_p & \longrightarrow & \textbf{swap}(p, i, 1) \\
& \{^*\ \text{maintain}\ b\text{:}\ ^*\} & & \\
[\!] & x = x_p \wedge b \neq b_p & \longrightarrow & b := b_p \\
[\!] & x + 1 = x_p \wedge b \wedge b_p & \longrightarrow & b := \textit{false} \\
& \{^*\ \text{maintain}\ ok\text{:}\ ^*\} & & \\
[\!] & x = x_p \wedge b = b_p & \longrightarrow & ok := \textit{true} \\
[\!] & x + 1 = x_p \wedge \neg b \wedge b_p & \longrightarrow & ok := \textit{true} \\
[\!] & x \neq x_p \wedge (x + 1 \neq x_p \vee b \vee \neg b_p) & \longrightarrow & ok := \textit{false} \\
[\!] & b \neq b_p \wedge x + 1 \neq x_p & \longrightarrow & ok := \textit{false} \\
[\!] & b \wedge \neg b_p & \longrightarrow & ok := \textit{false}
\end{array}
$$

end

Abbildung 5.12: Lokaler Algorithmus eines Nicht-Wurzelknotens im Algorithmus zum Lastausgleich. Das Kürzel $\textbf{swap}(i, j, k)$ steht für das Verschieben von k Lasteinheiten von i nach j.

Anmerkungen zu Algorithmus 5.18. In Abbildung 5.12 wurde für den Wert d der ausgetauschten Lasteinheiten jeweils ein konkreter Wert eingesetzt, der betragsmäßig die größte Annäherung der Werte von x und x_p mit sich bringt. Die obere und untere Gaußklammer sorgt in Fällen, in denen $|x - x_p|$ ungerade ist, dafür, daß jeweils der um eins größere Anteil beim Elternknoten verbleibt.

Die Variable ok zeigt an, ob der Zustand des Knotens in Einklang mit dem des Elternknotens ist (also $ok \equiv (x = x_p \wedge b = b_p) \vee (x + 1 = x_p \wedge \neg b \wedge b_p)$). Nur wenn die Variable ok des Elternknotens den Wert “wahr” hat, darf ein unterliegender Knoten wie im Algorithmus besprochen den Tausch einer Einheit aufgrund eines “zweiten Abstieges” veranlassen. Die letzten drei bewachten Anweisungen, die ok auf den Wert *false* setzen, sind die vereinfachten disjunktiven Terme der Negation der ok-Bedingung.

Das Kürzel $\textbf{swap}(i, j, k)$ steht für das Verschieben von k Lasteinhei-

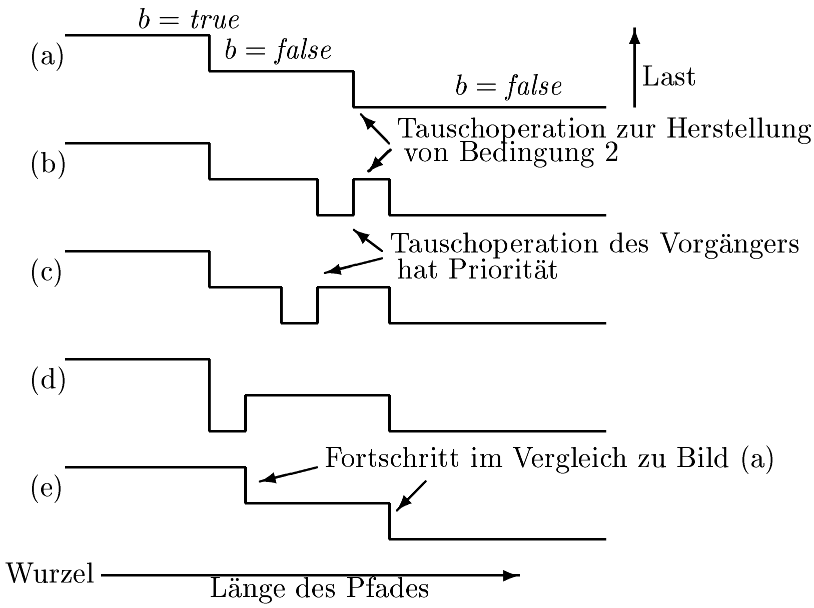

Abbildung 5.13: Visualisierung der Priorität von Tauschaktionen aus Algorithmus 5.18.

ten von i nach j. Im Originalartikel wird nicht darauf eingegangen wie dieses im einzelnen vor sich geht. Die Autoren übergehen dabei das Problem, daß ein solcher Tausch in einer Art "lockerer Transaktion" von beiden Knoten gemeinsam durchgeführt werden muß. Da der Baum lediglich über einen Verweis auf den Elternknoten realisiert wurde, ist den Elternknoten weder die Anzahl noch die Identität ihrer "Kinder" zum Tauschzeitpunkt bekannt. In einem Teil der Fälle muß darum die Information über den Tausch vom Nachfolgerknoten an den Elternknoten erst mühsam übermittelt werden, wofür zusätzliche Variablen und Aktionen notwendig wären.

Umsetzung im Rahmen von Replikationsalgorithmen. Die hier vorgeschlagene Realisierung von Replikation unter Kostengesichtspunkten baut auf dem eben vorgeschlagenen Algorithmus 5.18 auf. Es wurde unter Verweis auf den Originalartikel [11] argumentiert, daß sich ein derart gestaltetes System selbst zu einem Zustand stabilisiert, in dem alle Stationen etwa die gleiche Last tragen, wenn die Produktion und Konsumption von lokalen Lasteinheiten durch eine überlagerte Anwendung aufhören. Die Übertragung auf die Replikation basiert auf den vorläufigen Annahmen, die zum Ende von Abschnitt 5.4.3 getroffen wurden:

- Die Anzahl der Replikate ist konstant.
- Lesezugriffe stellen den Hauptteil der Last dar.
- Lesende Stationen erzeugen alle die gleiche Last.

- Die Betriebskosten sind bei gleicher Auslastung aller Replikate optimal.

Hier nun entspricht eine Lasteinheit einer Station, die ein Replikat liest. Wenn sich das Leseverhalten der einzelnen Stationen auf einem bestimmten Niveau eingeschwungen hat, werden die einzelnen Replikate untereinander dafür sorgen, daß sie alle etwa gleichviele lesende Stationen zugeteilt bekommen. Gemäß der Annahme, werden gerade dann die Kosten insgesamt besonders gut ausfallen.

In diesem Zusammenhang wird auch deutlich, wie man die oben noch als problematisch angesehene **swap**-Routine realisieren kann: Der Befehl **swap**(i, j, k) soll bewirken, daß sich k lesende Prozesse in Zukunft nicht mehr an i sondern an Knoten j wenden sollen. Dies ist realistisch entweder durch die Informierung des Auskunftsdienstes möglich, der Leseanfragen auf die Replikate verteilt, oder durch die direkte Information der entsprechenden Knoten durch den Versand einer Nachricht. (Im Modell mit gemeinsamen Speicher entstehen allerdings die gleichen Probleme, die oben bereits beschrieben wurden.) In diesem Falle realisieren die Replikate selbst eine Art Auskunftsdienst und entlasten damit die globale Auskunft, die für eine initiale Zuweisung der Replikate immer noch benötigt wird.

Bewertung der Ergebnisse. Es gibt eine ganze Reihe von Kritikpunkten an der vorgestellten Lösung. Zunächst kann man die vorläufig getroffenen Annahmen als unrealistisch kritisieren. Prozesse haben nicht alle das gleiche Leseverhalten, und es sind viele Fälle denkbar, in denen eine konstante Anzahl von Replikaten ungünstig ist, entweder weil alle überlastet sind oder alle fast keine Arbeit zu tun haben. Eine dynamische Anpassung des Systems wäre sehr wünschenswert. Diese Punkte sind sehr wichtig und sollen darum gesondert im folgenden Abschnitt 5.4.5 aufgegriffen werden.

Ein weiterer Kritikpunkt bezieht sich auf das Ignorieren der Lokalität von Lesezugriffen: Nahes und entferntes Lesen erzeugt in der Realität stark unterschiedliche Kosten. Dies wird in dem Modell nicht berücksichtigt. Das "Verschieben" von lesenden Knoten ohne Kostennachteile ist bis zu einem gewissen Grad dadurch zu rechtfertigen, daß durch eingebaute Regeln möglichst immer genau die Knoten verschoben werden, die auch "nahe bei" ihrer neu zugewiesenen Version des Replikates liegen. In Extremfällen, in denen beispielsweise viele lesenden Knoten nahe bei einem Replikat liegen und alle anderen Replikate keine Last tragen, wird das System in einen Zustand kommen, in dem Knoten auf Replikate zugreifen, die weit entfernt sind. Dies ist unerwünscht.

5.4.5 Dynamik in Last und Leseverhalten

In diesem Abschnitt will ich die zwei wesentlichen Annahmen betrachten, die weiter oben Anlaß zur größten Kritik gaben: die Gleichheit des

Leseverhaltens aller Knoten und die konstante Anzahl von Replikaten.

Dynamisches Leseverhalten. Das Leseverhalten unterschiedlicher Knoten ist in der Realität möglicherweise sehr verschieden und hat über die Zeit eine gewisse Dynamik. Eine Konsequenz daraus wäre, nicht die Anzahl der lesenden Stationen sondern die Anzahl der Lesezugriffe selbst als Maß der Last zu wählen. Ein System würde sich selbst dann (bei einer ersten Interpretation des Algorithmus) in einen Zustand bringen, in dem jedes Replikat die gleiche Anzahl von Lesezugriffen zu verarbeiten hat. Dies ist offensichtlich ein besseres Ziel als nur die lesenden Stationen überall gleich zu verteilen. Jedoch ergeben sich mit diesem Ansatz Probleme.

Das wesentliche Problem ist, daß Lasteinheiten nicht mehr in beliebigem Maße verschoben werden können, wenn man vermeiden will, daß ein und dieselbe lesende Station sich abwechselnd an mehrere Replikate wenden muß. Beharrt man auf einer eindeutigen Zuordnung von lesendem Knoten zu Replikat, ist eine absolute Gleichverteilung nicht immer möglich, wie das folgende Beispiel zeigt.

5.19 Beispiel Betrachte zwei Replikate r_1 und r_2, die beide jeweils von einem Prozeß p_1 bzw. p_2 gelesen werden. Prozeß p_1 lese in einer Zeiteinheit 90mal das Replikat r_1 und p_2 lese in derselben Zeit nur zehnmal r_2. Nimmt man die Anzahl der lesenden Stationen als Lastmaß, ist das System in einem ausgeglichenen Zustand. Will man jedoch die Anzahl der Leseoperationen ausgleichen, ohne die Anzahl der Lesezugriffe einzelner Knoten zu fragmentieren, stößt man hier schnell auf Grenzen. Zwar wird r_1 immer feststellen, daß die Differenz zwischen beiden Lasten größer als 1 ist, doch kann ein Austausch nicht stattfinden, da der auszutauschende Wert d im Bereich zwischen 0 und $90 - 10 = 80$ liegen muß. Der einzige Wert, der jedoch auf einmal ausgetauscht werden kann, ist 90. Ein Ausgleich kann also nicht stattfinden.

Eine Verallgemeinerung der Konstruktion des letzten Beispiels führt zu dem Schluß, daß es bei einer beliebigen Konfiguration von Replikaten und lesenden Stationen immer ein solches Szenario gibt, in der ein absoluter Lastausgleich unmöglich ist, wenn nur die Lesezugriffe eines Prozessors in einem Block verschoben werden dürfen. Dies gilt auch, wenn der Lastunterschied der Stationen nicht nur maximal 1, sondern auch einen beliebigen (festen) Wert erreichen darf. Es hilft dabei auch nicht, wenn man die Anzahl der Replikate beliebig wählen darf.

Das Problem hierbei sind bei näherer Betrachtung diejenigen Prozesse, die außerordentlich viele Lesezugriffe verursachen. Die allein durch diese Knoten verursachte Last kann das gesamte System blockieren. Eine mögliche Lösung dieses Problems wäre, diesen “Viel-Lesern” eine lokale Kopie des Replikates zuzuweisen und sie dadurch zufriedenzustellen. Dies führt unmittelbar auf die Behandlung der zweiten kritisierten Annahme.

Dynamik in der Anzahl der Replikate. Eine konstante Anzahl der Replikate ist in der Praxis zwar üblich, doch in vielen Fällen ungünstig: Manche Knoten haben zu bestimmten Tages- oder Saisonzeiten (oder ohne ersichtlichen Grund) eine erhöhte Leseaktivität und erzeugen dadurch auf den ihnen zugewiesenen Replikaten eine hohe Last. Ein Algorithmus sollte in solchen Fällen genau auf diesem Prozessen lokale Kopien des Replikates anlegen, um ihren Bedarf besser und effizienter zu decken. "Effizienter" ist es deshalb, weil lokale Lesezugriffe in der Regel um ein Vielfaches schneller und unkomplizierter sind als der Zugriff auf ein entferntes Replikat.

Allein die Anlage eines neuen Replikates löst allerdings noch nicht das oben geschilderte "Ausgleichsproblem"; es wird also immer noch Situationen geben, in denen die Last (gemessen in der Anzahl der Lesezugriffe) nicht ausgeglichen werden kann. Ein Lösungsansatz hierzu wäre, lokale Lesezugriffe als "leichter" zu betrachten, also die durch sie verursachte Last um einen gewissen Faktor zu reduzieren. Dies würde dazu führen, daß man die Lastanteile notorischer "Viel-Leser"-Knoten reduziert, dadurch die Spitzen aus der Leselast nimmt und durch diese Entspannung möglichweise eine Situation herbeiführt, die insgesamt doch (in einem gewissen Toleranzbereich) ausgeglichen werden kann.

Durch das Lösen des "Viel-Leser"-Problems ergeben sich aber eine Vielzahl neuer, interessanter und höchst diffiziler Fragen:

- Die Last eines Knotens x ist nun nicht mehr ohne weiteres mit der von Knoten y vergleichbar. Wie kann man also gewährleisten, daß durch den Austausch von Lasteinheiten von x nach y die Last auf dem Zielknoten tatsächlich reduziert wird?
- Wenn das lokale Leseinteresse eines Knotens erliegt, besteht im System im Vergleich zu vorher eine Überkapazität, die einen kurzzeitigen Kostenvorteil schnell wieder zunichte machen kann. Wie kann man also diese Überkapazitäten erkennen und beseitigen, ohne auf anderen Knoten wieder eine Überlast zu erzeugen?

Es ist nicht schwer, diese Fragen als Forderungen zu definieren und sie in Form lokaler Prädikate zu formulieren, doch ist der Nachweis der Konvergenz dadurch leider noch nicht einmal ansatzweise begonnen.

5.5 Ausblick

In diesem Kapitel ging es um den Entwurf und die Ausformulierung schwach-konsistenter Replikationsalgorithmen, welche selbststabilisierend sein sollten. Ich bin dabei ausgegangen von verschiedenen Formalisierungen zentraler Aspekte von Replikation und habe anschließend aus den entsprechenden Spezifikationen Algorithmen zur Lösung der Probleme abgeleitet. Als zentraler Punkt dieser Bemühungen ist ein selbststabilisierender zuverlässiger *broadcast* entstanden, der die Grundlage für komplexere Algorithmen auf diesem Gebiet sein kann. Bemühungen, die anfangs so erfolgreich eingesetzte Methodologie auf das Plazierungsproblem zu übertragen, sind zwar nicht gescheitert, aber doch nur zu einem sehr unbefriedigenden Ergebnis gekommen.

Im folgenden Kapitel, welches den Abschluß der Arbeit bildet, werden diese Ergebnisse nochmals im Hinblick auf ihre Relevanz für die Praxis betrachtet und zusammengefaßt.

6

Ergebnisse und weitere Arbeiten

Inhaltsübersicht

Kurzfassung

In Kapitel 5 wurden die Grundlagen geschaffen, um selbststabilisierende Replikationsalgorithmen zu realisieren. Die Ergebnisse versprechen Verfahren, die beinahe alle Formen von Fehlern tolerieren können und somit zur Zuverlässigkeit von Rechnernetzen beitragen könnten. In diesem Kapitel werden diese Ergebnisse kritisch beäugt: Wie können die entworfenen Verfahren in der Praxis umgesetzt werden? Wie schneiden sie im Vergleich zu bereits existierenden Algorithmen ab? Neben Antworten auf diese Fragen wird auch noch besprochen, welche wesentlichen Punkte in dieser Arbeit offengeblieben sind und wie die hier begonnenen Untersuchungen fortgesetzt werden können.

6.1 Einführung

Die im vorangehenden Kapitel vorgestellten Replikationsalgorithmen sind bisher nur entworfen und verifziert worden. In diesem Kapitel will ich sie theoretisch und simulativ analysieren um festzustellen, wie und in welchem Maße sie sich von bereits bestehenden Algorithmen für dieselbe Problemstellung unterscheiden. Dabei werde ich auch auf die praktische Realisierbarkeit der Verfahren eingehen und auch kurz besprechen, welche Einsichten sich für mich persönlich bei der Arbeit an diesem Projekt ergaben. Da dieses Kapitel auch den Abschluß der Diplomarbeit bildet, wurde der gewöhnlich am Ende jedes Kapitels befindliche Abschnitt "Ausblick" in "weitere Arbeiten" umbenannt. In ihm wird beschrieben, wie die in dieser Arbeit gewonnenen Ansätze weiter fortgeführt werden könnten.

6.2 Ergebnisse in praktischer Hinsicht

Bei den im vorangegangenen Kapitel entwickelten Algorithmen stellt sich sofort die Frage nach der praktischen Umsetzung. Zwar gibt es bereits Programmiersprachen, die *guarded commands* in irgendeiner Form zur Verfügung stellen (vergleiche Abschnitt 2.3.2), doch gehören diese nicht zum allgemeinen "Kanon" der Programmiersprachen. Heutzutage beherrschen Bibliotheken wie MPI [84] den Markt, die lediglich verschiedene Kommunikationskonstrukte herkömmlichen Programmiersprachen hinzufügen. Bewachte Anweisungen muß man anschließend selbst nachbilden.

Die Probleme bei der Realisierung bewachter Anweisungen entstehen aus dem Nichtdeterminismus, der Teil des Sprachkonstruktes ist, und dem sogenannten *aktiven Warten* (*busy waiting*). Letzteres resultiert aus der Notwendigkeit, andauernd die Wachen zu evaluieren um festzustellen, ob eine Anweisung ausführbar ist oder nicht. Der Nichtdeterminismus läßt sich in der Realität einfach implementieren: Anstatt zufällige Anweisungen auszuwählen, zu evaluieren und auszuführen, betrachtet man die einzelnen Kommandos sequentiell und zyklisch. Die daraus resultierende Anweisungsfolge ist zwar deterministisch, doch genügt sie trivialerweise der Semantik der bewachten Anweisungen, wenn man einfach "vergißt," in welcher Reihenfolge die Wachen ausgewählt und evaluiert werden. Wesentlich ist dabei lediglich, daß die Fairneß gewahrt bleibt.

Dem aktiven Warten (und dem damit verbundenen Ressourcenverbrauch an Speicher und Rechenzeit) ist etwas schwerer beizukommen. Eine mögliche Realisierung ist aus dem Bereich der Betriebssysteme entlehnt: Ein Knoten arbeitet immer "phasenweise," d.h. er evaluiert seine *guards* zyklisch und führt so lange die zugehörigen Anweisungen aus, bis keine Wache mehr "wahr" ist. Wenn dies der Fall ist, wird auch eine fortgesetzte Evaluierung keinen "Erfolg" mehr bringen. Darum kann sich der

Prozeß in diesem Fall “zur Ruhe setzen” und muß erst dann wieder mit der Arbeit beginnen, wenn “neue Informationen” (hier: eine neue Nachricht) verfügbar sind. Ein Prozeß wechselt somit immer zwischen Arbeits- und Ruhephasen. In den Ruhephasen kann das Betriebssystem dann den Prozeß in einen ressourcenschonenden “Schlafzustand” versetzen. (Eine zusätzliche Optimierung entspringt daraus, daß am Ende von Aktivitätsphasen jeweils nur ein Mal die Anweisung **shout**(m) ausgeführt wird, obwohl die Korrektheit dieser Vereinfachung nicht auf Anhieb einzusehen ist.)

Man beachte, daß durch diese Implementierung keine der Annahmen über die Semantik bewachter Anweisungen verletzt und somit die Korrektheit erhalten bleibt. Eine derartige Realisierung der bewachten Anweisungen hat aber zusätzlich noch den Vorteil, daß sie die entworfenen Algorithmen mit Goldings *TSAE*-Protokoll vergleichbar macht. Golding mißt viele der Leistungsdaten seines Algorithmus in einer Zeit, die in Vielfachen einer *mean anti-entropy time* ausgedrückt wird [73, S. 102f]. Dies ist die Zeit, die zwischen zwei Initiierungen von *anti-entropy sessions* verstreicht. Bei der Realisierung des in dieser Arbeit entworfenen *broadcast* muß man nur noch fordern, daß ein Prozeß frühestens nach einer bestimmten festen Zeit wieder “losarbeiten” darf, selbst wenn eine Nachricht für ihn anliegt. Solange diese Zeit t fest und beschränkt ist, wird die Semantik der bewachten Anweisungen nicht verletzt. Außerdem entpricht t dann der Goldingschen Maßeinheit.

Eine Implementierung muß auch die beiden Bemerkungen 5.11 und 5.12 beachten, welche die Realisierung der **timeout**-Anweisung und die Optimierung der Anzahl der Nachrichten betreffen.

6.2.1 Theoretische und simulative Leistungsanalyse

Bei der theoretischen Analyse des hier vorgestellten Algorithmus wird deutlich, daß die Leistung in großem Maß von der Struktur des unterliegenden aufspannenden Baumes abhängt, obwohl als grundlegendes Verteilungsschema in beiden Fällen *flooding* verwendet wird. Leider gehen die Autoren des *spanning tree*-Algorithmus in ihrem Artikel [39] nicht auf dessen Komplexität ein, und eine genaue Untersuchung dieses Sachverhaltes erscheint in der Tat schwierig (vergleiche Abschnitt 4.3.4). In Anlehnung an Goldings Kapitel zur Leistungsmessung [73, Kapitel 7] habe ich darum die von mir entworfenen Algorithmen in der eingangs besprochenen Art als ereignisgesteuerte Simulation implementiert.

Simulationsannahmen. Aus der Menge der verfügbaren Simulatoren [19, 20, 115] entschied ich mich für das `simjava`-Paket von Ross McNab und Fred Howell [115], weil es zusätzlich auch noch ansprechende Animationsmöglichkeiten in Form sogenannter *Java-Applets* [57] bot. Wie bei Golding [73, S. 102 u. 112] wurde ein Netzwerk mit n Knoten und einer vollständig verbundenen Topologie gewählt. Die lokalen Variablen der Knoten wurden durch Werte einer gleichverteilten Zufallsvariablen initialisiert und jeweils 1000 Durchläufe mit 5, 10 und 20 Knoten berechnet, sowie 400 Durchläufe fr $n = 40$.[1] Die zufällige Auswahl von Nachbarn wurde ebenfalls als ein gleichverteilter Zufallsprozeß modelliert. Die Zeit, die der Versand eine Nachricht zu einem anderen Knoten benötigt, wurde als verlachlässigbar gering im Vergleich zur Zeit zwischen zwei Kommunikationsphasen angesehen [73, S. 102].

Analyse der durchschnittlichen Baumtiefe. Bei den so entstehenden Bäumen wurde jeweils die Tiefe (gemessen in der Anzahl der Kanten, die im längsten Pfad des Baumes enthalten sind) des entstandenen Baumes gemessen und der Durchschnitt über die Anzahl der Simulationsabläufe berechnet. Die Ergebnisse sind in Tabelle 6.1 dargestellt. Sie zeigen relativ gute (=niedrige) Werte für die durchnittliche Baumtiefe, die eine annähernd logarithmischen Verlauf in der Zahl n der Knoten haben. Für spezielle Topologien bietet es sich trotzdem an, einen optimierten Algorithmus zur Spannbaumkonstruktion zu verwenden, der immer einen Baum geringster Tiefe berechnet. Derartige Algorithmen sind bereits veröffentlicht worden [10].

	Baumtiefe					Anzahl der
n	min.	durchschn.	max.	σ^2	σ	Durchläufe
5	1	2,252	4	0,467	0,683	1000
10	1	3,474	7	1,341	1,158	1000
20	1	5,102	13	3,942	1,985	1000
40	1	7,160	14	8,763	2,960	400

Tabelle 6.1: Minimale, durchschnittliche und maximale Tiefe des berechneten aufspannenden Baumes abhängig von der Anzahl n der Knoten zusammen mit empirischer Varianz (σ^2) und Standardabweichung (σ).

Analyse der Konvergenzzeit. Neben der Tiefe der entstandenen Bäume wurde zusätzlich auch die Zeit gemessen, die der Algorithmus benötigte, um zu einen aufspannenenden Baum zu konvergieren. Die verwendete Zeiteinheit entspricht der Zeitdauer, die zwischen zwei Aktivitätsphasen der einzelnen Prozesse mindestens verstreicht. Die Ergebnisse sind in Tabelle 6.2 dargestellt.

Bei Betrachtung der Daten fällt auf, daß sich die durchschnittliche Konvergenzzeit bei Verdopplung der Knotenzahl n ebenfalls verzweifacht. Außerdem ist die maximale Zeit, die zur Konvergenz benötigt wird, zum

[1] Hier zeigte sich einer der Nachteile von Java als Simulationssprache: Die Berechnungen mit 80 und 160 Knoten dauerten so lange, daß sie bei Abgabe der Diplomarbeit immer noch nicht beendet waren.

n	Konvergenzzeit min.	durchschn.	max.	σ^2	σ	Anzahl der Durchläufe
5	1	3,406	12	1,196	1,094	1000
10	3	6,861	25	3,873	1,968	1000
20	5	13,412	21	18,491	4,300	1000
40	8	26,193	41	81,739	9,041	400

Tabelle 6.2: Minimale, durchschnittliche und maximale Konvergenzzeit gemessen in Anzahl der Kommunikationsphasen der Prozesse bei der Berechnung des aufspannenden Baumes.

Teil erheblich. Abbildungen 6.1 bis 6.3 zeigen den Konvergenzverlauf grafisch. Auf der horizontalen Achse ist jeweils die Zeit (gemessen in Kommunikationsphasen) aufgetragen. Die vertikale Achse zeigt, wieviel Prozent der Knoten zum jeweiligen Zeitpunkt schon Teil des Baumes sind. Zur Verdeutlichung der Varianz sind die jeweiligen maximalen und minimalen Werte ebenfalls eingezeichnet. Die Entwicklung des Durchschnittes ist in alle Fällen nahezu linear.

Die Kurve der minimalen Werte zeigt dagegen eine lange "Durststrecke," die plötzlich und rasant jeweils auf 100 Prozent ansteigt. Während es bei vielen anderen verteilten Algorithmen einfach ist, einen beispielhaften Ablauf zur Abschätzung des *worst case* zu konstruieren, erscheint dies hier schwierig. Eine genauere Betrachtung der in der Simulation aufgetretenen "langen" Konvergenzzeiten könnte hier aufschlußreich sein (siehe Abschnitt 6.5 über "weitere Arbeiten").

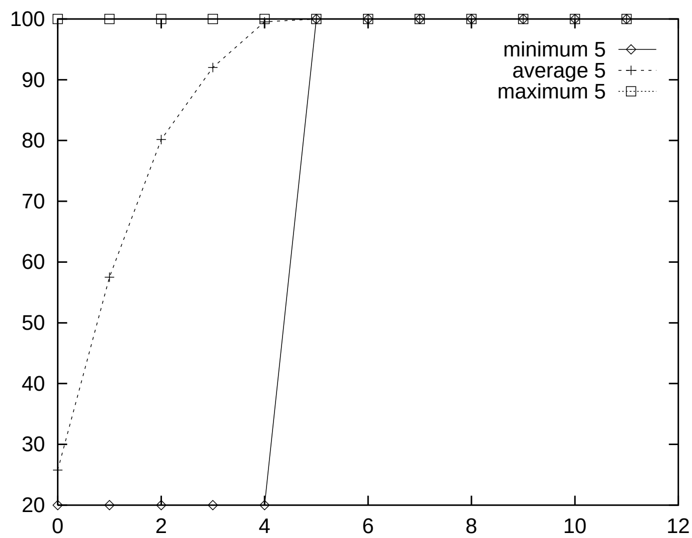

Abbildung 6.1: Durchschnittlicher Prozentsatz der Knoten, die bereits zum Baum gehören, über die Anzahl der Kommunikationsphasen im Szenario mit 5 Knoten (1000 Durchläufe).

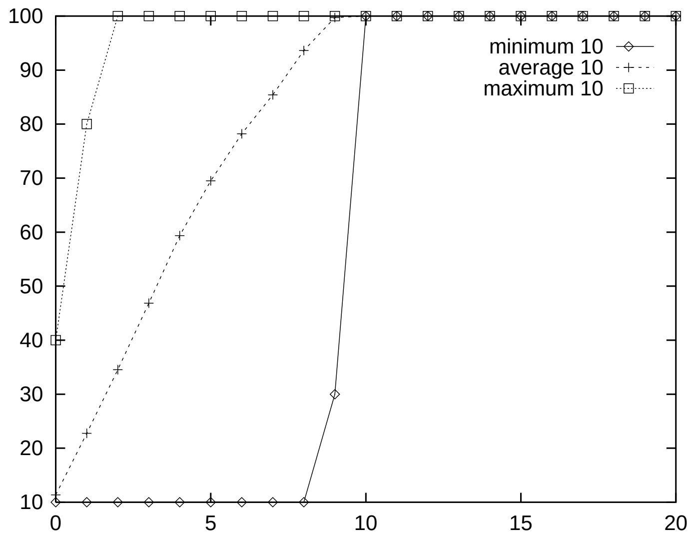

Abbildung 6.2: Durchschnittlicher Prozentsatz der Knoten, die bereits zum Baum gehören, über die Anzahl der Kommunikationsphasen im Szenario mit 10 Knoten (1000 Durchläufe).

Schlußfolgerungen für die Latenz. Für eine Beurteilung der Latenz seines Protokolls hat Golding die Zeit gemessen, die eine Nachricht brauchte, um auf alle Replikate zu gelangen. Da sowohl der hier vorgestellte Algorithmus als auch das Goldingsche $TSAE$ dieselben Simulationsannahmen und dieselben hier relevanten algorithmischen Abläufe enthalten, kann das Resultat seiner Simulationen auf den hier vorgelegten Algorithmus übertragen werden: Die relativ niedrigen Werte für die Baumtiefe führen dazu, daß Nachrichten auch bei steigender Knotenzahl schnell auf alle Replikate migrieren.

Analyse der Leitungsbelastung. Bei der Analyse der Leitungsbelastung geht ebenfalls die Struktur des verwendeten Baumes stark ein. Sowohl Goldings $TSAE$ als auch der hier entworfene Replikationsalgorithmus benutzen *flooding* als Verteilungsprinzip. Dies führt zu einer starken Belastung des Netzwerks, da pro versandter *broadcast*-Nachricht $O(e)$ Basisnachrichten versandt werden müssen, wobei e die Anzahl der Kanten im Kommunikationsnetzwerk darstellt. Durch die Verwendung selbststabilisierender Kommunikationsprotokolle auf den einzelnen Datenleitungen (vergleiche Algorithmus 4.11) ist es jedoch im hier präsentierten Fall möglich, neue *broadcast*-Nachrichten ähnlich wie in HPP über die Kanten des aufspannenden Baumes zu propagieren. Da es in herkömmlichen Kommunikationstopologien gewöhnlich mehr Kanten als Knoten gibt, resultiert diese Optimierung in einer auf $O(n)$ verminderten Nachrichtenkomplexität.

Die Belastung der einzelnen Kanten habe ich ebenfalls in einer Simu-

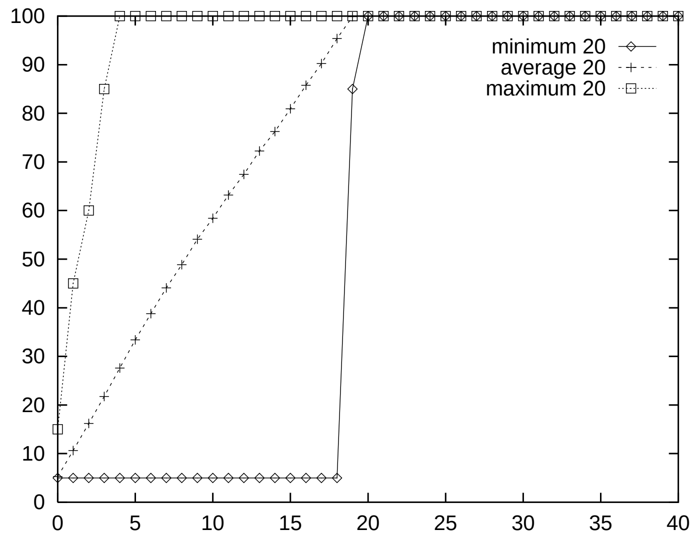

Abbildung 6.3: Durchschnittlicher Prozentsatz der Knoten, die bereits zum Baum gehören, über die Anzahl der Kommunikationsphasen im Szenario mit 20 Knoten (1000 Durchläufe).

lation untersucht. Dabei wurde zu einer beliebigen Kante v gemessen, in wievielen der n existierenden Bäumen sie enthalten ist. Der Durchschnitt über alle derartig berechneten Werte ergibt die "durchschnittliche Baumbelastung" l einer Kante. Wenn alle Knoten jeweils einen *broadcast*-Wert versenden, erzeugt dies pro Kante im Durchschnitt also $O(l)$ Nachrichten. Die Werte sind in Tabelle 6.3 aufgeführt. Dabei zeigt sich, daß die Last sich recht gut auf die Kanten verteilt. Der schlechteste Fall von n wird so gut wie nie erreicht. Abbildungen 6.4 bis 6.6 zeigen die durchschnittliche Anzahl von Knoten mit einer bestimmten Baumlast.

n	l	σ^2	σ	N
5	2	1,139	1,068	200
10	2	1,581	1,257	100
20	2	1,716	1,310	50
40	2	1,914	1,383	10

Tabelle 6.3: Durchschnittliche "Baumbelastung" l über N Szenarios pro Kante bei verschiedenen Knotenzahlen n mit Varianz (σ^2) und Standardabweichung (σ).

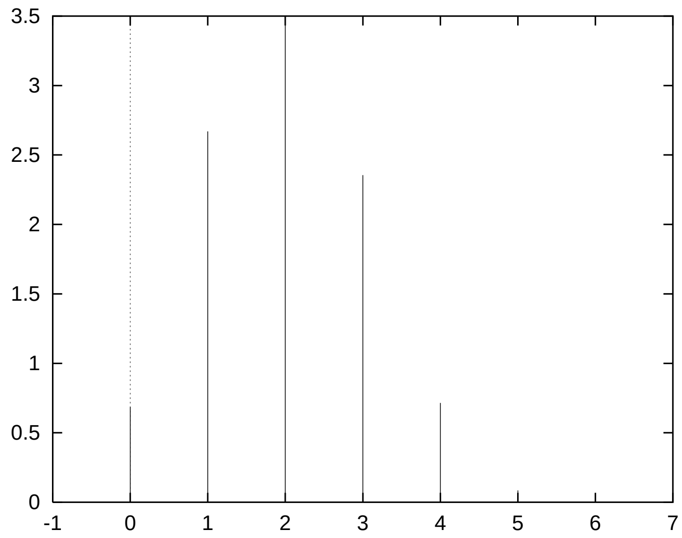

Abbildung 6.4: Verteilung der "Baumlast" (horizontale Achse) auf die Kanten (vertikale Achse, Durchschnitt über 200 Szenarien) für 5 Knoten. Die senkrechte Null-Linie (Anzahl der Kanten ohne "Baumlast") ist durch eine gepunktete Linie hervorgehoben.

6.2.2 Analyse der Fehlertoleranz

Durch die vorhandene Selbststabilisierungseigenschaft sind die Fähigkeiten des Algorithmus in Bezug auf seine Fehlertoleranz sicherlich die hervorstechendsten Merkmal des in dieser Arbeit präsentierten Algorithmus. Als einziger toleriert er *alle* Formen vorübergehender Fehler. Dies schließt den dauerhaften Ausfall von Knoten und Kommunikationsleitungen ein, solange das verbleibende Netzwerk noch zusammenhängend ist. Er hat damit deutlich bessere Fehlertoleranzeigenschaften als *TSAE*, *flood-d* und *HPP*.

Die Selbststabilisierung hat aber auch einige Eigenschaften zur Folge, die man normalerweise als "problematisch" bezeichnen würde. Das Hauptproblem dabei ist die nichtmaskierende Fehlertoleranz. Der realisierte zuverlässige *broadcast* ist eben nur solange zuverlässig, wie keine Fehler auftreten. Zwar braucht man sich nach Reparatur der Fehler nicht um den bei anderen Algorithmen oft notwendigen "Neustart" zu kümmern, doch kann man sich nie "sicher" sein, daß alles korrekt funktioniert hat.

Die zum Teil hohe Konvergenzzeit des Algorithmus, die ebenfalls durch die Selbststabilisierungseigenschaft hervorgerufen wird, sowie die Einschränkung der Schreibgeschwindigkeit der Prozesse, die Voraussetzung für die Korrektheit war (vergleiche Satz 5.16), sind weitere negative Faktoren.

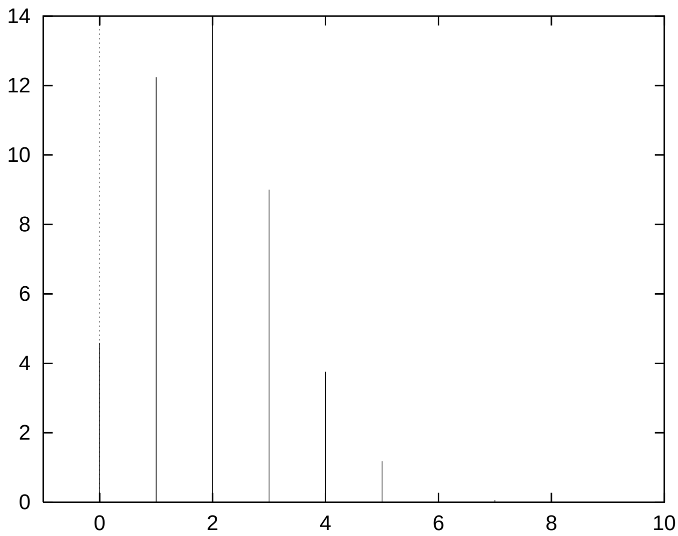

Abbildung 6.5: Verteilung der “Baumlast” (horizontale Achse) auf die Kanten (vertikale Achse, Durchschnitt über 100 Szenarien) für 10 Knoten. Die senkrechte Null-Linie (Anzahl der Kanten ohne “Baumlast”) ist durch eine gepunktete Linie hervorgehoben.

6.2.3 Mögliche Anwendungen

Die Analyse der Fehlertoleranzeigenschaften des hier vorgestellten Algorithmus deutet auf mögliche Anwendungsgebiete hin. Diese umfassen alle Domänen,

- in denen schwach-konsistente Replikation, bzw. ein zuverlässiger *broadcast* benötigt wird,
- die nur einen beschränkten Änderungs-, bzw. Kommunikationsbedarf besitzen,
- bei denen die Reparaturmöglichkeiten eingeschränkt sind und
- in denen wegen der nicht-maskierenden Fehlertoleranz die Korrektheit der Replikation, bzw. des *broadcast* allenfalls die Leistung der Anwendung beeinflußt und nicht deren Korrektheit.

Konkret fallen hierunter alle Formen von Diensten, die eine hohe Zuverlässigkeit haben sollen, für die aber aus Kostengründen bisher lediglich eine bestmögliche (*best effort*) Zustellung von Daten realisiert wurde. Beispiele sind verteilte Software-Archive oder elektronische Tageszeitungen.

Denkbar ist aber auch der Einsatz im Rahmen der Informationsverteilung auf autonome Systeme (Agenten) oder mobile Rechnereinheiten. Ge-

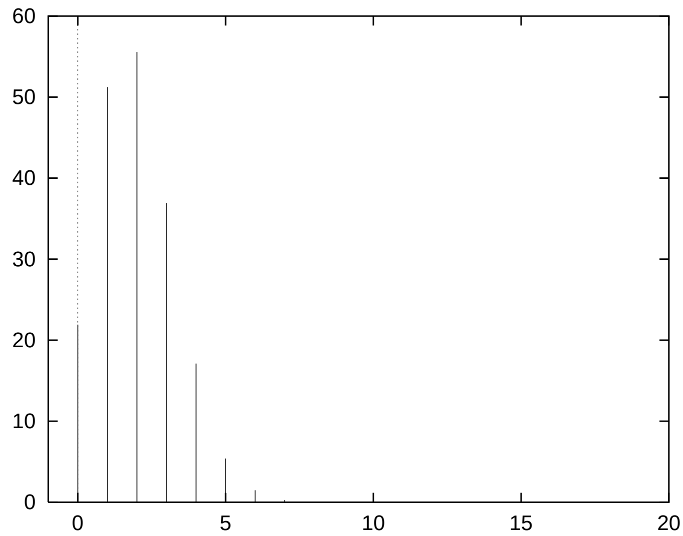

Abbildung 6.6: Verteilung der "Baumlast" (horizontale Achse) auf die Kanten (vertikale Achse, Durchschnitt über 50 Szenarien) für 20 Knoten. Die senkrechte Null-Linie (Anzahl der Kanten ohne "Baumlast") ist durch eine gepunktete Linie hervorgehoben.

rade diese Anwendungsgebiete haben stark mit ungünstigen Umgebungsbedingungen, mit ungewissen oder ständig wechselnden Umgebungseigenschaften, mit niedriger Bandbreite oder oft dauerhafter Abkopplung von bestehenden Festnetzen zu kämpfen. Hier scheinen die guten Fehlertoleranzeigenschaften des Algorithmus besonders benötigt zu werden.

Konkret gibt es heute schon Anwendungen, in denen die Verwendung des Verfahrens sinnvoll erscheint:

- Die Realisierung eines "Short Message Service – SMS" in Mobilfunksystemen.
  Derartige Dienste gestatten die kostenlose Übermittlung kurzer Texte an die Besitzer von Mobiltelefonen. Die Übertragung ist nicht zu 100 Prozent garantiert (und dafür kostenlos), soll aber trotzdem zuverlässig sein.

- Die Realisierung eines Zustelldienstes für die "Mobilbox."
  Viele Mobiltelefone erlauben heutzutage die Speicherung von gesprochenen oder textuellen Nachrichten im Speicher des Gerätes selbst. Falls kein Kontakt aus dem Festnetz zu den mobilen Geräten hergestellt werden kann, werden oft an zentraler Stelle Nachrichten entgegengenommen, die später ansynchron an die mobilen Teilnehmer weitervermittelt werden. Die Zustellung an den Teilnehmer soll dabei sehr zuverlässig sein, ist aber nicht zu 100 Prozent garantiert. Will man sicher wissen, welche Nachrichten für einen vorliegen, muß man sich noch an die zentrale Stelle wenden.

- Anwendungen im Rahmen autonomer Agenten.
  Da die momentane Entwicklung auf dem Gebiet der Internet-Agenten momentan sehr stürmisch verläuft, kann ich Anwendungen in diesem Bereich noch nicht absehen.

6.3 Ergebnisse in theoretischer Hinsicht

Die Ergebnisse in theoretischer Hinsicht liegen für mich im wesentlichen auf dem Gebiet der Selbststabilisierung. Ich habe meines Wissens erstmals auf Deutsch überblickshaft das Gebiet der Selbststabilisierung beschrieben und eine strukturierte Methodologie skizziert, die dem Wesen dieses Phänomens angepaßt erscheint. Diese Ergebnisse sind im Zusammenhang mit den Arbeiten von Arora, Gouda und Varghese [12] sowie Awerbuch, Patt-Schamir, Varghese und Dolev [17, 18] zu sehen. Das Spannungsfeld zwischen den Ansätzen dieser beiden Autorengruppen enthält viele Widersprüche und offene Fragen, die ich zwar nicht vollständig beantworten konnte, die aber trotzdem einmal gestellt werden mußten. Dazu gehören die Fragen nach der Existenz einer Methodenhierarchie und dem Verhältnis zu der bisher wenig beachteten Forschung auf dem Gebiet der globalen Prädikate in verteilten Systemen.

Als wesentliches "handfestes" Ergebnis verbleiben die entwickelten Algorithmen für Replikation und den zuverlässigen *broadcast*.

6.4 Ergebnisse in persönlicher Hinsicht

In persönlicher Hinsicht haben sich für mich zahlreiche Einsichten ergeben, die mir in dieser Form vorher (trotz einer guten Informatikausbildung) nicht bewußt waren. Die betrifft vor allem die Notwendigkeit, die Korrektheit der aufgestellten Theoreme formal nachzuweisen. Denn obwohl mich diese Teile der Arbeit mit Abstand die meiste Mühe und Anstrengung gekostet haben, wurde ich dadurch gezwungen, viele der implizit getroffenen Voraussetzungen explizit zu machen. Die vielen strukturellen Parallelen in Beweis und Algorithmus haben wieder in mir die Gewißheit gestärkt, daß es zwischen Beweisen und Programmen eine ganz enge Verbindung gibt [97]. Beweise "sind" Programme und Programme "sind" (unvollständige) Beweise.

Außerdem bin ich immer wieder (beispielsweise durch die Beschäftigung mit der Problematik von *timeout*-Zählern) auf die Probleme gestoßen, die aus dem Aufeinanderprallen von Theorie und Praxis erwachsen [118, Titelseite]. Trotzdem bleibt für mich die Einsicht, daß ein (semi-) formales Arbeiten manchmal unverzichtbar ist.

6.5 Weitere Arbeiten

Trotz der diversen Ergebnisse bleiben viele offene Fragen, mit denen man sich noch weiter auseinandersetzen könnte. In diesem Abschnitt, der diese Arbeit beschließt, will ich die meiner Meinung nach wichtigsten Punkte nennen und Arbeitsansätze aufzeigen.

Selbststabilisierung als Phänomen

Wie bereits erwähnt, ist das Thema "Selbststabilisierung" zu komplex, als daß es sich in einem einzigen Kapitel einer Diplomarbeit erschöpfend behandeln ließe. Dies liegt einerseits daran, daß die Informatik-Forschung dieses Gebiet in der Vergangenheit relativ stiefmütterlich behandelt hat und erst jetzt mit einer hektischen Euphorie zahllose neue Begriffe und Formalismen in den Raum stellt, die auf den ersten Blick unzusammenhängend und verwirrend wirken [18, 51, 69, 71, 77]. Andererseits ist die Tatsache, daß viele andere Wissenschaftsbereiche (unter Namen wie "Konvergenz", "Beruhigung" oder "Einschwingen") ähnliche Erscheinungen kennen, ein Hinweis darauf, daß es sich hierbei um ein ganz allgemeines Phänomen handelt, welches die Informatik erst in Teilen erfaßt und nutzbar gemacht hat.

Eine lohnende weitere Arbeit wäre also, aufbauend auf den Definitionen und Beispielen aus Kapitel 4 weitere Beschreibungen aus der Informatik und vor allem auch aus anderen Wissenschaftsbereichen (beispielsweise der Regelungstechnik) zu sichten und sie auf ein gemeinsames theoretisches Fundament zu stellen. Aber auch wenn dies nicht machbar ist, kann man versuchen, dem Phänomen der Selbststabilisierung (mit ihrer inneren Dynamik und den notwendigen Terminierungsbedingungen) möglichst nahe zu kommen.

Hierarchie der Verfahren = Hierarchie der Probleme?

Der zweite Punkt, dessen Untersuchung lohnenswert erscheint, ist die in Kapitel 4 vermutete Hierarchie der Verfahren. Die bisher zu diesem Thema veröffentlichten Artikel [3, 12, 17, 18, 140] enthalten meiner Meinung nach mehr Fragen als Antworten (und zum Teil sogar widersprüchliche Definitionen der lokalen Überprüfbarkeit [17, 18]). Die Methode der lokalen Überprüfung und Korrektur ist zwar schon auf wenige Teilprobleme angewandt worden [11, 12], doch fehlen notwendige und hinreichende Kriterien für die Aussage: "Problem A ist (nicht) lokal überprüfbar/korrigierbar!" Es sind sind noch zuwenige der Standardprobleme in verteilten Systemen unter diesem Aspekt untersucht worden. Möglicherweise verbirgt sich hinter der Hierarchie der Verfahren eine "Hierarchie der Probleme"? Mögliche weitere Forschung auf diesem Gebiet muß sich meiner Meinung nach an den neusten Erkenntnissen auf dem Gebiet der Erkennung globaler Prädikate orientieren [21, 61, 62, 64, 65, 91, 127, 132].

Die Arbeiten von Arora, Gouda und Varghese [11, 12, 140] deuten an, daß gerichtete azyklische Graphen eine wichtige Rolle bei bestimmten Formen der Selbststabilisierung spielen. Indizien hierfür sind der oft geäußerte Zusammenhang zwischen derartigen Graphen und der Eigenschaft der Terminierung sowie die Tatsache, daß viele selbststabilisierende Algorithmen auf Baumstrukturen aufsetzen [3, 10, 11, 53]. Eine genaue Untersuchung der Prinzipien von Algorithmen, die aufspannende Bäume in selbststabilisierender Weise berechnen, insbesondere eine Abschätzung ihrer Komplexität und etwaige Optimalitätsergebnisse stellen darum meiner Meinung nach wichtige Schritte dar auf dem Weg zu einem besseren Verständnis der Methodik. Hieraus könnten insbesondere Einsichten über die Mächtigkeit der Methode des lokalen Prüfens und der lokalen Korrektur.

Für den formalen Nachweis der vermuteten Hierarchie muß also noch viel Arbeit geleistet werden.

Weiterentwicklung, Optimierung und Implementierung der Algorithmen

Neben den eben geäußerten weiteren Arbeiten gibt es auch ganz andere, praktischere und “profanere” Aufgaben, die aus dieser Arbeit erwachsen. Dazu gehört natürlich die Optimierung und Implementierung der in dieser Arbeit beschriebenen Algorithmen in “echten” Umgebungen. Hier würde sich zeigen, ob die alleinige Konzentration auf die Selbststabilisierungeeigenschaft beim Entwurf tatsächlich die in Abschnitt 4.3.4 vermuteten hohe Kosten mit sich bringt.

In Kapitel 5 wurde behauptet, daß man aus dem vorgestellten zuverlässigen *broadcast* durch feste Transformationen [80] alle weiteren Formen von Gruppenkommunikationsmechansmen herleiten kann. Offen blieb dabei die Frage, ob die dabei notwendige Verwendung von (unbeschränkten) Sequenzummern zur Herstellung von Ordnungsrelationen Probleme in Bezug auf die Selbststabilisierung der Algorithmen mit sich bringt. Im Rahmen konkreter Implementierungen könnten hier Antworten gefunden werden.

Eine Idee, welche die Grundlage für weitere Arbeit sein könnte, fußt in der Beobachtung, daß schwache und starke Konsistenz eng verbunden sind mit maskierender und nicht-maskierender Fehlertoleranz. Interessant wäre also die Untersuchung, ob (und falls ja welche) Replikationsverfahren mit strenger Konsistenz aus einer systematischen Entwicklung von Replikationsalgorithmen mit maskierender Fehlertoleranz aus der Definition des Konsistenzproblems entstehen (vergleiche Bemerkung 5.2 auf Seite 122).

Schließlich bietet die in Abschnitt 5.4 besprochene Anwendung von

Lastausgleichsverfahren zur Kostenminimierung noch ein vielfältiges Betätigungsfeld. Die Lösungen, die dort in Bezug auf Replikation präsentiert wurden, waren für meinen Geschmack etwas künstlich konstruiert und dadurch unbefriedigend. Die magere Ausbeute ist verwunderlich, da sich das Problem der Kostenoptimierung von Replikationsverfahren bei "stabilem" Schreib-/Leseverhalten der Knoten auf den ersten Blick ganz deutlich als "Konvergenz-Problem" beschreiben läßt. Die Probleme, die bei meinen Überlegungen zutage traten, sind möglicherweise durch Einbeziehung von Erkenntnissen aus der Regelungstechnik lösbar. Dort sind bereits Modelle und Verfahren entwickelt worden, nach denen man das "Einschwingen" von Regelkreisen vorausberechnen kann. Es gilt, die Einflüsse auf das Systemverhalten genau zu analysieren, aus diesen Erkenntnissen zu lernen und sie in ein fundiertes algorithmisches Konzept zu gießen.

A

Beweise

A.1 Beweis von Lemma 5.5

Der Beweis bezieht sich auf Lemma 5.5 von Seite 127.

Beweis Der Beweis erfolgt durch Induktion über die Struktur des Baumes.

Induktionsverankerung. Betrachte ein triviales System mit einem Knoten, also einen Baum, der nur aus der Wurzel besteht. Die Wurzel schreibe einen Datenwert nach r_0. Trivialerweise hat nun nach einer Schreiboperation der Wurzel auf allen Knoten die "entsprechende" Schreiboperation stattgefunden.

Induktionshypothese. Betrachte ein System aus n Knoten und den darin enthaltenen Baum, für den die Behauptung aus Lemma 5.5 bereits gezeigt wurde. Das bedeutet, daß es nach n Schreiboperationen der Wurzel zu jeder solchen Schreiboperation ein entsprechendes Schreibereignis auf jedem Knoten innerhalb dieses Baumes gibt.

Induktionsschritt. Betrachte nun einen zusätzlichen Knoten i, der noch nicht Teil des Baumes ist, und füge ihn durch die Auswahl eines Elternknotens p zum Baum hinzu. Falls die letzte Schreiboperation auf p einen Wert nach r_p schrieb, der zufälligerweise gleich dem Wert r_i war, dann warte, bis das nächste Schreibereignis e der Wurzel auch in entsprechender Form auf Knoten p stattgefunden hat. Spätestens jetzt ist $r_i \neq r_p$ und die Wache $i \neq 0 \wedge r_i \neq r_{p_i}$ ist wahr. Nach Voraussetzung hält p den alten Wert so lange, bis i die zugehörige Aktion ausgeführt hat, was wegen der angenommenen Fairneß auch irgendwann eintreten muß. Das bedeutet, daß auf Knoten i nach spätestens $n+1$ Schreibereignissen der Wurzel auch das der Operation e entsprechende Schreibereignis stattgefunden hat.

Insgesamt hat also auf allen $n+1$ Knoten des Baumes nach spätestens $n+1$ Schreibereignissen der Wurzel die dem Schreibereignis e entsprechende Zuweisung stattgefunden. □

A.2 Beweis von Lemma 5.6

Der Beweis bezieht sich auf Lemma 5.6 von Seite 127 über die aktive Konsistenz des einfachen Replikationsalgorithmus aus Abbildung 5.2.

Beweis Zum Nachweis der Selbststabilisierung müssen zwei getrennte Teilbeweise geführt werden:

- Ein Beweis der Abgeschlossenheit von P und
- ein Beweis der Konvergenz zu P.

Dem Beweis liegt die Beobachtung zugrunde, daß sich die Werte bei fehlerfreier Abarbeitung des Algorithmus von der Wurzel her ausbreiten, wobei die gedachten Sequenznummern sequentiell die Knoten passieren.

Abgeschlossenheit von P. Gelte P. Zu zeigen ist, daß nach Abarbeitung einer beliebigen Anweisung des Algorithmus wieder P gilt.

Betrachte die Anweisung mit der Wache $i \neq 0 \wedge r \neq r_p$. Falls diese Wache "wahr" ist, gilt $r_i \neq r_{p_i}$ und somit wegen P auch $num(r_i) + 1 = num(r_{p_i})$. Nach Ausführung der Zuweisung $r_i := r_{p_i}$ nimmt r_i den Wert (und auch die Sequenznummer) von r_{p_i} an. Somit gilt nach der Aktion für die beiden Werte $r_i = r_{p_i} \wedge num(r_i) = num(r_{p_i})$. Das bedeutet aber, daß P wieder gilt.

Betrachte die zweite Anweisung, deren Wache mit $i = 0$ beginnt. Die dazugehörige Aktion wird also nur von der Wurzel ausgeführt. Da der Wert der Wurzel die Gültigkeit von P nicht beeinflußt, gilt P nach Ausführung der Aktion unverändert.

Konvergenz zu P. Betrachte einen beliebigen Knoten i. Nach Lemma 5.5 findet nach spätestens n Schreibereignissen der Wurzel ein der letzten Schreiboperation der Wurzel entsprechendes Schreibereignis auch auf i statt. Nach diesem Schreibereignis gilt $r_i = r_{p_i} \wedge num(r_{p_i}) = num(r_i)$ oder $r_i \neq r_{p_i} \wedge num(r_{p_i}) = num(r_i)+1$, falls auf dem Elternknoten bereits wieder ein Schreibereignis stattfand. Das bedeutet, daß nach spätestens n Schreiboperationen der Wurzel und einer endlichen zusätzlichen Zeit alle Knoten diese Bedingung erfüllen und somit P gilt. □

A.3 Beweis von Satz 5.9

Der Beweis bezieht sich auf Satz 5.9 von Seite 131 über die Korrektheit der in Abbildung 5.6 dargestellten Basisschicht zur Pflege der Nachbarschaftsbeziehungen.

Beweis Um die Selbststabilisierung nachzuweisen, müssen zwei Teilbeweise geführt werden:

- Ein Beweis der Abgeschlossenheit des Legalitätsprädikates P und
- einen Beweis der Konvergenz zum Prädikat P.

Abgeschlossenheit von P. Gelte P. Zu zeigen ist, daß jede Aktion des Algorithmus P unverändert läßt. Betrachte zuerst die Aktion mit der Wache $j \in N \wedge \mathbf{timeout}_j$. Wenn diese Wache "wahr" wird, ist der j-te **timeout**-Zähle ausgelaufen. Dies passiert auf Knoten i nach Voraussetzung nur, wenn zwischen den Knoten i und j keine Nachricht mehr unterwegs ist. Das bedeutet aber, daß Knoten j ein toter Nachbar ist, und da P gilt, ist $j \notin \mathit{neighbours}$. Darum bleibt die Menge *neighbours* durch die Zuweisung unverändert. Die Anweisung $\mathbf{out}(j, i)$ setzt zudem auf einen Kanal zwischen einem toten und einem lebenden Nachbarn genau eine Nachricht, so daß P weiterhin gültig ist.

Betrachte jetzt die zweite Aktion mit der Wache $\mathbf{in}(i, j)$. Dies ist genau dann wahr, wenn auf Knoten i eine Nachricht von Knoten j angekommen ist. Das bedeutet aber, daß j ein lebender Nachbar von i ist, also $j \in \mathit{neighbours}$, und durch die Hinzunahme von j ändert sich die Menge *neighbours* nicht. Wegen der Gültigkeit von P war vorher genau eine Nachricht auf diesem Kanal unterwegs. Durch die Anweisung $\mathbf{out}(j, i)$ wird wieder genau eine Nachricht auf den Weg gebracht, so daß P wieder gilt.

Konvergenz von P. Es bleibt zu zeigen, daß nach endlicher Zeit garantiert P erreicht wird. Betrachte hierzu für jeden Knoten i die folgende Terminierungsfunktion t_i:

$$\begin{aligned} t_i = {} & |\{j \in \mathit{Nodes} | j \notin \mathit{neighbours}_i \wedge j \text{ ist lebender Nachbar von } i\}| + \\ & |\{j \in \mathit{Nodes} | j \in \mathit{neighbours}_i \wedge j \text{ ist toter Nachbar von } i\}| \end{aligned}$$

Betrachte wieder die erste Aktion mit der Wache $j \in N \wedge \mathbf{timeout}_j$. Wenn diese Wache gilt, ist Knoten j ein toter Nachbar. Falls fälschlicherweise $j \in \mathit{neighbours}$, wird er durch die folgende Zuweisung entfernt und t_i nimmt um 1 ab (Fall 1). Falls $j \notin \mathit{neighbours}$ ändert sich t_i nicht (Fall 2).

Betrachte die zweite Aktion mit der Wache $\mathbf{in}(i, j)$. Wenn diese Wache "wahr" ist, ist Knoten j ein lebender Nachbar. Falls eh schon $j \in$

neighbours, ändert sich t_i nicht (Fall 3). Falls $j \notin$ *neighbours*, nimmt t_i um 1 ab (Fall 4).

Es fehlt nur noch zu zeigen, daß unter der Voraussetzung $t_i \neq 0$ die Fälle 2 und 3, in denen sich t_i nicht ändert, nicht unendlich oft nacheinander auftreten können, ohne daß zwischendurch die Fälle 1 und 4 auftreten müssen.

Sei also $t_i \neq 0$. Das bedeutet, es existiert noch mindestens ein toter Nachbar j von Knoten i mit $j \in$ *neighbours* oder es existiert noch mindestens ein lebender Nachbar j von i mit $j \notin$ *neighbours*.

Fall (a): Es existiert noch mindestens ein toter Nachbar j von Knoten i mit $j \in$ *neighbours*. Da j ein toter Nachbar von i ist, gehen auf dem Kanal zwischen i und j alle Nachrichten verloren. Das bedeutet, daß nach einer festen Zeit keine Nachrichten mehr zwischen i und j unterwegs sind und Fall 1 eintritt.

Fall (b): Es existiere noch mindestens ein lebender Nachbar j von i mit $j \notin$ *neighbours*. Falls keine Nachricht zwischen i und j unterwegs ist, tritt Fall 2 ein und t_i ändert sich wie oben gesehen nicht. In diesem Fall wird allerdings eine Nachricht von i nach j geschickt, die dort nach endlicher Zeit ankommt und von der anderen Seite zurückgeschickt wird. Spätestens jetzt tritt Fall 4 ein. Falls von vornherein noch eine Nachricht zwischen i und j unterwegs ist, folgt Fall 4 mit der gleichen Argumentation. □

A.4 Beweis von Satz 5.10

Der Beweis bezieht sich auf Satz 5.10 von Seite 134 über die Korrektheit des Algorithmus zur Pflege des aufspannenden Baumes aus Abbildung 5.7.

Beweis Der folgend vorgestellte Beweis benutzt Satz 4.23 und damit die auf Seite 104 angesprochene Beweistechnik der Konvergenzleiter.

Beobachtet man den Algorithmus in Aktion, so sieht man, daß sich der aufspannende Baum von der Wurzel her stabilisiert. Einmal dazugehörige Knoten bleiben zugehörig und deren Nachbarknoten schließen sich "von außen" dem bereits bestehenden Baum an, wenn ihre "Sicht" dies zuläßt. Die Folge von Prädikaten, die für die Konvergenzleiter angegeben werden muß, beginnt also mit einem trivialen Baum (bestehend nur aus der Wurzel) und wächst quasi mit der Anzahl der Knoten, die zum Baum gehören. Dabei hilft die Beobachtung, daß ein Knoten zunächst einmal die korrekte "Sicht" seines Nachbarn haben muß, bevor er sich dem Baum anschließen kann. Die Beweisidee entstammt einem Artikel von Arora und Gouda [10, S. 1037], in dem sie ihren eigenen Spannbaumalgorithmus vorstellen und den entsprechenden Korrektheitsnachweis skizzieren.

Zunächst ein paar Definitionen und abkürzende Schreibweisen: Das (rekursive) Prädikat $T(x)$ sei wahr für alle Knoten, die bereits zum in

Knoten 0 verwurzelten Baum gehören, also:

$$\begin{aligned}T(i) \equiv\ & (i = 0 \Rightarrow (p_i = 0 \wedge l_i = 0)) \wedge \\ & (i \neq 0 \Rightarrow (\exists j \in \mathit{neighbours}_i : p_i = j \wedge l_j = l_i + 1 \wedge T(j))) \wedge \\ & (\forall j \in \mathit{neighbours}_i : \tilde{l}_j[i] = l_i) \wedge \\ & (\forall \langle i, j, k\rangle \in M : k = l_j)\end{aligned}$$

Das Legalitätsprädikat läßt sich somit abkürzend schreiben als:

$$P \equiv \forall i \in \mathit{Nodes} : T(i)$$

Definiere die Menge *Tree* als die Menge aller Knoten, die bereits zum in Knoten 0 verwurzelten Baum gehören, also:

$$\mathit{Tree} = \{i \in \mathit{Nodes} | T(i)\}$$

Die Konvergenzleiter lautet dann wie folgt:

- $P_0 \equiv \mathit{true}$
- $P_1 \equiv T(0)$
- $P_2 \equiv |\mathit{Tree}| \geq 2$
- $P_k \equiv |\mathit{Tree}| \geq k$
- $P_n \equiv |\mathit{Tree}| \geq n$

Beweisverpflichtungen. Gemäß Satz 4.23 müssen drei Eigenschaften gezeigt werden:

1. $P_0 = \mathit{true}$ und $P_n = P$,
2. jedes P_k ist bei normaler Ausführung des Algorithmus abgeschlossen, und
3. Jede Ausführung, die mit einem Zustand in P_k beginnt, hat einen Zustand in P_{k+1} (für $k < n$).

Der Nachweis von Punkt 1 ist einfach einzusehen, da $P_0 \equiv \mathit{true}$ vorausgesetzt wird und man aus $|\mathit{Tree}| \geq n$ folgern kann, daß alle Knoten ein Element von *Tree* sind und somit für alle Knoten i das Prädikat $T(i)$ gilt.

Nachweis der Abgeschlossenheit (Punkt 2). Das Prädikat $P_1 \equiv T(0)$ ist trivialerweise abgeschlossen, da die Wurzel konstante Werte für p und l hat und nur die Wurzel die Sicht der Nachbarn von ihr selbst (also $\tilde{l}_j[0]$) beeinflussen kann (dritter Teilterm von T). Die Wurzel wird aber nur "korrekte" Nachrichten $\langle i, 0, 0\rangle$ verschicken, somit bleibt der vierte Teilterm ebenfalls gültig.

Gelte $P_k \equiv |Tree| \geq k$ für $k > 1$. Zu zeigen ist, daß P_k wieder gilt, egal welche Aktion ausgeführt wird. Betrachte also nun einen beliebigen Knoten i. Zunächst muß unterschieden werden, ob $i \in Tree$ oder nicht.

Fall 1: Sei also $i \in Tree$. Das bedeutet, daß auch $T(p_i)$ gilt, der Elternknoten auch zum Baum gehört. Da P_k gilt und $i \in Tree$, hat i eine korrekte Sicht seines Elternknotens ($\tilde{l}_i[p_i] = l_{p_i}$). Darum wird die erste Aktion nie ausgeführt werden. Da $p_i \in Tree$, ist $l_{p_i} = \tilde{l}_i[p_i] < n$ und somit wird die zweite Aktion auch nicht ausgeführt. Da $i \in Tree$ ist $l_i < n$ und somit wird die dritte Aktion nie ausgeführt. Die vierte Aktion wird jedoch dann ausgeführt, wenn die Wache **timeout** "wahr" wird. Das passiert nur, wenn keine Nachrichten an i unterwegs sind. Eine Ausführung von **shout**(l_i) bringt nur korrekte Nachrichten in das System, P wird also nicht falsifiziert.

In der fünften Aktion muß man unterscheiden, ob die eingegangene Nachricht von einem Knoten aus *Tree* kam oder nicht. Falls $j \in Tree$, dann hatte i gemäß P_k bereits eine korrekte Sicht von l_j und die Zuweisung $\tilde{l}_i[j] := k$ ändert daran nichts. Falls $j \notin Tree$, dann spielt die mögliche Werteänderung für die Gültigkeit von P_k keine Rolle.

Fall 2: Sei auf der anderen Seite $i \notin Tree$. Dann darf der Knoten i machen, was er will: P_k bezieht seinen Wahrheitswert einzig und allein aus dem Wirken von k Knoten, die bereits zur Menge *Tree* gehören, und so kann Knoten i daran nichts ändern.

Insgesamt gilt also P_k wieder, wenn P_k vorher gegolten hat. Damit ist P_k bei normaler Ausführung des Algorithmus abgeschlossen.

Konvergenz von P_k zu P_{k+1}. Gelte P_k. Im Beweis der Abgeschlossenheit wurde bereits gezeigt, daß Knoten, die schon zur Menge *Tree* gehören, nur durch den Versand von "korrekten" Nachrichten das restliche System beeinflussen. Man braucht also nur noch Knoten zu betrachten, die nicht in *Tree* enthalten sind.

Sei also i ein beliebiger Knoten mit $i \notin Tree$. Ohne Beschränkung der Allgemeinheit sei dies ein Nachbar eines Knotens j, der bereits Teil des Baumes ist, also $j \in neighbours_i$. Da P_k gilt, sind nur korrekte Nachrichten von j nach i unterwegs. Wenn eine davon angekommen ist, hat i eine korrekte Sicht von j, also $\tilde{l}_i[j] = l_j$. Jetzt muß unterschieden werden, ob i im "Fehlerzustand" mit $l_i = n$ ist oder nicht.

Falls $l_i = n$, dann ist die Wache $j \in neighbours_i \wedge l_i = n \wedge \tilde{l}_i[j] < n-1$ wahr und gemäß der vorausgesetzten Fairneß wird diese Aktion nach endlicher Zeit ausgewählt. Durch die Zuweisungen $l_i := \tilde{l}_i[j] + 1$ und $p_i := j$ werden die drei ersten Bedingungen des Prädikates $T(i)$ wahr. Durch die Anweisung **shout**(l_i) werden nach endlicher Zeit nur noch "korrekte" Nachrichten im System vorhanden sein und $T(i)$ gilt. Das bedeutet aber, daß $i \in Tree$ und somit gilt $|Tree| \geq k+1 \equiv P_{k+1}$. $\square$

A.5 Beweis von Satz 5.16

Dieser Beweis bezieht sich auf Satz 5.16 von Seite 144 über die Korrektheit des zuverlässigen *broadcast* (Algorithmus 5.15).

Beweis Da hier die alternative Selbststabilisierungsdefinition von Tel (Definition 4.5) verwendet wurde, müssen nach der Angabe einer Menge L von "legalen" Zuständen zwei getrennte Teile nachgewiesen werden:

1. Korrektheit: Jede Ausführung, die in L beginnt, ist ein zuverlässiger *broadcast* gemäß Definition 3.5.

2. Konvergenz: Jede Ausführung enthält eine Konfiguration aus L.

Die Menge L sind im Zusammenhang dieses Beweises die Zustände, in denen für alle n Teilalgorithmen das Prädikat der aktiven Konsistenz aus Lemma 5.6 von Seite 127 gilt.

Im Verlaufe des Beweises dürfen die Korrektheitsergebnisse der verwendeten Teilalgorithmen benutzt werden. Die Korrektheit des zusammengesetzten Algorithmus ergibt sich aus der Anwendung des Kompositionstheorems 4.22. Die Anwendbarkeit des Satzes wird der Vollständigkeit halber zuerst betrachtet.

Anwendbarkeit des Kompositionstheorems 4.22. Ich zeige, daß der momentan untersuchte Algorithmus 5.15 einer n-fachen Überlagerung des einfachen Replikationsalgorithmus 5.4 entspricht.

Der einfache Replikationsalgorithmus enthielt für den Wurzelknoten k die Anweisung:

$$\{\text{* update wanted *}\} \quad \longrightarrow \quad r := \{\text{* new value *}\}$$

und für einen anderen Knoten i die Zeile:

$$i \neq k \wedge r \neq r_p \quad \longrightarrow \quad r := r_p$$

In einem Prozeßkörper zusammengefaßt ergibt dies die folgenden beiden Zeilen:

$$\begin{array}{lll}
\multicolumn{3}{l}{\textbf{begin}} \\
\quad i = k \wedge \{\text{* update wanted *}\} & \longrightarrow & r := \{\text{* new value *}\} \\
[\!] \; i \neq k \wedge r \neq r_p & \longrightarrow & r := r_p \\
\multicolumn{3}{l}{\textbf{end}}
\end{array}$$

Die Variablen r und p der einzelnen Algorithmeninstanzen sollen nun bezüglich der jeweilien Wurzel innerhalb der Notation unterschieden werden. Dabei bezeichnet $x_i[k]$ die Variable x aus dem Algorithmus mit

Wurzel k auf Knoten i. Mit Hilfe dieser Notation kann man nun alle Algorithmeninstanzen innerhalb *eines* Prozeßkörpers von einem beliebigen Knoten i wie folgt darstellen:

$$
\begin{array}{lll}
\textbf{begin} & & \\
\quad i = 1 \wedge \{\text{* update wanted *}\} & \longrightarrow & r_i[1] := \{\text{* new value *}\} \\
\| \; i \neq 1 \wedge r_i[1] \neq r_{p_i[1]}[1] & \longrightarrow & r_i[1] := r_{p_i[1]}[1] \\
\| \; i = 2 \wedge \{\text{* update wanted *}\} & \longrightarrow & r_i[2] := \{\text{* new value *}\} \\
\| \; i \neq 2 \wedge r_i[2] \neq r_{p_i[2]}[2] & \longrightarrow & r_i[2] := r_{p_i[2]}[2] \\
\| \; \vdots & \longrightarrow & \vdots \\
\| \; i = n \wedge \{\text{* update wanted *}\} & \longrightarrow & r_i[n] := \{\text{* new value *}\} \\
\| \; i \neq n \wedge r_i[n] \neq r_{p_i[n]}[n] & \longrightarrow & r_i[n] := r_{p_i[n]}[n] \\
\textbf{end} & &
\end{array}
$$

Man beachte, daß ein jeder Teil, der einer eigenen Instanz des Algorithmus entspricht, nur Variablen seiner eigenen Instanz verändert.

In derselben notationellen Kurzform, wie sie bereits in Kapitel 4 (Beispielsweise bei Algorithmus 4.10) verwendet wurde, kann man diesen Block wie folgt vereinfachen:

$$
\begin{array}{lll}
\textbf{begin} & & \\
\quad j \in \{1, \ldots, n\} \wedge i \neq j \wedge r_i[j] \neq r_{p_i[j]}[j] & \longrightarrow & r_i[j] := r_{p_i[j]}[j] \\
\| \; i = i \wedge \{\text{* update wanted *}\} & \longrightarrow & r_i[i] := \{\text{* new value *}\} \\
\textbf{end} & &
\end{array}
$$

Wie man sieht, entspricht dies genau der Formulierung von Algorithmus 5.15 aus Abbildung 5.11 auf Seite 143.

Nachweis der Korrektheit. Zum Nachweis der Korrektheit muß gezeigt werden, daß jede Ausführung, die in L beginnt, die drei Eigenschaften (*validity*, *agreement* und *integrity*) des zuverlässigen *broadcast* hat.

Bedingung der *validity* ("Wenn ein korrekter Prozeß **broadcast**(m) aufruft, dann werden alle anderen Prozesse garantiert **deliver**(m) aufrufen."): Ein Prozeß k rufe **broadcast**(m) auf. Da sich das Gesamtsystem und somit insbesondere auch der k-te Teilalgorithmus in einem aktivkonsistenten Zustand befindet, wird es nach Lemma 5.7 zu dem "Schreibereignis" **broadcast**(m) garantiert ein entsprechendes Schreibereignis auf den Replikationsvariablen aller Prozesse geben. Dies hat dann zur Folge, daß **deliver**(m) aufgerufen wird.

Bedingung des *agreement* ("Wenn ein korrekter Prozeß **deliver**(m) ausführt, dann werden alle anderen korrekten Prozesse garantiert **deliver**(m) ausführen."): Nach Lemma 5.5 gilt die Folgerung der "entsprechenden" Ereignisse auch rückwärts, d.h. wenn **deliver**(m) aufgerufen wurde, ist auch ein entsprechendes Schreibereignis auf der Wurzel geschehen. Nach der *validity* geschieht darum auch ein Aufruf von **deliver**(m) auf jedem erreichbaren Knoten.

Bedingung der *integrity* ("Nur Nachrichten, auf denen zuvor **broadcast**(m) ausgeführt wurde, werden tatsächlich – und zwar maximal ein Mal – zugestellt."): Die Bedingungen der *validity* und des *agreement* zeigen die Äquivalenz von einem Aufruf von **broadcast**(m) auf einem Knoten und dem Aufruf von **deliver**(m) auf allen Knoten. Bleibt zu zeigen, daß zu einem Aufruf von **broadcast**(m) der Aufruf von **deliver**(m) nur genau ein Mal erfolgt. Dies wäre nur möglich, wenn zweimal hintereinander die Zuweisung $r_i := r_{p_i}$ mit dem gleichen Wert r_{p_i} erfolgen würde. Dies ist aber durch die Wache $r_i \neq r_{p_i}$ ausgeschlossen.

Nachweis der Konvergenz. Nach Lemma 5.6 stabilisieren sich die n einzelnen Algorithmen zu aktiver Konsistenz. Da die Algorithmen sich gegenseitig nicht beeinflussen, stabilisiert sich das Gesamtsystem zur aktiven Konsistenz aller Teilalgorithmen. □

Literaturverzeichnis

[1] ADLY, N., AND KUMAR, A. HPP: a hierarchical propagation protocol for large scale replication in wide area networks. In *ICCI95 Seventh International Conference on Computing and Information* (July 1995).

[2] AFEK, Y., AND BROWN, G. M. Self-stabilization over unreliable communication media. *Distributed Computing 7* (1993), 27–34.

[3] AFEK, Y., KUTTEN, S., AND YUNG, M. Memory-efficient self-stabilization on general networks. In *WDAG90 Distributed Algorithms 4th International Workshop Proceedings, Springer-Verlag LNCS:486* (1990), pp. 15–28.

[4] ALPERN, B., AND SCHNEIDER, F. B. Defining liveness. *Information Processing Letters 21* (1985), 181–185.

[5] ANAGNOSTOU, E., AND HADZILACOS, V. Tolerating transient and permanent failures. In *WDAG93 Distributed Algorithms 7th International Workshop Proceedings, Springer-Verlag LNCS:725* (1993), pp. 174–188.

[6] ANTONOIU, G., AND SRIMANI, P. A self-stabilizing distributed algorithm to construct an arbitrary spanning tree of a connected graph. *Computers and Mathematics with Applications 30* (1995), 1–7.

[7] ARORA, A., ATTIE, P., EVANGELIST, M., AND GOUDA, M. Convergence of iteration systems. *Distributed Computing 7* (1993), 43–53.

[8] ARORA, A., AND GOUDA, M. Closure and convergence: a formulation of fault-tolerant computing. In *FTCS92 Proceedings of the 22nd International Symposium on Fault-Tolerant Computing Systems* (1992), IEEE Computer Society Press, pp. 396–403.

[9] ARORA, A., AND GOUDA, M. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering 19*, 11 (1993), 1015–1027.

[10] ARORA, A., AND GOUDA, M. G. Distributed reset. *IEEE Transactions on Computers 43*, 9 (September 1994), 1026–1038.

[11] ARORA, A., AND GOUDA, M. G. Load balancing: an exercise in constrained convergence. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972* (1995), pp. 183–197.

[12] ARORA, A., GOUDA, M. G., AND VARGHESE, G. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. In *ICDCS94 Proceedings of the 14th International Conference on Distributed Computing Systems* (1994), pp. 424–431.

[13] ARORA, A., AND KULKARNI, S. S. Designing masking fault-tolerance via nonmasking fault-tolerance. In *SRDS95 Proceedings of the 14th Symposium on Reliable Distributed Systems* (1995), pp. 174–185.

[14] ARORA, A. K. *A foundation of fault-tolerant computing.* PhD thesis, The University of Texas at Austin, December 1992.

[15] ARORA, R., AND GUPTA, M. More comments on "Distributed termination detection algorithm for distributed computations" (Letter to the Editor). *Information Processing Letters 29* (September 1988), 53–55.

[16] ARORA, R., RANA, S., AND GUPTA, M. Distributed termination detection algorithm for distributed computations. *Information Processing Letters 22* (May 1986), 311–314.

[17] AWERBUCH, B., PATT-SHAMIR, B., AND VARGHESE, G. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (1991), pp. 268–277.

[18] AWERBUCH, B., PATT-SHAMIR, B., VARGHESE, G., AND DOLEV, S. Self-stabilizing by local checking and global reset. In *WDAG94 Distributed Algorithms 8th International Workshop Proceedings, Springer-Verlag LNCS:857* (1994), pp. 326–339.

[19] BAAN, H. R. Simulating networks — Netsim system concepts. `ftp:ftp.cs.ruu.nl/pub/UNIX/netsim.zip`, June 1994.

[20] BAAN, H. R. Simulating networks — Netsim user manual. `ftp:ftp.cs.ruu.nl/pub/UNIX/netsim.zip`, June 1994.

[21] BABAOĞLU, O., AND MARZULLO, K. Consistent global states of distributed systems: fundamental concepts and mechanisms. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison-Wesley, 1993, ch. 4, pp. 55–96.

[22] BARNES, J. G. P. *Programmieren in Ada.* Carl Hanser Verlag, München, 1983.

[23] Baumgarten, B. *Petri-Netze – Grundlagen und Anwendungen.* BI-Wissenschaftsverlag, Mannheim, 1990.

[24] Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[25] Beuter, T., and Dadam, P. Prinzipien der Replikationskontrolle in verteilten Systemen. Tech. Rep. Ulmer Informatik-Berichte Nr. 95-11, Universität Ulm, Fakultät für Informatik, 1995.

[26] Birman, K., and van Renesse, R. Software für zuverlässige netzwerke. *Spektrum der Wissenschaft* (September 1997), 76–81.

[27] Birman, K. P., and Glade, B. B. Reliability through consistency. *IEEE Software* (May 1995), 29–41.

[28] Birman, K. P., and van Renesse, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[29] Bourgon, B., Das, S., Datta, A., and Natarajan, V. A self-stabilizing shortest path algorithm in a DAG. In *Proceedings of the IEEE 14th Annual International Phoenix Conference on Computers and Communication* (1995), pp. 341–345.

[30] Brown, G. M., Gouda, M. G., and Wu, C.-L. Token systems that self-stabilize. *IEEE Transactions on Computers 38*, 6 (June 1989), 845–852.

[31] Burns, J., Gouda, M., and Miller, R. Stabilization and pseudo-stabilization. *Distributed Computing 7* (1993), 35–42.

[32] Burns, J., and Pachl, J. Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems 11* (1989), 330–344.

[33] Chandra, T., and Toueg, S. Unreliable failure detectors for asynchronous systems. In *PODC91 Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing* (1992), pp. 325–340.

[34] Chandrasekar, S., and Srimani, P. A self-stabilizing algorithm to synchronize digital clocks in a distributed system. *Computers and Electrical Engineering 20*, 6 (1994), 439–444.

[35] Chandy, K. M., and Lamport, L. Distributed snapshots: determining global states of distributed systems. *ACM Transations on Computing Systems 3*, 1 (1985), 63–75.

[36] Chang, E., Gonnet, G., and Rotem, D. On the costs of self-stabilization. *Information Processing Letters 24* (1987), 311–316.

[37] Charron-Bost, B. Concerning the size of logical clocks in distributed systems. *Information Processing Letters 39*, 1 (July 1991), 11–16.

[38] Charron-Bost, B., Mattern, F., and Tel, G. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing 9* (1996), 173–191.

[39] Chen, N.-S., Yu, H.-P., and Huang, S.-T. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters 39* (1991), 147–151.

[40] Cin, M. D. *Fehlertolerante Systeme*, vol. 50 of *Leitfäden der angewandten Mathematik und Mechanik.* Teubner, 1979.

[41] Collin, Z., and Dolev, S. Self-stabilizing depth-first search. *Information Processing Letters 49* (1994), 297–301.

[42] Cooper, R., and Marzullo, K. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (1991), pp. 163–173.

[43] Costello, A. M., and Varghese, G. Self-stabilization by window washing. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (1996), pp. 35–44.

[44] Cristian, F. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering SE-11*, 1 (January 1985), 23–31.

[45] Cristian, F. Understanding fault-tolerant distributed systems. *Communications of the Association of the Computing Machinery 34*, 2 (February 1991), 56–78.

[46] Dadam, P. *Verteilte Datenbanken und Client/Server-Systeme.* Springer, 1996.

[47] Dijkstra, E. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the Association of the Computing Machinery 18*, 8 (August 1975), 453–457.

[48] Dijkstra, E. W. Co-operating sequential processes. *Communications of the Association of the Computing Machinery 11*, 5 (1968), 341–346.

[49] Dijkstra, E. W. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery 17*, 11 (1974), 643–644.

[50] Dijkstra, E. W. A belated proof of self-stabilization. *Distributed Computing 1* (1986), 5–6.

[51] Dolev, S., and Herman, T. Superstabilizing protocols for dynamic distributed systems. In *Proceedings of the Second Workshop on Self-Stabilizing Systems* (1995), pp. 3.1–3.15.

[52] Dolev, S., Israeli, A., and Moran, S. Resource bounds for self stabilizing message driven protocols. In *PODC91 Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing* (1991), pp. 281–293.

[53] Dolev, S., Israeli, A., and Moran, S. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing 7* (1993), 3–16.

[54] Dolev, S., and Welch, J. Self-stabilizing clock synchronization in the presence of byzantine faults. In *Proceedings of the Second Workshop on Self-Stabilizing Systems* (1995), pp. 9.1–9.12.

[55] Echtle, K. *Fehlertoleranzverfahren.* Springer Verlag, 1990.

[56] Fischer, M. L., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the Association of the Computing Machinery 32*, 2 (April 1985), 374–382.

[57] Flanagan, D. *Java in a Nutshell*, second ed. O'Reilly & Associates, 1997.

[58] Flatebo, M., Datta, A., and Ghosh, S. Self-stabilization in distributed systems. In *Readings in Distributed Computing Systems.* IEEE Computer Society Press, 1994, ch. 2, pp. 100–114. TL Casavant and M Singal, Editors.

[59] Flatebo, M., Datta, A., and Schoone, A. Self-stabilizing multi-token rings. *Distributed Computing 8* (1994), 133–142.

[60] Fontane, T. *Cécile.* Ullstein, 1996.

[61] Garg, V. K., and Chase, C. M. Distributed algorithms for detecting conjunctive predicates. In *Proceedings of the 15th International Conference on Distributed Computing Systems* (June 1995), pp. 423–430.

[62] Garg, V. K., Chase, C. M., Mitchell, J. R., and Kilgore, R. Conjunctive predicate detection. In *Proceedings Hawaii International Conference on System Sciences HICSS95* (January 1995), IEEE Computer Society.

[63] Garg, V. K., and Waldecker, B. Detection of unstable predicates in distributed programs. In *FSTTCS92 Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science* (December 1992), R. Shyamasundar, Ed.,

no. 652 in Lecture Notes in Computer Science, Springer Verlag, pp. 253–264.

[64] GARG, V. K., AND WALDECKER, B. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems 5*, 3 (1994), 299–307.

[65] GARG, V. K., AND WALDECKER, B. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems 7*, 12 (1996), 1323–1333.

[66] GÄRTNER, F. Building a language independent prettyprinter. Studienarbeit am Fachgebiet Systemprogrammierung des Fachbereichs Informatik der Technischen Hochschule Darmstadt, 1994.

[67] GÄRTNER, F. The Pretzel Homepage. Internet: `http://www.iti.informatik.tu-darmstadt.de/~gaertner/-pretzel.html`, 1997.

[68] GELERNTER, D. Das Programmieren modernster Computer. *Spektrum der Wissenschaft* (December 1987), 74–82.

[69] GHOSH, S., GUPTA, A., HERMAN, T., AND PEMMARAJU, S. Fault-containing self-stabilizing algorithms. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (1996), pp. 45–54.

[70] GHOSH, S., GUPTA, A., AND PEMMARAJU, S. A fault-containing self-stabilizing algorithm for spanning trees. *Journal of Computing and Information 2* (1996), 322–338.

[71] GHOSH, S., AND PEMMARAJU, S. V. Tradeoffs in fault-containing self-stabilization. In *Proceedings of the Third Workshop on Self-Stabilizing Systems* (1997), Carleton University Press, pp. 157–169.

[72] GOLDING, R. A. Accessing replicated data in a large-scale distributed system. Master's thesis, University of California, Santa Cruz, June 1991. Also published as Technical Report UCSC-CRL-91-18.

[73] GOLDING, R. A. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, December 1992. Also published as Technical Report UCSC-CRL-92-52.

[74] GOPAL, A. S., AND PERRY, K. J. Unifying self-stabilization and fault-tolerance. In *PODC93 Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing* (1993), ACM Press, pp. 195–206.

[75] GOUDA, M. Stabilizing observers. *Information Processing Letters 52* (1994), 99–103.

[76] GOUDA, M., AND EVANGELIST, M. Convergence/response tradeoffs in concurrent systems. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing* (1990), pp. 288–292.

[77] GOUDA, M. G. The triumph and tribulation of system stabilization. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972* (1995), pp. 1–18.

[78] GOUDA, M. G., HOWELL, R. R., AND ROSIER, L. E. The instability of self-stabilization. *Acta Informatica 27* (1990), 697–724.

[79] GOUDA, M. G., AND MULTARI, N. J. Stabilizing communication protocols. *IEEE Transactions on Computers 40*, 4 (April 1991), 448–458.

[80] HADZILACOS, V., AND TOUEG, S. Fault-tolerant broadcasts and related problems. In Mullender [118], ch. 5, pp. 97–145.

[81] HALPERN, J. Y., AND FAGIN, R. Modellung knowledge and action in distributed systems. *Distributed Computing 3* (1989), 159–177.

[82] HAWKING, S. W. *Eine kurze Geschichte der Zeit.* Rowohlt Verlag, 1988.

[83] HEIDTMANN, K. *Zuverlässigkeitsbewertung technischer Systeme: Modelle für Zuverlässigkeitsstrukturen und ihre analytische Auswertung*, vol. 21 of *Teubner Texte zur Informatik.* Teubner, 1997.

[84] HEMPEL, R. The MPI standard for message passing. In *High-performance computing and networking*, W. Gentzsch and U. Harms, Eds., no. 797 in Lecture Notes in Computer Science. Springer Verlag, 1994, pp. 247–252.

[85] HERMAN, T. *Adaptivity through distributed convergence.* PhD thesis, Department of Computer Science, University of Texas at Austin, 1991.

[86] HERMAN, T. A comprehensive bibliography on self-stabilization. Internet: `http://www.cs.uiowa.edu/ftp/selfstab/-bibliography/`, July 1997.

[87] HOARE, C. Communicating sequential processes. *Communications of the Association of the Computing Machinery 21*, 8 (August 1978).

[88] HORTON, M. R. Standard for interchange of usenet messages. Internet Request for Comments RFC 850, June 1983.

[89] HORTON, M. R., AND ADAMS, R. Standard for interchange of usenet messages. Internet Request for Comments RFC 1036, December 1987.

[90] Huang, S.-T., Wuu, L.-C., and Tsai, M.-S. Distributed execution model for self-stabilizing systems. In *ICDCS94 Proceedings of the 14th International Conference on Distributed Computing Systems* (1994), pp. 432–439.

[91] Hurfin, M., Mizumo, M., Raynal, M., and Singhal, M. Efficient distributed detection of conjunctions of local predicates. Tech. Rep. Internal Publication 967, Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France, November 1995.

[92] Johnen, C. Memory-efficient self-stabilizing algorithm to construct BFS spanning trees. In *Proceedings of the Third Workshop on Self-Stabilizing Systems* (1997), Carleton University Press, pp. 125–140.

[93] Kantor, B., and Lapsley, P. Network news transfer protocol. Internet Request for Comments RFC 977, February 1986.

[94] Katz, S., and Perry, K. J. Self-stabilizing extensions for message-passing systems. *Distributed Computing 7* (1993), 17–26.

[95] Knuth, D. E. *The Art of Computer Programming*, vol. I–III. Addison-Wesley, 1969.

[96] Knuth, D. E. Literate programming. *The Computer Journal 27*, 2 (May 1984), 97–111.

[97] Kreitz, C. Automatisierte Logik und Programmierung. Skriptum zur Vorlesung im Wintersemester 1994/95 und Sommersemester 1995 an der TH Darmstadt, 1994.

[98] Kruijer, H. Self-stabilization (in spite of distributed control) in tree-structured systems. *Information Processing Letters 8* (1979), 91–95.

[99] Kumar, A., and Adly, N. HPP: a reliable causal broadcast protocol for large scale replication in wide area networks. Tech. rep., Cambridge University Computer Laboratory, 1996.

[100] Lamport, L. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering SE-3*, 2 (March 1977), 125–143.

[101] Lamport, L. Time, clocks and the ordering of events in a distributed system. *Communications of the Association of the Computing Machinery 21*, 7 (July 1978), 558–565.

[102] Lamport, L. Solved problems, unsolved problems and non-problems in concurrency, invited address. In *PODC84 Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (1983), pp. 63–67.

[103] LAMPORT, L. The mutual exclusion problem: part ii-statement and solutions. *Journal of the Association of the Computing Machinery 33* (1986), 327–348.

[104] LAMPORT, L., AND LYNCH, N. Distributed computing: models and methods. In *Handbook of Theoretical Computer Science (Volume B: Formal Models and Semantics).* Elsevier, 1990, ch. 18, pp. 1157–1199. J van Leeuwen, Editor.

[105] LEVINE, B. N., AND GARCIA-LUNA-ACEVES, J. J. A comparison of reliable multicast protocols. *Multimedia Systems* (1998). To appear.

[106] LIN, C., AND SIMON, J. Observing self-stabilization. In *PODC92 Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing* (1992), pp. 113–123.

[107] LONG, D. D. E., CARROLL, J. L., AND PARK, C. J. A study of the reliability of Internet sites. In *SRDS91 Proceedings of the 10th Symposium on Reliable Distributed Systems* (September 1991), pp. 177–186.

[108] LYNCH, N. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

[109] LYNCH, N., MERITT, M., WEIHL, W., AND FEKETE, A. *Atomic Transactions.* Morgan Kaufmann, 1994.

[110] LYNCH, N. A., AND TUTTLE, M. R. An introduction to input/output automata. *CWI Quarterly 2*, 3 (1989), 219–246.

[111] MATTERN, F. *Verteilte Basisalgorithmen.* Springer Verlag, 1989.

[112] MATTERN, F. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms* (1989), M. C. et al., Ed., Elsevier Science Publishers, pp. 215–226.

[113] MATTERN, F. Verteilte systeme. Folienkopien zur Vorlesung im Sommersemester 1996 an der Technischen Hochschule Darmstadt, 1996.

[114] MAY, D. Occam. *ACM SIGPLAN Notices 18* (1983), 69–79.

[115] MCNAB, R., AND HOWELL, F. W. Using java for discrete event simulation. The University of Edinburgh, September 1996.

[116] MENEZES, A., VAN OORSCHOT, P., AND VANSTONE, S. *Handbook of applied cryptography.* CRC Press, 1996.

[117] MOCKAPETRIS, P. Domain names – concepts and facilities. Internet Request for Comments RFC 1034, November 1987.

[118] MULLENDER, S., Ed. *Distributed Systems*, second ed. Addison-Wesley, 1993.

[119] OBRACZKA, K. *Massively replicating services in wide-area internetworks*. PhD thesis, University of Southern California, December 1994. Published as Technical Report 94-595.

[120] PORTREE, D. S. F. *Mir Hardware Heritage*. Johnson Space Center Reference Series. NASA Reference Publication 1357, March 1995.

[121] RAMSEY, N. Literate programming simplified. *IEEE Software 11*, 5 (Sept. 1994), 97–105.

[122] REITER, M. Für die Marsmission Pathfinder setzt die Nasa auf bodenständige Technologie. *Computer Zeitung 28*, 29 (July 1997), 6.

[123] ROSENDORFER, H. *Briefe in die chinesische Vergangenheit*. Deutscher Taschenbuch Verlag, 1986.

[124] SANDERS, B. A. The information structure of distributed mutual exclusion algorithms. *ACM Transactions on Computer Systems 5*, 3 (August 1987), 284–299.

[125] SCHNEIDER, F. B. What good are models and what models are good? In *Distributed Systems*, S. Mullender, Ed., 2nd ed. Addison-Wesley, 1993, ch. 2, pp. 17–26.

[126] SCHNEIDER, M. Self-stabilization. *ACM Computing Surveys 25*, 1 (1993), 45–67.

[127] SCHWARZ, R., AND MATTERN, F. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing 7* (1994), 149–174.

[128] SHIVARATRI, N. G., KRUEGER, P., AND SINGHAL, M. Load distributing for locally distributed systems. *IEEE Computer 25*, 12 (1992), 33–45.

[129] SINGHAL, M., AND KSHEMKALYANI, A. An efficient implementation of vector clocks. *Information Processing Letters 43* (August 1992), 47–52.

[130] SPECTOR, A., AND GIFFORD, D. The space shuttle primary computer system. *Communications of the Association of the Computing Machinery 27*, 9 (1984), 874–900.

[131] SPINELLI, J. M. Self-stabilizing sliding window arq protocols. In *Proceedings of IEEE Infocomm 93* (1993).

[132] STOLLER, S. D. Detecting global predicates in distributed systems with clocks. In *WDAG97 Distributed Algorithms 11th International Workshop Proceedings* (September 1997), M. Mavronicolas and P. Tsigas, Eds., no. 1320 in Lecture Notes in Computer Science, Springer Verlag, pp. 185–199.

[133] TAN, R., TEL, G., AND VAN LEEUWEN, J. Comments on “Distributed termination detection algorithm for distributed computations” (Letter to the Editor). *Information Processing Letters 23* (October 1986), 163.

[134] TANENBAUM, A. S. *Computer Networks.* Prentice Hall, 1981.

[135] TEL, G. *Topics in Distributed Algorithms.* Cambridge University Press, 1991.

[136] TEL, G. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[137] TENNANT, R. Fehlertolerante Rikscha (Karikatur). Computer Zeitung, May 1997.

[138] TSAI, M., AND HUANG, S. A self-stabilizing algorithm for the shortest paths problem with a fully distributed demon. *Parallel Processing Letters 4* (1994), 65–72.

[139] TUREK, J., AND SHASHA, D. The many faces of consensus in distributed systems. *IEEE Computer 25*, 6 (June 1992), 8–17.

[140] VARGHESE, G. *Self-stabilization by local checking and correction.* PhD thesis, MIT, 1993. Published as Technical Report MIT/LCS/TR-583.

[141] WIRTH, N. Program development by stepwise refinement. *Communications of the Association of the Computing Machinery 26*, 1 (January 1983), 70–74.

[142] WOLFSON, O., AND JAJODIA, S. An algorithm for dynamic data allocation in distributed systems. *Information Processing Letters 53* (1995), 113–119.

Entstehungsgeschichte

Diese Arbeit wurde zwischen Juli und Dezember 1997 auf einem selbstgebastelten PC mit einem Intel 486SX-Prozessor (20 MHz) und 8 MB RAM unter Linux 2.0.0 aus der Slackware-Distribution 3.1 geschrieben. Zur weiteren Ausstattung gehörte eine mindestens zehn Jahre alte Herkules-Grafikkarte und ein ebensoalter ADI DM-14 Bernstein-Monitor, der jeden Versuch, das X-Fenstersystem zum Laufen zu bringen, hartnäckig ignorierte. Durch eine fehlende *preview*-Möglichkeit konnte ich mich ganz auf den Inhalt konzentrieren.

Der Satz erfolgte mit LaTeX2ϵ aus der hervorragenden teTeX-Distribution 0.4 von Thomas Esser. Das Vorbild für das Seitenlayout war das wunderschöne "Handbook of applied cryptography" von Menezes, van Oorschot und Vanstone [116], dessen Aussehen in ein selbstgeschriebenes LaTeX-Paket übersetzt wurde. Der Zeichensatz für die Unterüberschriften ist die Helvetica aus der Reihe der Postscript-Fonts von Adobe. Die anderen Zeichensätze sind die Standard-Computer-Modern-Fonts von Donald Knuth. Die Inhaltsübersichten zu Beginn jedes Kapitels wurden mit dem `minitoc`-Paket von Jean-Pierre Drucbert erzeugt. Für das *prettyprinting* der Programmbeispiele kamen mir meine Erfahrung und die Ergebnisse meiner Studienarbeit über das *Pretzel*-System [66, 67] sehr zugute.

Alle Programmierarbeiten wurden an Rechnern des Fachbereichs Informatik der TU Darmstadt unter Verwendung des `noweb`-Systems von Norman Ramsey durchgeführt [121]. Für die Simulationen fand das `simjava`-Paket von Ross McNab und Fred Howell [115] Verwendung. Mehrere Workstations des Instituts für Systemarchitektur und der Rechnerbetriebsgruppe benötigten über eine Woche für die Berechnung der Ergebnisse. Bei Verwendung einer anderen Programmiersprache als Java wären die Berechnungen für Netzwerke aus 80 oder 160 Knoten sicherlich auch zum Abgabetermin der Arbeit fertiggeworden. Als Ausgleich für dieses Manko gibt es im Internet eine Reihe von animierten Simulationen, an denen man die Funktionsweise der hier präsentierten Algorithmen interaktiv nachvollziehen kann.

… und als Belohnung für die Leser, die bis zum Schluß lesen …

Logik
von Joachim Ringelnatz

Die Nacht war kalt und sternenklar,
Da trieb im Meer bei Norderney
Ein Suahelischnurrbarthaar. –
Die nächste Schiffsuhr wies auf drei.
Mir scheint da mancherlei nicht klar,
Man fragt doch, wenn man Logik hat,
Was sucht ein Suahelihaar
Denn nachts um drei am Kattegatt?