Institut für Informatik
der Technischen Universität München

Kausale Ordnungen in dynamischen nebenläufigen Systemen

Tobias Landes

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Uwe Baumgarten

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Peter Paul Spies
2. Univ.-Prof. Dr. Arndt Bode

Die Dissertation wurde am 18.9.2007 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 17.1.2008 angenommen.

Danksagung

Es ist mir ein Bedürfnis, einer Reihe von Personen an dieser Stelle meinen Dank auszudrücken, ohne deren direkten oder indirekten Einfluss die Entstehung der vorliegenden Dissertation nicht möglich gewesen wäre.

Zunächst gilt mein Dank Herrn Prof. Dr. Peter Paul Spies, an dessen Lehrstuhl ich in den vergangenen Jahren beschäftigt war, für die Betreuung meiner Arbeit und seine zahlreichen Denkanstöße. Außerdem danke ich Herrn Prof. Dr. Arndt Bode für die Begutachtung der Arbeit und Herrn Prof. Dr. Uwe Baumgarten für die Bereitschaft, den Vorsitz der Prüfungskommission zu übernehmen.

Des Weiteren danke ich meinen Kollegen, insbesondere Jörg Preißinger, für die angenehme langjährige Zusammenarbeit.

Ferner möchte ich meinen Eltern Helge und Rainer Landes danken, deren Opfer mir das Studium, und damit auch diese Arbeit, ermöglicht haben.

Dank gebührt außerdem meinem Lehrer Andreas „Chief“ Reincke, ohne den mein Werdegang zweifellos eine gänzlich andere Richtung genommen hätte.

Ein besonderer Dank gilt auch meinem Freund Heiko Drewes für moralische Unterstützung, zahlreiche nützliche Tipps, anregende Diskussionen und das Korrekturlesen dieser Dissertation.

Tobias Landes München, September 2007

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Nutzung, Auswertung und Speicherung der ständig wachsenden Menge anfallender Informationen erfolgt zunehmend mittels elektronischer Rechensysteme und wäre anders auch gar nicht mehr durchführbar. Aufgrund des steigenden Datenaufkommens steigt gleichermaßen auch die Komplexität der Verarbeitungsaufgaben, so dass immer mehr Rechenleistung, Speicherkapazität und sonstige Ressourcen benötigt werden. Auf der anderen Seite werden diese Ressourcen auch in immer größerem Umfang und, zumindest die Standardkomponenten, zu stetig sinkenden Preisen verfügbar. Es besteht daher zunehmend die Möglichkeit und das Interesse, möglichst viele solcher Standardkomponenten zusammenzuschließen und die immer komplexeren und aufwändigeren Datenverarbeitungsaufgaben durch kooperative parallele Rechenverfahren zu bewältigen.

Ein derartiger räumlich verteilter Zusammenschluss von zunächst unabhängigen rechenfähigen Komponenten wirft jedoch spätestens dann neue konzeptuelle Problemstellungen auf, wenn die einzelnen Komponenten, bzw. die darauf ausgeführten Berechnungen, miteinander kooperieren sollen und entsprechende Abhängigkeiten entstehen. Viele dieser Problemstellungen lassen sich letztendlich darauf zurückführen, dass es in nebenläufigen verteilten Systemen extrem schwierig ist, den aktuellen Zustand des Gesamtsystems oder auch nur die Auswirkungen einzelner Operationen zu bestimmen. Der Mangel einer global einheitlichen Zeit aufgrund nicht perfektionierbarer Synchronisation und die parallele Ausführung nebenläufiger Programminstruktionen in verschiedenen Prozessen verhindern die Charakterisierung des aktuellen Systemzustands durch einen einfachen „globalen“ Blick auf die Ausführung der Berechnung, wie sie in pseudo-parallelen Einprozessorsystemen durchaus noch möglich ist.

Die Lösung dieser Problematik ist jedoch der Schlüssel zu vielerlei Anwendungen; beispielhaft seien hier nur die folgenden genannt: Systembeobachtung (Monitoring), Breakpointing und Rollback, Erkennung von Deadlocks und anderen globalen Prädikaten, Debugging und

automatisiertes Management. Wie diese Beispiele nahe legen, ist die Lösung der genannten Problematik essentiell für die notwendige Bestrebung, nebenläufige verteilte Systeme besser verstehen, entwerfen, steuern und kontrollieren zu können.

Die allgemein gebräuchliche Lösung geht zurück auf Lamport [1, 20, 28] und basiert auf der Grundidee, anstatt einer Zeit im herkömmlichen Sinn die Kausalität zwischen den Ereignissen zu verfolgen, aus denen die Berechnung aufgebaut ist. Die kausalen Abhängigkeiten entstehen direkt aus der Programmlogik. Jeder Prozess kann als eine durch die Programmspezifikation total geordnete Sequenz von Ereignissen betrachtet werden. Darüber hinaus können die Prozesse auf die eine oder andere Weise miteinander kooperieren, so dass weitere Abhängigkeiten zwischen den Ereignissen verschiedener Prozesse entstehen.

Um diese kausalen Abhängigkeiten verfolgen zu können, werden sogenannte logische Uhren verwendet. Die ursprünglich von Lamport vorgeschlagenen skalaren logischen Uhren wurden inzwischen zu den mächtigeren Vektor-Uhren weiterentwickelt. Aber auch diese haben immer noch einen gravierenden Nachteil: Die Anzahl der an der Berechnung beteiligten Prozesse muss konstant und im Vorfeld bereits bekannt sein. Diese Voraussetzung ist für die vielen modernen Anwendungen, die auf der dynamischen Erzeugung und Beendigung von Prozessen basieren, ganz offensichtlich nicht tragbar. Neue Lösungen sind erforderlich.

In der vorliegenden Arbeit wird die eingangs genannte Problematik, eine sinnvolle Sicht auf eine nebenläufige verteilte Berechnung zu konstruieren, unter verschiedenen Gegebenheiten und unter verschiedenen Gesichtspunkten beleuchtet. Der Schwerpunkt liegt dabei auf der Suche nach einem sinnvollen und effizienten Mechanismus zur Erfassung der logischen Zeit in Systemen mit dynamischer Prozesserzeugung. Aber auch Systeme mit einem gemeinsam nutzbaren verteilten Speicher werden betrachtet.

1.1 Ziele und Aufbau dieser Arbeit

Ziel der vorliegenden Arbeit ist eine umfassende Betrachtung von Voraussetzungen, Werkzeugen und Methoden zur Konstruktion aussagekräftiger Sichten auf nebenläufige und kooperative verteilte Berechnungen. Ausgehend von den Defiziten der herkömmlichen Werkzeuge und Methoden sollen auf der Basis von Kausalität bzw. logischer Zeit neue Ansätze und Lösungen entwickelt werden, die durch höhere Leistungsfähigkeit und/oder größere Flexibilität deren Unzulänglichkeiten überwinden. Dazu sollen zunächst die für die Leistungsfähigkeit (und damit auch für die Defizite) relevanten Zusammenhänge erfasst und durchdrungen werden, um danach auf dieser Basis die neuen Ansätze und Lösungen erarbeiten zu können. Je nach Anwendbarkeit finden dabei auch verteilte Systeme unterschiedlicher Ausprägung Berücksichtigung, etwa Systeme mit und ohne Nachrichtenaustausch oder mit und ohne DSM (distributed shared memory). Dadurch sollen die Einschränkun-

gen und Voraussetzungen für den Einsatz der erarbeiteten Konzepte minimiert und eine möglichst allgemeine Anwendbarkeit sichergestellt werden.

Der Aufbau der Arbeit gestaltet sich wie folgt:

Kapitel 2 gibt einen kurzen Überblick über das Themengebiet, in dem sich die weitere Arbeit bewegt. Dabei werden insbesondere die relevanten Eigenschaften verteilter Systeme vorgestellt.

Kapitel 3 stellt die beiden wichtigsten herkömmlichen Werkzeuge zur Erfassung logischer Zeit vor: skalare Uhren und Vektor-Uhren. Alle ihre wichtigen Eigenschaften werden werden erörtert. Auf eine Art und Weise, die eine Analogie zu den beiden nächsten Kapiteln herstellt, wird beurteilt, was diese Uhren zu leisten imstande sind und was nicht. Dies bildet die Motivation und die Basis für die im Rahmen der vorliegenden Arbeit entwickelten Ansätze.

Kapitel 4 führt dynamische Vektor-Uhren ein, eine Erweiterung von Vektor-Uhren für Systeme mit dynamischer Prozesserzeugung. Die Funktionsweise ist weitgehend analog zu Vektor-Uhren, was gewisse Effizienzprobleme aufwirft, für die hier ebenfalls eine Lösung vorgeschlagen wird. Um eine optimale Vergleichbarkeit zu gewährleisten, erfolgt die Erläuterung und Beurteilung von Algorithmen, Leistungs- und Effizienzmerkmalen streng analog zu Kapitel 3.

Kapitel 5 präsentiert und diskutiert Baum-Uhren als Verwirklichung des ursprünglich angepeilten Ziels einer effizienten logischen Uhr, die sich implizit und automatisch dem dynamischen Prozessaufkommen anzupassen imstande ist. Um eine optimale Vergleichbarkeit zu gewährleisten, erfolgt die Erläuterung und Beurteilung von Algorithmen, Leistungs- und Effizienzmerkmalen wieder streng analog zu den Kapiteln 3 und 4.

Kapitel 6 beschreibt die im Rahmen der vorliegenden Arbeit realisierte Referenzimplementierung von Baum-Uhren. Diese entstand mit dem Ziel, die Funktionalität und die praktische Realisierbarkeit der in Kapitel 5 theoretisch entwickelten Konzepte und Algorithmen unter Beweis zu stellen. Der Quellcode der wichtigen Funktionen ist in Anhang A abgedruckt.

Kapitel 7 untersucht, ob und wie man die logische Zeit mit den Zeitstempeln aller Ereignisse einer verteilten Berechnung unter Abstraktion konsistent halten kann, d. h. wenn man Teile der Berechnung aus der Sicht ausblendet. Dabei wird sowohl die Abstraktion von aktiven als auch von passiven Komponenten behandelt.

Kapitel 8 bezieht in die Überlegungen zur Konstruktion einer konsistenten Sicht auch einen gemeinsam nutzbaren verteilten Speicher mit ein. Dies war bis hierhin lediglich in Kapitel 7 im Zusammenhang mit der Abstraktion passiver Komponenten der Fall. Hier

werden nun theoretische Grundlagen erarbeitet und eine exemplarische Implementierung besprochen.

Kapitel 9 fasst die wesentlichen Inhalte der Arbeit kurz zusammen und gibt Aufschluss über neu aufgeworfene und immer noch offene Fragestellungen.

Kapitel 2

Überblick über das Themengebiet

Dieses Kapitel gibt einen kurzen Überblick über das Themengebiet. Es liefert eine knappe Beschreibung verteilter Systeme und der für den Rest der Arbeit relevanten Problemstellungen, die sie aufwerfen.

Im Anschluss daran werden noch einige weitere Forschungsarbeiten aus dem Themengebiet aufgelistet.

2.1 Charakteristika verteilter Systeme

In diesem Teil des Kapitels sollen einige Charakteristika verteilter Systeme skizziert werden, die für den Rest der Arbeit auf die eine oder andere Weise von Bedeutung sind. In diesem Zusammenhang stellt sich natürlich zunächst die Frage, was denn ein verteiltes System überhaupt ist. In der Literatur existieren dazu verschiedenste Definitionen, alle mehr oder weniger unterschiedlich und keine in letzter Konsequenz befriedigend.

Es liegt daher nahe, die Definition dem jeweiligen Kontext anzupassen und entsprechende Schwerpunkte zu setzen. Im Folgenden soll deshalb keine Definition angegeben werden, die den Begriff in einem kurzen Satz auf den Punkt bringt, sondern es sollen, wie gesagt, die relevanten Merkmale zusammengestellt werden.

Als Einstieg dazu seien exemplarisch zwei Definitionen verteilter Systeme zitiert. Die erste davon stammt von Tanenbaum und van Steen [51]:*

* Die zitierte Aussage wird in der angegebenen Quelle nicht als Definition, sondern als „loose characterization“ bezeichnet.

> A distributed system is a collection of independent computers that appears to its users as a single coherent system.

Die zweite Definition stammt von Coulouris, Dollimore und Kindberg [52]:

> Bei einem verteilten System arbeiten Komponenten zusammen, die sich auf vernetzten Computern befinden und ihre Aktionen durch den Austausch von Nachrichten koordinieren.

Die erste wichtige Eigenschaft verteilter Systeme geht aus beiden zitierten Definitionen hervor: Eine gemeinsame Aufgabenstellung (im weitesten Sinne) wird durch mehrere zunächst unabhängige rechenfähige Komponenten bearbeitet.

Dies führt direkt zu einer weiteren Feststellung, die im zweiten Zitat auch explizit formuliert wird: Die zunächst unabhängigen Komponenten bleiben nicht unabhängig, sondern kooperieren miteinander. Dies setzt natürlich eine zur Kommunikation geeignete Infrastruktur voraus, im zweiten Zitat durch die Vernetzung wiederum explizit erwähnt.

Die jeweils letzten Teile der beiden gegebenen Definitionen drücken unterschiedliche Aspekte bzw. Forderungen aus, die nichtsdestoweniger beide recht instruktiv sind.

Tanenbaum und van Steen stellen die Forderung nach Transparenz, d. h. sie verlangen, dass der Benutzer eines verteilten Systems nichts darüber wissen muss, dass oder wie das System, an dem er arbeitet, verteilt ist. Dies erfordert zwangsläufig irgendeine Art von Kooperation der verteilten Komponenten, so dass auch in dieser Definition implizit die Eigenschaft der Kooperation sichtbar ist. Im zweiten Zitat wird keine Transparenz gefordert. Transparenz gibt es jedoch in sehr unterschiedlichen Ausprägungen und Graden, von denen zumindest einige für fast alle verteilten Systeme gefordert werden, während andere zumindest wünschenswert sind. Einige der möglichen Ausprägungen von Transparenz werden in Abschnitt 2.1.2 kurz erklärt.

Anstatt der Forderung nach Transparenz stellen Coulouris, Dollimore und Kindberg in ihrer Definition die Art der Kommunikation zwischen den verteilten Komponenten in den Vordergrund. Die Festlegung auf Nachrichtenkommunikation ist an dieser Stelle nicht ganz unproblematisch, aber dennoch instruktiv, weil sie unser Augenmerk auf die verschiedenen Ebenen lenkt, auf denen wir verteilte Systeme betrachten können. Auf einer niedrigen Abstraktionsebene, etwa der Ebene der verteilten Hardwarekomponenten, wird die Definition auf fast alle verteilten Systeme zutreffen. Lediglich Systeme mit echten physikalisch geteilten Speicherkomponenten könnten anstatt durch Nachrichtenaustausch über die Daten in diesem Speicher kommunizieren. Ob solche Systeme, im Allgemeinen sogenannte Multiprozessorsysteme, als verteilte Systeme betrachtet werden, ist jedoch wiederum strittig – Coulouris, Dollimore und Kindberg tun dies offensichtlich nicht. Hingegen muss zumindest

auf höheren Abstraktionsebenen die Kommunikation über andere Konzepte unbedingt zugelassen werden. Insbesondere kann durch eine sogenannte Middleware, oder auch durch das verteilte Betriebssystem selbst, ein von den verteilten Prozessen gemeinsam nutzbarer Speicher auf der Basis von physikalisch verteilten Komponenten realisiert werden. Dass dazu auf der Realisierungsebene womöglich durchaus Nachrichtenkommunikation zum Einsatz kommt, ist dann beispielsweise auf der Ebene der Anwendungsentwicklung nicht mehr erkennbar.*

Um eine allgemeinstmögliche Anwendbarkeit der in der vorliegenden Arbeit behandelten Konzepte sicherzustellen, seien zusammenfassend für den Rest der Arbeit lediglich folgende Eigenschaften für verteilte Systeme angenommen:

- Das verteilte System besteht aus einer nicht näher spezifizierten Anzahl von Computern, d. h. die einzelnen Stationen verfügen sowohl über Rechen- als auch über Speicherfähigkeiten.
- Diese Stationen, und die auf ihnen ausgeführten Prozesse, seien in der Lage, miteinander zu kommunizieren, und zwar entweder explizit mittels Nachrichtenaustausch oder† indirekt über die gemeinsame Nutzung eines allen Prozessen gleichermaßen zur Verfügung stehenden Speichers.‡

Transparenz spielt auf der hier betrachteten Abstraktionsebene keine Rolle. Sie wird lediglich in einem anderen Zusammenhang, nämlich als Transparenz einer Abstraktion als solcher und als Abstraktion von Speicherobjekten, in Kapitel 7 behandelt.

2.1.1 Sinn und Zweck verteilter Systeme

Mit der ständig steigenden Menge von Daten aller Art steigt auch die Komplexität ihrer Verarbeitung, so dass immer mehr Ressourcen in Form von Rechenleistung und Speicherkapazität benötigt werden. Auf der anderen Seite werden diese Ressourcen auch in immer größerem Umfang und, zumindest die Standardkomponenten, zu stetig sinkenden Preisen verfügbar. Es liegt daher nahe, viele solcher Standardkomponenten miteinander zu koppeln und die immer komplexeren und aufwändigeren Datenverarbeitungsaufgaben durch kooperative parallele Problemlösungsverfahren zu bewältigen.

---

* Dies ist z. B. bei dem Experimentalsystem MoDiS der Fall, das zur exemplarischen Implementierung verschiedener in dieser Arbeit entwickelter Konzepte verwendet wurde und in diesem Zusammenhang in Kapitel 6 sowie Abschnitt 8.3 erwähnt wird. Beschrieben wird MoDiS in Abschnitt 6.3.

† Dieses ‚oder' sei nicht exklusiv!

‡ Welche Konfiguration gerade angenommen wird, wird jeweils an geeigneter Stelle klargestellt, üblicherweise bei der Besprechung des Systemmodells.

Die Vorteile dieses Ansatzes liegen klar auf der Hand: mehr Rechenleistung, mehr Flexibilität, mehr Parallelität, bessere Skalierbarkeit und bessere Nutzung von Ressourcen zu günstigeren Preisen.

Flexibilität umfasst dabei verschiedene Aspekte. Zum einen ist damit die Flexibilität bei der Auswahl der Komponenten gemeint; viele verteilte Systeme sind in dieser Hinsicht sehr heterogen. Zum anderen betrifft die Flexibilität aber auch beispielsweise die physikalische Platzierung der Komponenten oder die Austauschbarkeit von Komponenten.

Parallelität spricht weitgehend für sich selbst. Es ist mit verteilten Systemen möglich, komplexe Datenverarbeitungsaufgaben in Teilaufgaben zu zerlegen und diese gleichzeitig abzuarbeiten.* Dies ist zwar durch die höhere Rechenleistung eines nicht-verteilten Spezialrechners bis zu einem gewissen Grad ausgleichbar, aber ein solcher ist im Allgemeinen deutlich teurer als viele weniger leistungsfähige Standardkomponenten. Außerdem ist seine Rechenleistung bei Bedarf schwieriger steigerbar, während ein verteiltes System normalerweise hochgradig skalierbar ist, also relativ problemlos um weitere Komponenten erweitert werden kann.

Der parallele Einsatz vieler gleicher oder ähnlicher Komponenten hat darüber hinaus den Vorteil, dass eine gesteigerte Redundanz vorliegt, und daher beim Ausfall einzelner Komponenten deren Aufgaben auf andere Komponenten verteilt werden können, ohne das ganze System oder die verteilte Berechnung anhalten zu müssen. Dadurch kann in verteilten Systemen eine hohe Fehlertoleranz und Robustheit erzielt werden.

Wie wichtig verteilte Systeme heute sind und in Zukunft noch werden, wird auch an der wachsenden Bedeutung des Internet deutlich. Dieses stellt vermutlich das größte und bekannteste verteilte System dar, ständig im Wandel, mit unzähligen Substrukturen und ebenso vielen Anwendungsbereichen. Letztere reichen von einfacher Client-/Server-Kommunikation beim Abruf von Webseiten bis hin zu komplexen verteilten Berechnungen, etwa im Rahmen des SETI@home-Projekts [58].

2.1.2 Problemstellungen in verteilten Systemen

Im letzten Unterabschnitt wurden die Vorteile verteilter Systeme aufgelistet. Die Vorteile sind allerdings auch mit gewissen Nachteilen verbunden. Diese bestehen insbesondere in entsprechenden Schwierigkeiten für die Entwickler eines verteilten Systems und in Zugeständnissen, die der Benutzer eines derartigen Systems machen muss.

* Vergleiche hierzu das Problem der Parallelisierbarkeit, kurz umrissen in Abschnitt 2.1.2.

In diesem Unterabschnitt werden nun einige Problemkreise angesprochen, die speziell in verteilten Systemen auftreten, weil sie sich direkt oder indirekt aus den in Abschnitt 2.1 geforderten Eigenschaften verteilter Systeme ergeben.

Parallelisierbarkeit

In der Einleitung wurde die parallele Ausführbarkeit von Teilen einer Gesamtberechnung als Vorteil verteilter Systeme hervorgehoben. Auf der anderen Seite sollte man jedoch auch nicht übersehen, dass eine sinnvolle Zerlegung einer komplexen Berechnung in mehrere nebenläufig berechenbare Teilberechnungen eine nicht-triviale (um nicht zu sagen schwierige) Angelegenheit ist. In diesem Zusammenhang ist das Theorem von Bernstein [32] zu erwähnen, welches besagt, dass im allgemeinen Fall nicht entscheidbar ist, ob zwei Codesegmente eines gegebenen sequentiellen Programms nebenläufig ausführbar sind.

Einige Aspekte dieses Problems wurden bereits (unter anderem) in [15], der Diplomarbeit des Autors, sowie in [16] behandelt.

Konsistente Betrachtung

In einem sequentiellen System ist es relativ einfach, sich ein Bild von einem momentanen Zustand des Systems zu machen, d. h. einen Schnappschuss zu irgendeinem Zeitpunkt während einer laufenden Berechnung. Es existiert nur ein einziger Ausführungsfaden, der zur Not angehalten werden kann. Die Inhalte des Speichers und der Prozessor-Register einschließlich des Programmzählers können ohne weiteres zum selben logischen Zeitpunkt ausgelesen und zur aktuellen Position im Ausführungsfaden in Bezug gesetzt werden. Die Folgen einer Programminstruktion hinsichtlich der hervorgerufenen Zustandsänderung des Systems sind problemlos nachvollziehbar.

Selbst in pseudo-parallelen Monoprozessor-Systemen ist diese Situation nahezu unverändert. Es liegen hier zwar mehrere prinzipiell nebenläufige Ausführungsfäden vor, aber zu jedem beliebigen Zeitpunkt ist nach wie vor die aktuelle Position in jedem dieser Ausführungsfäden samt der zugehörigen Speicherinhalte eindeutig bestimmbar. Dies liegt zum Einen daran, dass sich, wie in sequentiellen Systemen, tatsächlich immer nur ein Ausführungsfaden in Ausführung befindet; die anderen sind unterdessen in ihrem aktuellen Zustand eingefroren. Zum Anderen, und dies ist der entscheidende Punkt, ist in derartigen Systemen ein einmal spezifizierter Zeitpunkt problemlos auf alle beteiligten Komponenten anwendbar, weil diese alle demselben Zeitverständnis unterliegen.

Genau dies ist in verteilten Systemen nicht der Fall. Die in Abschnitt 2.1 festgelegten Eigenschaften eines verteilten Systems stellen voneinander unabhängige, wenn auch koope-

rierende, Komponenten fest, ohne irgendeine Art von zeitlicher Synchronisation vorauszusetzen. Es muss also davon ausgegangen werden, dass jede der verteilten speicher- und rechenfähigen Komponenten ihre eigene lokale Uhr besitzt. Über die Genauigkeit, mit der diese lokalen Uhren synchronisiert sind, kann keine Aussage getroffen werden. Eine perfekte Synchronisation wäre auch praktisch nicht realisierbar. Was in verteilten Systemen also fehlt, ist eine *globale Uhr* oder *globale Zeit*. Das bedeutet, dass in verteilten Systemen kein Zeitpunkt spezifiziert werden kann, unter dem alle beteiligten Komponenten tatsächlich denselben realen Zeitpunkt verstehen.

Dieser Mangel hat weitreichende Konsequenzen. Da die auf einem Rechensystem ausgeführte Berechnung mit der Zeit fortschreitet (egal ob das System sequentiell oder nebenläufig arbeitet), ist der *Zustand* des Systems abhängig von der Zeit. Die Zeit und die möglichen Zustände des Systems bilden also gemeinsam einen zweidimensionalen Raum. In diesem *Zustandsraum* orientiert man sich gewöhnlich anhand der Zeit, die naturgemäß eine totale Ordnung liefert. Anhand der Zeit können die Zustände daher geordnet und in einen Kontext gebracht werden, so dass sie sinnvoll interpretiert werden können. Dadurch, dass in verteilten Systemen eine globale Zeit fehlt, fehlt hier auch der gewohnte Ordnungsfaktor im Zustandsraum. Der Kontext eines Systemzustands, und damit letztendlich seine Bedeutung oder Aussagekraft, ist nicht mehr ohne weiteres feststellbar.

Dies führt dazu, dass beispielsweise Aussagen der folgenden Arten in verteilten Systemen nicht auf der gewohnten Zeitbasis getroffen werden können:

- Ereignis x (in einem Ausführungsfaden) trat vor Ereignis y (in einem anderen Ausführungsfaden) auf.
- Als sich der Ausführungsfaden A an der Position x befand, wurde von Ausführungsfaden B gerade die Instruktion y ausgeführt.
- Zum Zeitpunkt t befand sich Komponente k in diesem Zustand, Komponente l in jenem.

Derartige Fragen wären beispielsweise für das Debugging einer verteilten Anwendung relevant. Die erste der drei Aussagen begründet darüber hinaus das Problem der Speichersemantik, das in Abschnitt 2.1.2 erläutert wird. Ferner verdeutlicht insbesondere die dritte Aussage, dass sich schlicht und einfach keine Aussage über den globalen Zustand des Gesamtsystems zum Zeitpunkt t treffen lässt, da sich dieser aus den den lokalen Zuständen der verteilten Komponenten zusammensetzen würde. Ein solcher globaler Gesamtzustand ist jedoch seinerseits wieder für viele Anwendungen interessant, vom Debugging über Systembeobachtung (Monitoring) oder Deadlock-Erkennung bis hin zur Erstellung sogenannter Breakpoints für Rollback-Verfahren.

Um all diese Probleme lösen zu können, muss ein neuer Zeitbegriff eingeführt werden, mittels dessen, in Abwesenheit einer absoluten Zeit, eine Orientierung im Zustandsraum möglich wird. Da verteilte Systeme der allgemeinen Kausalität unterliegen, bestehen zwischen den Ereignissen kausale Beziehungen, die eine zeitliche Anordnung erlauben. Auf der Basis der inhärenten Kausalität wird daher der Begriff der *logischen Zeit* definiert. Dieses Konzept wurde von Lamport [1, 20, 28] eingeführt und ist, unter verschiedenen Aspekten, der hauptsächliche Gegenstand dieser Arbeit.

Da die logische Zeit im Gegensatz zur realen Zeit zwar keine totale, aber immerhin eine partielle Ordnung etabliert, können auf dieser Basis Aussagen der obigen Art zumindest näherungsweise getroffen werden. Die Menge der Zustände etwa, in denen sich Komponente l befunden haben kann, als sich Komponente k in einem gegebenen Zustand befand, kann auf diese Weise immerhin sinnvoll eingeschränkt werden. Und da man nicht feststellen kann, dass dieser oder jener Gesamtzustand im Laufe der Berechnung aufgetreten *ist*, entscheidet man, ob dieser Zustand während der Berechnung auftreten *konnte* (in diesem Fall gilt der besagte Zustand als *konsistent*).

Logische Uhren leisten jedoch mehr als die bloße Ersetzung realer Zeit in verteilten Systemen. Erst die Erschließung der Kausalität ermöglicht eine semantische Einordnung von Ereignissen und Zuständen. Mittels realer Zeit könnten sie lediglich in eine zeitliche Beziehung zueinander gesetzt werden. Von besonderem Interesse sind jedoch Fragen hinsichtlich gegenseitiger Abhängigkeiten und Einflussnahme, und diese lassen sich ausschließlich mittels logischer Zeit behandeln.

Für weiterführende Erklärungen zu Kausalität und logischen Uhren sei auf Kapitel 3 verwiesen.

Speichersemantik

In Systemen mit gemeinsam nutzbarem verteiltem Speicher (DSM) tritt das Problem der Semantik von Speicheroperationen auf, die ein Anwendungsprogrammierer erwarten und vorfinden kann. In sequentiellen Systemen erwarten wir, dass das Auslesen einer Speicherzelle genau den Wert zurückliefert, der mit dem letzten Schreibvorgang dort hineingeschrieben wurde. Nun ist in verteilten Systemen jedoch aufgrund der fehlenden globalen Uhr (s. o.) der „letzte“ Schreibvorgang nicht mehr eindeutig identifizierbar. Damit der Programmierer trotzdem weiß, welche Werte von welchen Speicherzugriffen zurückgeliefert werden können, d. h. mit welcher Speichersemantik er zu rechnen hat, wird diese durch das *Konsistenzmodell* des verteilten Speichers genau spezifiziert.

Mehr zu Konsistenzmodellen, einschließlich einer Auflistung der wichtigsten Konsistenzmodelle und ihrer Eigenschaften, findet sich in Abschnitt 8.2.3.

Transparenz

Transparenz bezeichnet die Eigenschaft eines Systems, bestimmte Details vor dem Benutzer zu verbergen. Diese Eigenschaft ist ganz allgemein wünschenswert, weil die Sicht des Benutzers auf das System dadurch so unkompliziert wie möglich gehalten wird, und der Benutzer mit Dingen, die ihn in letzter Konsequenz nicht interessieren würden, gar nicht erst konfrontiert wird.

Transparenz gibt es in unterschiedlichsten Ausprägungen, die sich auf die Art der jeweils verborgenen Details beziehen. In verteilten Systemen geht es ganz allgemein darum, die Verteiltheit des Systems als solche zu verbergen. Aber auch die Verteiltheit umfasst eine Reihe sehr unterschiedlicher Aspekte, die unabhängig voneinander transparent gemacht werden können. Es folgt nun ein kurzer Abriss der wichtigsten Formen von Transparenz in verteilten Systemen.

- *Zugriffstransparenz*
  Auf lokale und entfernte Ressourcen kann in derselben Weise zugegriffen werden, ungeachtet unterschiedlicher Datenrepräsentationen oder Ähnlichem.
- *Ortstransparenz*
  Für den Zugriff auf eine Ressource muss deren physische Position nicht bekannt sein. Sie kann über einen Identifikator referenziert werden, der von der tatsächlichen Position abstrahiert.
- *Migrationstransparenz*
  Die physische Position von Ressourcen kann sich ändern, ohne dass der Benutzer darüber etwas wissen muss oder sich die Zugriffsmechanismen ändern.
- *Replikationstransparenz*
  Ressourcen können im System repliziert werden, ohne dass sich der Benutzer dessen bewusst ist.
- *Nebenläufigkeitstransparenz*
  Es wird verborgen, dass Ressourcen von mehreren Benutzern oder Prozessen gleichzeitig verwendet werden können.
- *Fehlertransparenz*
  Auftretende Fehler werden vor dem Benutzer verborgen. Er kann ungehindert weiterarbeiten und seine Daten bleiben konsistent.

Die Realisierung von Transparenz stellt natürlich eine signifikante Hürde bei der Entwicklung verteilter Systeme dar. Welche Formen von Transparenz in welcher Vollständigkeit (Grad der Transparenz) umgesetzt werden, ist von System zu System unterschiedlich.

Es ist außerdem zu erwähnen, dass der Grad einer Transparenz auch von der jeweiligen Abstraktionsebene abhängt. Was auf einer hohen Ebene verborgen ist, muss auf einer niedrigeren Ebene dennoch explizit umgesetzt werden, dort also voll sichtbar sein. Dies führt direkt dazu, dass häufig feste Schichten mit spezifischen Aufgaben definiert werden, deren Realisierung für die jeweils höheren Schichten dann transparent ist. Das bekannteste Beispiel ist das ISO-OSI-Referenzmodell [50].

2.2 Verwandte Arbeiten

Grundlage für die in dieser Arbeit angestellten Betrachtungen, und damit die wichtigsten verwandten Arbeiten, sind Leslie Lamports Überlegungen zur Kausalität in verteilten Systemen, mit denen die Konzepte logischer Zeit eingeführt wurden. Als entsprechende Meilensteine sind hier die Veröffentlichungen [1], [20] und [28] hervorzuheben.

Weitere wichtige Arbeiten, auf denen die vorliegende aufbaut, stammen von Friedemann Mattern [9, 26, 2] und Colin Fidge [10, 11], die, unabhängig voneinander, die Leistungsfähigkeit von Lamports logischen Uhren durch die Einführung der Vektor-Uhren verbessert haben. Die in der vorliegenden Arbeit entwickelten Konzepte stellen wiederum eine deutliche Verbesserung gegenüber den Vektor-Uhren dar.

Die Arbeit zu konsistenten globalen Zuständen von Özalp Babaoğlu und Keith Marzullo [5] ist verwandt in ihrem Ansatz, das Grundproblem der konsistenten Betrachtung verteilter Systeme samt den Möglichkeiten zusammenfassend zu umreißen, die herkömmliche Uhrensysteme bieten oder auch nicht bieten. Unter anderem wird in der vorliegenden Arbeit ein ähnlicher Ansatz verfolgt, jedoch mit dem Ziel, durch Analogie eine optimale Vergleichbarkeit der Leistungsmerkmale und sonstigen Eigenschaften von herkömmlichen und neu entwickelten Werkzeugen zu erreichen.

Etwas aktuellere Überlegungen, ähnlich denen in dieser Arbeit und zeitlich parallel, wurden von Anurag Agarwal und Vijay K. Garg im Juli 2005 veröffentlicht [24]. Hier wird versucht, die Defizite von Vektor-Uhren durch Verallgemeinerung in den Griff zu bekommen. Vektor-Uhren werden zu sogenannten *chain clocks* verallgemeinert, bei denen die total geordneten sequentiellen Ketten von Ereignissen nicht zwangsläufig nach den Prozessen gebildet werden, sondern so, wie es im Einzelfall gerade effizient ist. Durch diese flexible Herangehensweise entsteht nicht ein Algorithmus zur Führung logischer Zeit, sondern eine Klasse davon. Die chain clocks (bzw. eine Instanz davon) werden auch im Zusammenhang mit gemeinsam nutzbaren Speicherobjekten betrachtet. Chain clocks basieren allerdings auf der Annahme, dass je nach Anwendung lediglich ein Teil der Ereignisse relevant ist, und sind daher nur für spezielle Anwendungen geeignet bzw. effizient, etwa für die Erkennung globaler Prädikate.

Im Zusammenhang kausaler Ordnungen auf einem gemeinsam nutzbaren verteilten Speicher (DSM) seien die Arbeiten von Phillip B. Gibbons und Ephraim Korach [25] sowie von Hon F. Li und Gabriel Girard [41, 42] erwähnt. Bei diesen Arbeiten liegt der Schwerpunkt auf Konsistenzmodellen und deren Verifikation bzw. Implementierung. Insbesondere die Arbeit von Gibbons und Korach wird in Kapitel 8 als Basis verwendet.

Mit der grundsätzlichen Führung logischer Zeit auf passiven Speicherobjekten beschäftigen sich L. Gunaseelan und Richard J. LeBlanc, Jr. in [17]. Diese Überlegungen werden unter anderem in Kapitel 7 im Zusammenhang mit der Abstraktionshierarchie derartiger Objekte aufgegriffen und weitergeführt.

Kapitel 3

Logische Uhren

Wann immer man nicht ein statisches, sich nicht veränderndes Gebilde betrachtet, sondern ein dynamisches Gebilde im weitesten Sinne, muss man das, was man wahrnimmt, in Bezug setzen zur Zeit. Das liegt ganz einfach daran, dass alles, was sich dynamisch, also mit der Zeit, verändert, zu jedem Betrachtungszeitpunkt unterschiedlich aussehen kann. Somit ändert sich auch die Interpretation dessen, was man wahrnimmt, je nach dem, was man über den Zeitpunkt der Wahrnehmung weiß oder vermutet.

Ein einfaches anschauliches Beispiel ist die Beobachtung der Tageslichtverhältnisse. Wenn man sieht, dass Dämmerung herrscht, kann man daraus noch nicht folgern, ob es nun hell oder dunkel zu werden im Begriff ist. Dazu muss man zusätzlich wissen, ob die Betrachtung morgens oder abends stattfindet. Das Beispiel verdeutlicht gleich noch einen weiteren wichtigen Punkt. Zur Bestimmung des Zeitpunkts (morgens oder abends) kann es hilfreich sein, etwa wenn man keine Uhr oder nur eine Zwölf-Stunden-Anzeige zur Verfügung hat, zu wissen, wie der Zustand (zeitlich gesehen) vor dem Beobachtungszeitpunkt war. Dazu wiederum muss allerdings außerdem der zu Grunde liegende Ablauf bereits im Vorfeld bekannt und keiner potentiellen Änderung unterworfen sein. Hier:

$$\ldots \longrightarrow \text{dunkel} \longrightarrow \text{Dämmerung} \longrightarrow \text{hell} \longrightarrow \text{Dämmerung} \longrightarrow \text{dunkel} \longrightarrow \ldots$$

Bei der Beobachtung von Berechnungsabläufen ist man genau in einer solchen Situation. Um einen beobachteten Zustand, etwa einen Schnappschuss, sinnvoll beurteilen zu können, muss man zunächst den Beobachtungszeitpunkt beurteilen, d. h. den *Kontext* des Zustands. In einem verteilten System ist man nun gemeinhin damit konfrontiert, dass keine objektive Realzeitmessung möglich ist, also keine Uhr im herkömmlichen Sinne zur Verfügung steht. Man muss sich also, gerade wie im obigen Beispiel, damit behelfen, den Beobachtungszeitpunkt dadurch näher zu qualifizieren, dass man ihn in den Ablauf einordnet, soweit dieser bekannt bzw. bereits beobachtet worden ist. Der wahrgenommene Zustand oder das

wahrgenommene Ereignis wird also *relativ* zu anderen Zuständen/Ereignissen interpretiert, wobei die *Logik* des Ablaufs an sich zu Grunde gelegt wird. Mit genau diesem Ziel und Hintergrund entwickelte Leslie Lamport [1, 20, 28] die sogenannten *logischen Uhren*. Logische Uhren sollen die Zustände oder Ereignisse eines Programmablaufs nicht zeitlich im Sinne einer herkömmlichen Uhr, sondern relativ zueinander ordnen. Das heißt, im Vordergrund steht nicht die reale, sondern die *logische Zeit*. Nichtsdestoweniger ermöglicht diese wieder eine Orientierung in dem von Zeit und Systemzuständen aufgespannten zweidimensionalen Raum. Einem wahrgenommenen Zustand kann dadurch wieder ein Kontext und damit eine sinnvolle Interpretation zugeordnet werden.

Abgesehen davon ist die Messung der logischen Zeit jedoch mehr als nur ein Vehikel im Angesicht der Nicht-Messbarkeit realer Zeit in verteilten Systemen. Gerade bei der Beobachtung verteilter Anwendungen sind sehr oft Fragen von hohem Interesse, die sich allein durch Realzeitmessung überhaupt nicht beantworten ließen. Erst die logische Zeit ermöglicht das Erschließen von *Kausalität*, das heißt Zusammenhängen zwischen beobachteten Ereignissen oder Zuständen im Hinblick auf ihre jeweilige Funktion innerhalb der Programmlogik. Zwei Zustände, über die nichts bekannt wäre als die Realzeitpunkte ihres Auftretens, ließen sich vom Beobachter auch in keine andere Beziehung zueinander setzen als nur die rein zeitliche. Man könnte zwar Aussagen treffen wie „A kam vor B“, nicht aber erkennen, ob A in irgendeiner Weise Einfluss auf B hatte. Und gerade derartige Zusammenhänge sind bei den meisten Anwendungen von Systembeobachtung von besonderem Interesse. Als nahe liegendes Beispiel sei an dieser Stelle nur Debugging genannt.

Die kausalen Zusammenhänge, die von der logischen Zeit erfasst werden, spiegeln auch jede nur denkbare Art von möglichem *Informationsfluss* wider.

Im Laufe dieses Kapitels werden die beiden herkömmlichen Arten logischer Uhren besprochen, also skalare und Vektor-Uhren. Im Wesentlichen werden detailliert die Konzepte, Funktionsweisen und Algorithmen dieser Uhren vorgestellt. Es werden aber auch einige eigene Analysen und Beobachtungen angeführt, um Analogie und Vergleichbarkeit für die danach folgenden Kapitel herzustellen.

Diese herkömmlichen Uhren haben in der heutigen Praxis einen großen Nachteil, der darin besteht, dass sie entweder die logische Zeit nur unvollständig wiedergeben, oder aber inhärent ungeeignet sind für Anwendungen mit dynamischer Prozesserzeugung. Die Ursachen hierfür werden im Folgenden ebenfalls erörtert. In den Kapiteln 4 und 5 werden dann verbesserte logische Uhren vorgestellt, die im Rahmen dieser Arbeit mit dem Ziel entwickelt wurden, vollständige logische Zeiterfassung auch in dynamischen Prozesssystemen zu ermöglichen.

Zuallererst wird jedoch das zu Grunde liegende Systemmodell beschrieben. Im Anschluss daran erfolgt die Erläuterung allgemeiner Eigenschaften und möglicher Leistungsmerkmale logischer Uhren.

3.1 Systemmodell

Dieser Abschnitt beschreibt das Systemmodell, anhand dessen die folgenden Abschnitte skalare Uhren und Vektor-Uhren behandeln werden. Dieses Modell wurde von Anfang an so allgemein wie möglich gehalten und verzichtet daher auf gemeinsamen verteilten Speicher (DSM) oder andere komplexe Features. Um die Uhren und ihre Funktionsweisen zu erläutern ist es jedoch vollkommen ausreichend. Für die im Rahmen dieser Arbeit entwickelten und in Kapitel 5 vorgestellten Konzepte wird dieses Systemmodell den Anforderungen entsprechend angepasst werden.

Da logische Uhren zur Erfassung dynamischer Berechnungsabläufe dienen sollen, ist die Grundlage des Systemmodells konsequenterweise der *Prozess*. Die Berechnung selbst besteht aus einer Menge $P = \{p_1, p_2, \ldots, p_n\}$ von n Prozessen. n sei größer als Null und beliebig groß, aber endlich. Die Nummern von 1 bis n seien den Prozessen systemweit eindeutig zugeordnet (globaler Namensraum).

Damit man von einer verteilten Berechnung sprechen kann, und nicht bloß von vielen unabhängigen Einzelberechnungen, sollen die Elemente von P, also die Prozesse, die Teil derselben Berechnung sind, auch miteinander kommunizieren können. Zu diesem Zweck sei es den Prozessen möglich und gestattet, untereinander *Nachrichten* auszutauschen. Diese Nachrichten seien beliebige (temporäre) Datenobjekte, die vom zu Grunde liegenden System *zuverlässig* vom Absender zum Empfänger (beides Prozesse aus P) transportiert werden. Über die zwischen dem Versand und dem Empfang einer Nachricht auftretende Verzögerung sei nichts weiter angenommen, als dass sie endlich sei.

Für viele Anwendungen ist es unabdingbar, dass die Reihenfolge der Nachrichten beim Transport erhalten bleibt, dass die Nachrichten also in derselben Reihenfolge zugestellt werden, in der sie auch abgeschickt wurden. Diese Funktionalität des zu Grunde liegenden Transportsystems wird im Allgemeinen als *FIFO-Kanäle* oder *FIFO channels* bezeichnet. Es liegt auf der Hand, dass solche FIFO-Kanäle in engem Zusammenhang mit logischen Uhren stehen. Da logische Uhren eben dazu dienen können, die (notwendige) Reihenfolge des Versands der Nachrichten zu rekonstruieren, sollen die FIFO-Kanäle hier nicht als gegeben vorausgesetzt werden. Sie stellen vielmehr ein Ziel dar, das mit Hilfe der logischen Uhren erreicht bzw. implementiert werden kann.

Jeder Prozess p_i zerfällt in eine Sequenz von *Ereignissen* $E_i = \{e_i^1, e_i^2, \ldots\}$, welche einer totalen Ordnung unterliegen, und zwar der *Prozessordnung* $\rightarrow$. Diese Ereignisse stellen die Berechnungsschritte dar, deren ordnungsgemäße Ausführung schließlich das Ergebnis liefert. Jedes Ereignis ist atomar und verändert den *Zustand* oder *Status* des Prozesses. Was genau ein solches Ereignis umfasst, wird offen gelassen, da es von der betrachteten Abstraktionsebene abhängt. Ein Ereignis kann eine einzelne Programmanweisung oder auch eine

ganze Prozedur umfassen. Wichtig ist nur, dass es auf der betrachteten Abstraktionsebene als unteilbare Einheit gesehen wird. E sei die Menge aller Ereignisse der Berechnung.

Von besonderer Bedeutung für alle Betrachtungen des globalen Verhaltens von Systemen mit interagierenden/kooperierenden Prozessen sind, nach dem beschriebenen Systemmodell, diejenigen Ereignisse, die den Versand oder den Empfang einer Nachricht repräsentieren. Diese Ereignisse werden im Folgenden als *Sendeereignisse* bzw. *Empfangsereignisse* bezeichnet. Ihre spezielle Signifikanz liegt darin begründet, dass sie Abhängigkeiten zwischen Teilen (Ereignissen) verschiedener Prozesse etablieren, die ohne die Nachrichten völlig unabhängig voneinander wären. Die Nachrichten haben den wichtigen Effekt, Prozesse oder Prozessteile miteinander zu synchronisieren. Sie erweitern die totale Ordnung über den Ereignissen eines Prozesses zu einer partiellen Ordnung über den Ereignissen aller Prozesse. Diese partielle Ordnung spiegelt jede nur denkbare Art von möglichem Informationsfluss wider.

3.2 Eigenschaften logischer Uhren

Wie am Anfang von Kapitel 3 bereits erklärt, existieren verschiedene Motivationen dafür, in verteilten Systemen anstatt der realen Zeit die logische Zeit zu erfassen. Logische Zeit basiert nicht auf dem herkömmlichen zeitlichen Kontinuum, sondern auf den kausalen Abhängigkeiten, die zwischen den (in Abschnitt 3.1 definierten) Ereignissen bestehen. Dieses kausale Beziehungsgeflecht wiederum entspringt direkt der Programmlogik, wie sie letztendlich vom Programmierer vorgegeben wird. Im Zusammenhang mit logischer Zeit muss eine Aussage wie „Ereignis e_i tritt vor Ereignis e_j auf“ verstanden werden als „Ereignis e_i muss vor Ereignis e_j auftreten“ – oder, bei rekonstruierender, rückblickender Anwendung: „Ereignis e_i muss vor Ereignis e_j aufgetreten sein“. Natürlich gibt es im Zuge der Berechnung auch Ereignisse, die in keiner kausalen Beziehung zueinander stehen und für die eine derartige Aussage nicht getroffen werden kann; die Ereignisse bleiben ungeordnet. Solche Ereignisse müssen immer zu verschiedenen Prozessen gehören, da die Ereignisse ein und desselben Prozesses durch die Prozessordnung tatsächlich total geordnet sind. Prozessübergreifend betrachtet unterliegen die Ereignisse jedoch, wie bereits bei der Vorstellung des Systemmodells erwähnt, lediglich einer partiellen Ordnung.

Wie man es auch von einer realen Uhr erwarten würde, macht eine logische Uhr nichts anderes, als einem Ereignis e_i^x einen *Zeitstempel* $C(e_i^x)$ zuzuordnen. Dieser Zeitstempel repräsentiert den logischen Zeitpunkt, an dem das Ereignis stattgefunden hat. Logische Zeitstempel lassen sich vergleichen und interpretieren, gerade wie reale Zeitstempel auch.

Um alle kausalen Abhängigkeiten formal zu erfassen, definierte Lamport [1] die *happened-before-Relation*, und zwar als eine partielle Ordnungsrelation, welche die transitive Hülle der

Prozessordnung auf der einen Seite und der natürlichen Sende-/Empfangs-Abhängigkeiten auf der anderen Seite darstellt. Die formale Definition lautet wie folgt:

Definition 3.1. Die *happened-before-Relation* $\Rightarrow$ ist die kleinste Relation (im Sinne der Mächtigkeit), welche die folgenden Bedingungen erfüllt:

- Wenn $e_i^x \rightarrow e_i^y$, dann $e_i^x \Rightarrow e_i^y$.
- Wenn e_i^x ein Sendeereignis und e_j^y das Empfangsereignis der dazugehörigen Nachricht ist, dann $e_i^x \Rightarrow e_j^y$.
- Wenn $e_i^x \Rightarrow e_j^y$ und $e_j^y \Rightarrow e_k^z$, dann $e_i^x \Rightarrow e_k^z$.

□

Wenn $e_i^x \Rightarrow e_j^y$, dann gilt e_j^y als *kausal abhängig* von e_i^x. Das bedeutet, dass e_j^y erst dann ausgeführt werden kann, wenn die Ausführung von e_i^x bereits beendet ist. Daher kann e_i^x als *Vorbedingung* zu e_j^y gesehen werden; es existiert ein möglicher Informationsfluss von e_i^x nach e_j^y. Beispielsweise bedeutet dies, dass eine Nachricht nicht empfangen werden kann bevor sie verschickt wurde.

Wenn dagegen gilt $e_i^x \not\Rightarrow e_j^y$ und $e_j^y \not\Rightarrow e_i^x$, dann sind e_i^x und e_j^y *nebenläufig*. In diesem Fall können die beiden Ereignisse parallel ausgeführt werden, denn keines von ihnen kann das jeweils andere in irgendeiner Weise beeinflussen: Es existiert kein potentieller Informationsfluss zwischen den beiden Ereignissen. Nebenläufigkeit wird im Folgenden mit $e_i^x \parallel e_j^y$ bezeichnet.

Hierbei ist folgendes zu beachten: Im Gegensatz zur Prozessordnung $\rightarrow$ beinhaltet die hier verwendete Definition der happened-before-Relation $\Rightarrow$ bereits die volle Transitivität. Im Folgenden wird für den Spezialfall, dass ein Ereignis e_i^x direkter, d. h. nicht-transitiver, Vorgänger von e_i^y im Sinne der happened-before-Relation ist, explizit eine andere Relation verwendet, nämlich die *directly-happened-before-Relation*:

Definition 3.2. Die directly-happened-before-Relation $\overset{\circ}{\Rightarrow}$ ist die kleinste Relation (im Sinne der Mächtigkeit), welche die folgenden Bedingungen erfüllt:

- Wenn $e_i^x \rightarrow e_i^y$, dann $e_i^x \overset{\circ}{\Rightarrow} e_i^y$.
- Wenn e_i^x ein Sendeereignis und e_j^y das Empfangsereignis der dazugehörigen Nachricht ist, dann $e_i^x \overset{\circ}{\Rightarrow} e_j^y$.

□

Die happened-before-Relation ist die Grundlage aller Konsistenzbetrachtungen auf der Basis logischer Zeit. Ein beliebiges System logischer Uhren kann nur dann korrekt sein und sinnvolle Rückschlüsse auf die Berechnung zulassen, wenn es die happened-before-Relation berücksichtigt. Auf der Grundlage dieser Relation formulierte Lamport daher die *clock condition* oder *Uhrbedingung*, die Grundanforderung an jede Art von logischer Uhr:

Definition 3.3. Jede korrekte Uhr muss die *clock condition* oder *Uhrbedingung* erfüllen. Diese lautet wie folgt:

$$\forall e_i^x, e_j^y \in E : \text{ wenn } e_i^x \Rightarrow e_j^y, \text{ dann } C(e_i^x) < C(e_j^y).$$

□

Da die Definition der Uhrbedingung direkt auf der Definition der happened-before-Relation basiert, ist es nur natürlich, dass sich die Erfüllung der Uhrbedingung zurückführen lässt auf die Erfüllung der Bedingungen der happened-before-Relation. Das heißt, um die Uhrbedingung zu erfüllen, muss ein korrektes Uhrensystem bei genauerer Betrachtung die beiden folgenden Bedingungen erfüllen:

1. Wenn $e_i^x \rightarrow e_i^y$, dann $C(e_i^x) < C(e_i^y)$.
2. Wenn e_i^x ein Sendeereignis und e_j^y das Empfangsereignis der dazugehörigen Nachricht ist, dann $C(e_i^x) < C(e_j^y)$.

Es ist leicht zu sehen, dass jede dieser beiden Bedingungen eine der Forderungen umsetzt, die die happened-before-Relation definieren. Der dritte Teil der Definition der happened-before-Relation ist hier nicht als Bedingung formuliert, da diese Forderung bereits in der Definition der Uhrbedingung selber zum Ausdruck kommt. Der besagte dritte Teil fordert lediglich die Transitivität der Relation, und diese ist in der Uhrbedingung bereits durch die inhärente Transitivität der bekannten Relation $<$ berücksichtigt.

Es ist wichtig zu beachten, dass diese Bedingungen, ebenso wie die happened-before-Relation nach Definition 3.1, vom zu Grunde gelegten Systemmodell abhängen und in der hier angegebenen Form nur für das in Abschnitt 3.1 beschriebene Systemmodell Gültigkeit besitzen. Andere Systemmodelle könnten durchaus andere Abhängigkeiten zwischen den Ereignissen etablieren, denen dann natürlich auch entsprechend Rechnung getragen werden müsste.

Ein Uhrensystem, welches die Uhrbedingung erfüllt, ist zwar „korrekt“ im bisher verwendeten Sinne, lässt aber nur sehr begrenzte Rückschlüsse auf die Kausalität der Berechnung zu, da diese unter Umständen nicht vollständig erfasst wird. Einen Hinweis darauf liefert bereits die Formulierung der Uhrbedingung selbst: Die Korrektheit des Schlusses wird nur

in einer Richtung gefordert. Die Folgen für ein Uhrsystem, das lediglich die Uhrbedingung in der angegebenen Form erfüllt, können in der Besprechung skalarer Uhren nachgelesen werden (siehe Abschnitte 3.4.3, 3.4.4 und 3.4.6).

Uhren, die die Ereigniskausalität vollständig erfassen sollen, müssen eine stärkere Forderung erfüllen, die sogenannte *strong clock condition* oder *starke Uhrbedingung*. Diese definiert sich naheliegenderweise durch die zusätzliche Forderung nach dem Umkehrschluss:

Definition 3.4. Die *strong clock condition* oder *starke Uhrbedingung* lautet wie folgt:

$$\forall e_i^x, e_j^y \in E : \; C(e_i^x) < C(e_j^y) \text{ genau dann, wenn } e_i^x \Rightarrow e_j^y.$$

□

3.2.1 Allgemeine Funktionsweise logischer Uhren

Das Führen logischer Zeit bedeutet, jedem Ereignis im System einen Uhrwert oder Zeitstempel zuzuordnen. Dieser Zeitstempel muss eindeutig sein, da sonst die Uhrbedingung verletzt würde. Jeder Prozess p_i im System unterhält seine eigene Uhr C_i und aktualisiert sie mit jedem seiner Ereignisse. Damit die Uhrbedingung stets erfüllt ist, müssen sowohl die Aktualisierung als auch die Auswertung der Uhren C_i nach aufeinander abgestimmten Regeln erfolgen. Diese Regeln sind für die verschiedenen Arten logischer Uhren individuell unterschiedlich und werden in den entsprechenden Abschnitten erklärt.

Da es der Sinn logischer Uhren ist, die kausalen Abhängigkeiten zwischen den Ereignissen zu erfassen, hängt der Uhrwert, der einem Ereignis zugeordnet werden muss, von den Uhrwerten seiner kausalen Vorgänger ab. Das bedeutet, dass die Uhrwerte aller direkten kausalen Vorgänger eines Ereignisses bekannt sein müssen, um diesem seinen individuellen Zeitstempel zuordnen zu können.

Im vorliegenden Systemmodell (siehe Abschnitt 3.1) kann jedes Ereignis entweder einen oder zwei Vorgänger haben: einen obligatorischen Vorgänger nach der Prozessordnung und eventuell, nämlich im Falle eines Empfangsereignisses, zusätzlich ein externes Sendeereignis.* Der Zeitstempel des Vorgängers nach der Prozessordnung ist immer bekannt, da es sich einfach um den aktuellen lokalen Uhrwert des Prozesses handelt. Schwieriger ist die Bereitstellung des Zeitstempels eines externen Sendeereignisses. Hier besteht die Lösung

* Zudem gibt es auch Ereignisse ohne Vorgänger, nämlich die Startereignisse der Prozesse. Diesen Ereignissen wird immer der niedrigste mögliche Zeitstempel zugeordnet, den das entsprechende Uhrensystem vorsieht, wie auch immer ein solcher Zeitstempel im jeweiligen Fall aussehen mag. Für die hier vorgestellten Arten logischer Uhren wird dieser Wert jeweils im entsprechenden Abschnitt angegeben.

darin, dass der Uhrwert des Absenderprozesses einer Nachricht zusammen mit der Nachricht an den Empfängerprozess übermittelt wird. Daher wird jede Nachricht beim Versand mit dem aktuellen Zeitstempel des Absenderprozesses versehen.

3.3 Grundlegende Anwendungen logischer Uhren

In diesem Unterabschnitt sollen kurz einige der wichtigsten grundlegenden Ziele vorgestellt werden, die den Einsatz logischer Uhren rechtfertigen, motivieren oder notwendig machen. „Grundlegend“ soll hier zum Ausdruck bringen, dass diese Ziele nicht unbedingt Anwendungen per se darstellen müssen, sondern normalerweise als Erfordernis für eine solche auftreten, etwa für Monitoring, Breakpointing oder Debugging. In den folgenden Kapiteln wird in unterschiedlichen Zusammenhängen auf die hier eingeführten Begriffe Bezug genommen.

Das erste wichtige Konzept, das hier in diesem Sinne kurz besprochen werden soll, ist die sogenannte *delivery rule* des Nachrichtensystems. Mangels eines etablierten deutschen Begriffs wird hier lediglich die englische Variante verwendet. Gemeint ist die Regel, welche der Auslieferung von Nachrichten an den jeweiligen Empfängerprozess zu Grunde liegt, und zwar im Hinblick auf die Reihenfolge, in der der Empfängerprozess seine Nachrichten zugestellt bekommt. Es handelt sich also um Einschränkungen der Reihenfolge, in der ein Empfängerprozess die bei ihm eingehenden Nachrichten verarbeiten kann.

Ein wichtiges und häufig verwendetes Beispiel ist *FIFO delivery*. Diese Regel besagt, dass jeder Prozess p_j die Nachrichten, die ihm ein beliebiger Prozess p_i schickt, in der Reihenfolge ihrer Versendung zu erhalten hat. Über die Einordnung von Nachrichten anderer Absenderprozesse wird indes keine Aussage getroffen. Formal lautet die Regel also

$$\text{Wenn } e_i^s \rightarrow e_i^S, \text{ dann } e_j^r \rightarrow e_j^R,$$

wobei s und S Sendeereignisse markieren, und r und R die jeweils dazugehörigen Empfangsereignisse. Diese delivery rule ordnet nur die Nachrichten desselben Absenders an denselben Empfänger. Über die Ordnung von Nachrichten verschiedener Absender wird nichts ausgesagt. Es gibt auch andere Definitionen von FIFO delivery, die fordern, dass alle Nachrichten im System in der Reihenfolge ihrer Versendung abgeliefert werden müssen. Diese Forderung soll hier als *strict FIFO delivery* bezeichnet werden.*

* Viele verteilte Systeme gewährleisten FIFO delivery bereits als Leistungsmerkmal ihres Nachrichtenaustauschsystems, unterhalb der Anwendungsebene. In diesem Fall spricht man von Nachrichtenaustausch über die *FIFO channels* oder *FIFO-Kanäle* des Systems.

Eine andere wichtige delivery rule ist die sogenannte *causal delivery*. Diese Regel ist deutlich stärker als FIFO delivery und lautet folgendermaßen:

$$\text{Wenn } e_i^s \Rightarrow e_k^S, \text{ dann } e_j^r \rightarrow e_j^R.$$

Die FIFO delivery wird hier dahingehend erweitert, dass auch Nachrichten unterschiedlicher Absenderprozesse geordnet ausgeliefert werden sollen, sofern zwischen ihnen eine kausale Beziehung besteht. Die kausale Beziehung ist dabei wieder durch die happened-before-Relation gegeben. Diese delivery rule ist besonders wichtig in allen Anwendungen, die eine Beobachtung des Systemablaufs erfordern, etwa Debugging oder Monitoring. Dies liegt daran, dass diese Regel eine Erfassung der Ereigniskausalitäten erlaubt. Ein Monitor-Prozess kann die Meldungen über stattfindende Ereignisse nur dann richtig ordnen und wiedergeben, wenn er sie, die ja auch nichts anderes sind als unwägbaren Transportverzögerungen unterworfene Nachrichten, entsprechend ihrer kausalen Abhängigkeiten zu ordnen in der Lage ist.

Ein weiteres grundlegendes und wünschenswertes Leistungsmerkmal logischer Uhren ist die sogenannte *gap detection* oder *Lückenerkennung*. Diese wird beispielsweise (aber nicht ausschließlich) zur Implementierung einer causal delivery benötigt, wie sie soeben beschrieben wurde.

Die Lückenerkennung besagt, dass ein Prozess, welcher zwei Nachrichten erhält, anhand ihrer Zeitstempel entscheiden können sollte, ob zwischen den beiden Nachrichten eine dritte Nachricht erwartet werden muss, die er aufgrund der unwägbaren Verzögerungen auf dem Transportweg womöglich noch nicht erhalten hat. Formal formuliert heißt das:

Definition 3.5. Ein System logischer Uhren hat die Eigenschaft der *gap detection* oder *Lückenerkennung*, wenn gilt: Falls zwei Zeitstempel $C_i(e_i^x) < C_j(e_j^y)$ gegeben sind, kann entschieden werden, ob ein Ereignis e_k^z existiert, für das gilt:

$$C_i(e_i^x) < C_k(e_k^z) < C_j(e_j^y).$$

□

Eine ausführlichere Diskussion dieses Themas findet sich in [5]. Im Folgenden werden wir sehen, dass eine vollständige Lückenerkennung nur in den seltensten Fällen erreicht werden kann.

Nach der Einführung der obigen Konzepte können nun effizient verschiedene Typen von logischen Uhren einschließlich ihrer jeweiligen Leistungsmerkmale beleuchtet werden. Die beiden folgenden Unterkapitel behandeln die bis dato gängigen Typen von logischen Uhren, nämlich die ursprüngliche skalare logische Uhr von Leslie Lamport [1] und ihre Weiterentwicklung, die bekannte Vektor-Uhr [9, 10, 11]. Diese Uhren haben in der heutigen Praxis

einen schwerwiegenden Nachteil, der darin besteht, dass sie entweder die logische Zeit nur unvollständig wiedergeben, oder aber inhärent ungeeignet sind für Anwendungen mit dynamischer Prozesserzeugung. Die jeweiligen Ursachen dafür werden im Folgenden ebenfalls erörtert.

Beiden genannten Typen von Uhren liegt dasselbe stark abstrahierte und damit sehr allgemein anwendbare Systemmodell zu Grunde, so dass diese logischen Uhren in nahezu jedem beliebigen System genutzt werden können. Dieses Modell wurde in Abschnitt 3.1 bereits erläutert.

3.4 Skalare Uhren

Lamports skalare Uhren waren ein erster Schritt in die Richtung, den Fortschritt einer Berechnung logisch und nicht realzeitorientiert zu erfassen. Sie wurden derart gestaltet, dass sie die Uhrbedingung erfüllen. Im Folgenden werden Funktionsweise und Leistungsfähigkeit dieses einfachsten Systems logischer Uhren erörtert.

3.4.1 Funktionsweise

In [1] definiert Lamport sein Uhrsystem als eine Funktion, die jedem Ereignis einen einfachen skalaren Wert zuordnet:

Definition 3.6. Eine *logische Uhr* C ist eine Funktion, die jedem Ereignis $e \in E$ eine Zahl $C(e)$ zuordnet. □

Die Zuordnung selbst erfolgt nach den grundlegenden Prinzipien, die in Abschnitt 3.2.1 bereits erklärt wurden, sowie nach den für skalare Uhren spezifischen Regeln, die im folgenden Unterabschnitt aufgezeigt werden.

3.4.2 Aktualisierung

Damit die Uhrbedingung stets erfüllt ist, werden die skalaren Uhren C_i nach den zwei folgenden Regeln geführt:

- Wenn e_i^x kein Empfangsereignis ist, dann $C_i(e_i^x) := C_i(e_i^{x-1}) + 1$.
- Wenn e_i^x das Empfangsereignis einer Nachricht ist, die mit e_j^y verschickt wurde, dann

$$C_i(e_i^x) := \max\left(C_i(e_i^{x-1}),\ C_j(e_j^y)\right) + 1.$$

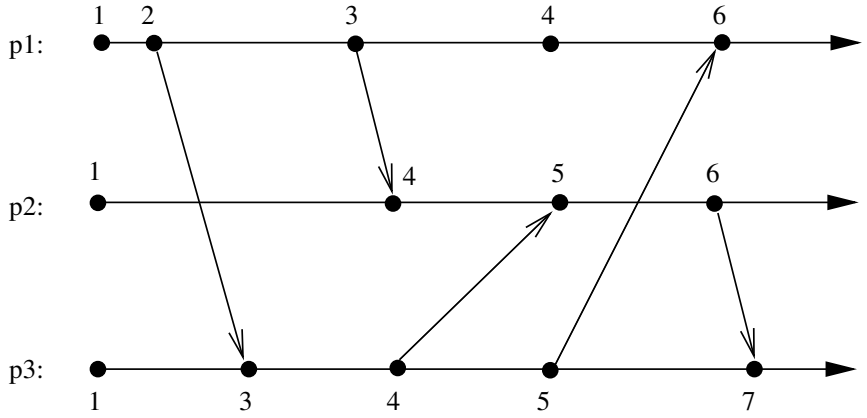

Abbildung 3.1: Skalare Uhren

Damit sind alle im hier verwendeten Systemmodell vorliegenden Möglichkeiten kausaler Abhängigkeit berücksichtigt (vergleiche Abschnitte 3.1 und 3.2.1).

Ereignisse ohne Vorgänger erhalten o. B. d. A. den Uhrwert 1.*

Ein Beispiel zur Funktionsweise skalarer Uhren ist in Abbildung 3.1 gegeben.

3.4.3 Auswertung

Dass die skalaren Uhren die Uhrbedingung erfüllen, ist leicht zu verifizieren. Dennoch sind sie problematisch, und zwar deshalb, weil sie nicht die starke Uhrbedingung erfüllen und somit die Kausalitätsbeziehungen zwischen den Ereignissen nur zum Teil erfassen, wie bereits die folgenden einfachen Richtlinien zur Auswertung deutlich machen:

Aus dem Vergleich zweier Zeitstempel $C_i(e_i^x) = C_j(e_j^y)$ kann man ohne weiteres schließen, dass $(e_i^x) \parallel (e_j^y)$. Die umgekehrte Folgerung ist jedoch trügerisch: Wenn $C_i(e_i^x) < C_j(e_j^y)$, ist es nicht möglich, daraus abzuleiten, ob $(e_i^x) \Rightarrow (e_j^y)$ oder $(e_i^x) \parallel (e_j^y)$. Die Möglichkeit zu dieser Folgerung ist jedoch oft, gerade etwa beim verteilten Debugging, von zentraler Bedeutung.

3.4.4 Lückenerkennung

Eine Folge der unvollständigen Kausalitätserfassung (siehe Abschnitt 3.4.3) ist die, dass skalare Uhren von vornherein ungeeignet sind für jede Art von Lückenerkennung. Dass

* Selbstverständlich könnte man auch mit einer beliebigen anderen Zahl anfangen zu zählen.

Lücken zwischen zwei Zeitstempeln nicht erkannt werden können (im Sinne von Definition 3.5), lässt sich schon anhand des kleinen Beispiels in Abbildung 3.1 leicht nachvollziehen.

3.4.5 Effizienz

In diesem Abschnitt soll ein kurzer Überblick über den Speicher- und Laufzeitbedarf von skalaren Uhren gegeben werden.

Platzbedarf

Jeder Zeitstempel benötigt einen einzigen Skalar, unabhängig davon, wie viele Prozesse im System vorhanden sind. Der Platzbedarf ist also konstant (O(1)).

Zeitbedarf

Auch der Zeitbedarf für die Aktualisierungs- und Auswertungsalgorithmen ist konstant. In allen Fällen müssen bis zu zwei skalare Werte betrachtet, und im Falle des Updates eine Inkrementierung durchgeführt werden. Ein Lückenerkennungsalgorithmus ist nicht gegeben.

3.4.6 Leistungsfähigkeit

Da die skalaren Uhren zwar die Uhrbedingung aber nicht die starke Uhrbedingung erfüllen, liegt hier, wie bereits im Unterabschnitt zur Auswertung skalarer Uhren festgestellt wurde, ein Verlust von Kausalitätsinformation vor. Dieser mag zwar für manche Anwendungen akzeptabel sein, im Allgemeinen ist er es jedoch nicht.

Aufgrund dieser Eigenschaft beschränkt sich die Mächtigkeit skalarer Uhren auf Anwendungen, bei denen die Konstruktion einer Ereignisordnung im Vordergrund steht, die konsistent mit den kausalen Abhängigkeiten ist. Skalare Uhren sind jedoch nicht geeignet für Anwendungen, die die Betrachtung oder Analyse der Abhängigkeiten selbst erfordern. Beispielsweise kann man nicht entscheiden, ob zwei Ereignisse in einer konsistenten Ordnung vertauscht werden könnten, ohne die Konsistenz zu verletzen. Eine derartige Fragestellung tritt beispielsweise dann auf, wenn entschieden werden soll, ob zwei Programmteile parallel ausgeführt werden können oder nicht, ob es also z. B. sinnvoll wäre, sie auf unterschiedlichen Knoten auszuführen (Platzierungsentscheidung).

Darüber hinaus macht die fehlende Möglichkeit zur Lückenerkennung skalare Uhren inhärent ungeeignet für alle Anwendungen, die causal delivery erfordern (dazu gehören Monitoring und Debugging), sowie für alle Anwendungen, die zwar nur FIFO delivery benötigen, aber auf Systemen laufen, die nicht bereits fertige FIFO-Kanäle bereitstellen.

3.5 Vektor-Uhren

Vektor-Uhren sind eine Erweiterung von Lamports skalaren Uhren und werden häufig eingesetzt. Sie wurden von Mattern [9] und Fidge [10, 11] (unabhängig voneinander) mit dem Ziel entwickelt, die Schwächen und Defizite von Lamports skalaren Uhren zu beseitigen, d. h. die Ereigniskausalität einer verteilten Berechnung uneingeschränkt widerzuspiegeln. Daher sind sie leistungsfähiger als skalare Uhren.

Analog zur Erörterung von skalaren Uhren im vorangegangenen Abschnitt 3.4 werden in den folgenden Unterabschnitten Funktionsweise und Leistungsfähigkeit von Vektor-Uhren diskutiert. Dabei wird sich zeigen, dass die Lösung der Probleme skalarer Uhren ihrerseits wieder neue Problematiken aufwirft.

3.5.1 Funktionsweise

Wie bei allen logischen Uhren unterhält auch hier jeder Prozess p_i seine eigene lokale Uhr C_i (vgl. Abschnitt 3.2.1). Ein Uhrwert besteht jedoch nun nicht mehr aus einem einzigen skalaren Wert für *alle* Prozesse, sondern aus einem Skalar für *jeden* Prozess im System. Bei n Prozessen ergibt sich als Zeitstempel also ein n-dimensionaler Vektor.

Die Zuordnung zwischen einem Prozess und dem dazugehörigen Skalar erfolgt über die Position des Eintrags im Vektor. Damit das möglich ist, muss ein globaler Namensraum über den Prozessen gegeben sein, was im hier verwendeten Systemmodell (siehe Abschnitt 3.1) schon implizit dadurch der Fall ist, dass die Prozesse eindeutig von 1 bis n durchnummeriert sind.

Auf diese Weise spiegelt jeder Zeitstempel jeweils den letzten Informationsstand wider, der dem zugehörigen Prozess zum Zeitpunkt des entsprechenden Ereignisses über jeden einzelnen anderen Prozess vorlag: Jede Vektorkomponente $C_i(e_i^x)[j]$ enthält den letzten (skalaren) Uhrwert, den Prozess i zum Zeitpunkt von Ereignis e_i^x von Prozess j weiß. Damit ist die kausale Historie eines jeden Ereignisses vollständig erfasst, und die Vektor-Uhren sind prinzipiell in der Lage, die starke Uhrbedingung zu erfüllen. Dazu müssen natürlich noch passende, aufeinander abgestimmte Algorithmen zur Aktualisierung und zur Auswertung der Zeitstempel festgelegt werden. Diese werden in den folgenden Unterabschnitten erklärt.

3.5.2 Aktualisierung

Damit die (starke) Uhrbedingung stets erfüllt ist, werden die Vektor-Uhren C_i nach den zwei folgenden Regeln geführt:

- Wenn e_i^x das Empfangsereignis einer Nachricht ist, die mit e_j^y verschickt wurde, dann

$$C_i(e_i^x) := \overline{\max}\left(C_i(e_i^{x-1}),\ C_j(e_j^y)\right),$$

  wobei $\overline{\max}$ die komponentenweise Maximumsfunktion ist.

- Wenn e_i^x irgendein Ereignis ist (Empfangsereignisse eingeschlossen), dann inkrementiert der Prozess seinen eigenen skalaren Uhrwert:

$$C_i(e_i^x)[i] := C_i(e_i^{x-1})[i] + 1.$$

  Im Falle eines Empfangsereignisses erfolgt dies *zusätzlich* zur ersten Regel.

Damit sind alle im hier verwendeten Systemmodell vorliegenden Möglichkeiten kausaler Abhängigkeit berücksichtigt (vergleiche Abschnitte 3.1 und 3.2.1).

O. B. d. A. beginnen alle Prozesse bei 1 mit dem Zählen, so dass ein Ereignis e_i^1 als Zeitstempel immer einen Vektor erhält, dessen i-ter Eintrag 1 ist, und alle anderen Null*.

Ein Beispiel zur Funktionsweise von Vektor-Uhren ist in Abbildung 3.2 gegeben.

3.5.3 Auswertung

Die Auswertung von Zeitstempeln basiert grundsätzlich auf dem Vergleich zweier Zeitstempel. Ein solcher Vergleich kann prinzipiell drei Ergebnisse liefern:

1. Die beiden Zeitstempel können gleich sein.
2. Einer der beiden Zeitstempel kann kleiner sein als der andere.
3. Es kann sein, dass weder das eine noch das andere der Fall ist.

Die Semantik dieser drei Fälle wird durch die starke Uhrbedingung folgendermaßen erklärt:

---

* Die Null repräsentiert den Uhrwert eines Prozesses, von dem der Prozess, der die Uhr führt, noch keinen anderen Wert empfangen hat.

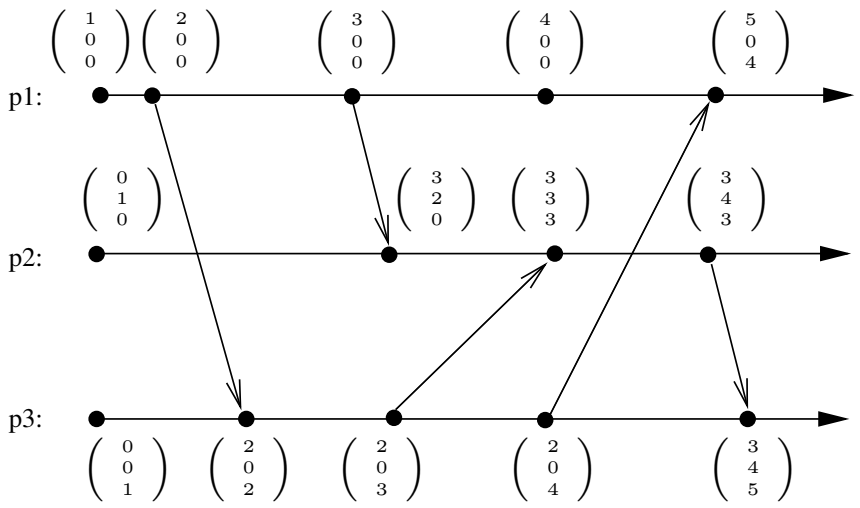

Abbildung 3.2: Vektor-Uhren

1. Da jeder Uhrwert in eindeutiger Weise nur einem einzigen Ereignis zugeordnet sein kann, bedeuten zwei gleiche Zeitstempel, dass sie dasselbe Ereignis bezeichnen. Eine Aussage über eine gegenseitige Abhängigkeit kann daher nicht getroffen werden; es ist überhaupt nur ein Ereignis im Spiel.

2. Das Ereignis mit dem größeren der beiden Zeitstempel ist als kausal abhängig vom anderen einzustufen.

3. Es liegen zwei Ereignisse vor, die voneinander unabhängig sind.

Mit „Auswertung“ ist nun die Frage gemeint: Wie bestimmt man, welcher der drei Fälle vorliegt? Im Fall der skalaren Uhren (siehe Abschnitt 3.4.3) wurden dazu ohne weitere Erklärung die entsprechenden herkömmlichen Relationen für skalare Werte angewandt: $=$ und $<$.* Nun, da es sich bei den Zeitstempeln nicht mehr um einfache Skalare handelt, bedürfen die Relationen $=$ und $<$ einer genaueren Erklärung.

Definition 3.7. Die Relation $=$ wird für Vektor-Zeitstempel folgendermaßen definiert:

$$C_i(e_i^x) = C_j(e_j^y) \text{ genau dann, wenn } \forall k | 1 \leq k \leq n : C_i(e_i^x)[k] = C_j(e_j^y)[k].$$

□

* Die drei möglichen Ergebnisse konnten in diesem Fall gar nicht recht unterschieden werden, da die skalaren Uhren nicht die starke Uhrbedingung erfüllen.

Definition 3.8. Die Relation $<$ wird für Vektor-Zeitstempel folgendermaßen definiert:

$$C_i(e_i^x) < C_j(e_j^y) \text{ genau dann, wenn } \begin{cases} C_i(e_i^x) \neq C_j(e_j^y) \text{ und} \\ \forall k | 1 \leq k \leq n : C_i(e_i^x)[k] \leq C_j(e_j^y)[k]. \end{cases}$$

□

Mittels dieser Relationen lässt sich nun für zwei gegebene Zeitstempel entscheiden, in welchem der drei möglichen Verhältnisse sie zueinander stehen. Für Fall drei gilt dabei logischerweise (wie auch für alle anderen logischen Uhren, die die starke Uhrbedingung erfüllen):

Korollar 3.1.

$$e_i^x \parallel e_j^y \text{ genau dann, wenn } \begin{cases} C_j(e_j^y) \not\leq C_i(e_i^x) \text{ und} \\ C_i(e_i^x) \not\leq C_j(e_j^y). \end{cases}$$

□

An dieser Stelle soll noch darauf hingewiesen werden, dass diese mathematischen Definitionen, wenn sie direkt so umgesetzt werden, keineswegs die effizienteste Implementierung darstellen. Wenn n die Anzahl der Vektorkomponenten ist (und damit gleichzeitig die Anzahl der Prozesse), benötigt der soeben angegebene Nebenläufigkeitstest $2n$ skalare Vergleiche. Angesichts der in Abschnitt 3.5.2 angegebenen Aktualisierungsregeln lässt sich dieser lineare Aufwand sehr einfach auf konstante 2 Vergleiche reduzieren, denn es gelten die beiden folgenden Aussagen:

Korollar 3.2.

$$e_i^x \Rightarrow e_j^y \text{ genau dann, wenn } C_i(e_i^x)[i] \leq C_j(e_j^y)[i],$$

□

Korollar 3.3.

$$e_i^x \parallel e_j^y \text{ genau dann, wenn } \begin{cases} C_j(e_j^y)[i] < C_i(e_i^x)[i] \text{ und} \\ C_i(e_i^x)[j] < C_j(e_j^y)[j]. \end{cases}$$

□

Die Formel in Korollar 3.2 ist nicht symmetrisch. Wenn ihre Anwendung $e_i^x \not\Rightarrow e_j^y$ liefert, muss sie gegebenenfalls noch ein zweites Mal angewendet werden, um entscheiden zu können, ob $e_j^y \Rightarrow e_i^x$ oder $e_i^x \parallel e_j^y$. Im Gegensatz dazu brauchen die angegebenen Nebenläufigkeitstests, sowohl der effiziente aus Korollar 3.3 als auch der ineffiziente aus Korollar 3.1, nur einmal angewendet zu werden, um das genaue Verhältnis der beiden mit den Zeitstempeln assoziierten Ereignisse bestimmen zu können: Wenn einer der beiden Nebenläufigkeitstests angewendet wird und $e_i^x \not\parallel e_j^y$ liefert, kann sofort entschieden werden, ob $e_i^x \Rightarrow e_j^y$ oder $e_j^y \Rightarrow e_i^x$. Dazu muss lediglich in Betracht gezogen werden, welcher der beiden Bedingungsterme unwahr gewesen ist.

3.5.4 Lückenerkennung

Im Gegensatz zu skalaren Uhren (siehe Abschnitt 3.4.4), sind Vektor-Uhren immerhin in der Lage, eine eingeschränkte Form der Lückenerkennung zu erlauben. Diese Fähigkeit basiert auf der Eigenschaft der Vektor-Uhren, die starke Uhrbedingung zu erfüllen. Aus der starken Uhrbedingung nämlich folgt, in Verbindung mit den Aktualisierungsregeln für Vektor-Uhren aus Abschnitt 3.5.2, die folgende eingeschränkte Eigenschaft der Lückenerkennung:

Definition 3.9. Als *weak gap detection* oder *eingeschränkte Lückenerkennung* wird die folgende Eigenschaft bezeichnet:

$$\text{Wenn } \exists\, k \neq j :\ C_i(e_i^x)[k] < C_j(e_j^y)[k],\ \text{dann } \exists\, e_k^z :\ (e_k^z \Rightarrow e_j^y) \wedge \neg(e_k^z \Rightarrow e_i^x).$$

□

Diese Eigenschaft stellt insofern eine Einschränkung der Lückenerkennung dar, wie sie in Abschnitt 3.3 eingeführt wurde, als dass sie eine Entscheidung über $e_i^x \Rightarrow e_k^z \Rightarrow e_j^y$ ausschließlich für den Spezialfall $k = i$ zulässt.

Unter Berücksichtigung der genannten Einschränkungen kann über eine Lücke zwischen zwei gegebenen Zeitstempeln folgendermaßen entschieden werden:

Korollar 3.4. Für Vektor-Uhren gilt:

$$\exists\, e_i^z \mid e_i^x \Rightarrow e_i^z \Rightarrow e_j^y \text{ genau dann, wenn } C_i(e_i^x)[k] < C_j(e_j^y)[k] \text{ für irgendein } k \neq j.$$

□

Diese eingeschränkte Form der Lückenerkennung ist immerhin ausreichend für die Implementierung von FIFO delivery oder gar causal delivery für einen dedizierten Monitorprozess. Voraussetzung dazu ist allerdings, dass die Uhren aller Prozesse nur bei Ereignissen aktualisiert werden, über welche der Monitorprozess benachrichtigt wird. Oder andersherum: Der Monitorprozess muss über *alle* Ereignisse im System informiert werden, um die Nachrichten in kausaler Reihenfolge entgegennehmen zu können.

Eine ausführlichere Abhandlung über die Eigenschaft eingeschränkter Lückenerkennung und ihre Verwendung zur Implementierung von causal delivery findet sich in [5].

3.5.5 Effizienz

In diesem Abschnitt soll ein kurzer Überblick über den Speicher- und Laufzeitbedarf von Vektor-Uhren gegeben werden. Dabei orientiert sich der Laufzeitbedarf für die Auswertungs- und Aktualisierungs-Algorithmen naturgemäß am Platzbedarf der Zeitstempel (hier Vektoren).

Platzbedarf

Der Platzbedarf für einen Vektor-Zeitstempel hängt linear von der Anzahl der Prozesse ab. Es existiert jeweils ein Eintrag für jeden Prozess, so dass die Zeitstempelgröße bei n Prozessen immer genau n beträgt.

Zeitbedarf

Für die Aktualisierung von Zeitstempeln sind, folgend aus der Fallunterscheidung in der Aktualisierungsregel, zwei Fälle zu unterscheiden. Die Aktualisierung von Empfangsereignissen benötigt Zeit in Größenordnung $O(n)$, da im Wesentlichen das komponentenweise Maximum zu bilden ist. Bei allen anderen Ereignissen genügt das Inkrementieren eines einzigen Vektoreintrags. Dieser kann über seinen Index direkt angesprungen werden, so dass der Vektor nicht durchsucht werden muss und ein konstanter Zeitaufwand vorliegt ($O(1)$).

Die Auswertung der Zeitstempel kann über den Nebenläufigkeitstest ebenfalls mit konstantem Zeitaufwand erfolgen, wie bereits in Abschnitt 3.5.3 dargelegt.

Die Lückenerkennung benötigt einen Zeitaufwand von $O(n)$, da hier alle Vektoreinträge betrachtet werden müssen.

3.5.6 Leistungsfähigkeit

Vektor-Uhren, als Weiterentwicklung der in Abschnitt 3.4 vorgestellten skalaren Uhren, weisen nicht deren Problem eines inhärenten kausalitätsbezogenen Informationsverlustes auf; sie erfassen die kausalen Abhängigkeiten zwischen den Ereignissen einer verteilten Berechnung vollständig. Damit sind sie nicht nur geeignet zur Konstruktion einer konsistenten totalen Ereignisordnung (wie auch die skalaren Uhren), sondern auch für alle Anwendungen, die in irgendeiner Form die Analyse der Abhängigkeiten selbst erfordern, beispielsweise für die bereits in Abschnitt 3.4.6 angesprochene Platzierungsentscheidung.

Die Eigenschaft der eingeschränkten Lückenerkennung macht sie außerdem anwendbar zur Implementierung von FIFO delivery oder causal delivery, etwa zu Monitoring- oder Debuggingzwecken – wenn auch unter einschränkenden Voraussetzungen, wie in Abschnitt 3.5.4 beschrieben.

Nichtsdestotrotz besitzen Vektor-Uhren einen großen Nachteil (der übrigens bei den skalaren Uhren nicht vorhanden ist). Vektor-Uhren führen in einem jeden Zeitstempel einen skalaren Uhrwert für jeden im System vorhandenen Prozess mit. Das Problem, das aus

dieser Konstruktion entsteht, äußert sich in zwei schwerwiegenden Voraussetzungen, über die bisher stillschweigend hinweggesehen wurde:

- Die Anzahl der Prozesse im System ist konstant, d. h. sie ändert sich nicht im Laufe der Berechnung.
- Die Anzahl der Prozesse ist a priori bekannt.

Dass diese Voraussetzungen, insbesondere die erste, für moderne verteilte und kooperierende Rechensysteme eigentlich nur in Spezialfällen anwendbar sind, liegt auf der Hand. Selbst in herkömmlichen Desktop- oder sogar Single-User-Systemen ist das Erzeugen und Beenden von Prozessen bereits seit geraumer Zeit Standard. Für die Vektor-Uhren würde dies jedoch bedeuten, dass die Dimension der Vektoren im Laufe der Berechnung zunehmen und auch wieder abnehmen müsste. Dies jedoch ließe sich nicht mit der in den vorangegangenen Abschnitten dargelegten Funktionsweise der Uhren vereinbaren. Die Zuordnung der Vektoreinträge zu den Prozessen wäre nicht mehr eindeutig.

Die Lösung dieses Problems war das Ziel der für die im Rahmen dieser Arbeit vorgenommene Entwicklung zweier besserer Systeme logischer Uhren, die in den Kapiteln 4 und 5 vorgestellt werden. Es sollte eine Möglichkeit zur vollständigen Erfassung der Kausalität in Systemen geschaffen werden, die die dynamische Erzeugung und Beendigung von Prozessen erlauben.

Kapitel 4

Dynamische Vektor-Uhren

In Kapitel 3 wurden die beiden gängigen Arten logischer Uhren vorgestellt. Außer ihrer Funktionsweise und den entsprechenden Algorithmen wurde insbesondere ihre jeweilige Leistungsfähigkeit und ihre Eignung für verschiedene Applikationen beleuchtet. Die dabei erläuterten Defizite dieser Uhren bilden die Motivation zur Entwicklung der beiden verbesserten Uhrensysteme, die in diesem und dem folgenden Kapitel vorgestellt werden. Daher werden die wichtigsten Punkte der Leistungsmerkmale von skalaren und Vektor-Uhren im Rahmen der Zielerfassung im folgenden Abschnitt noch einmal zusammengefasst. Danach erfolgt die Erläuterung des ersten der beiden neuen Systeme, der *dynamischen Vektor-Uhren* oder *dynamic vector clocks*. Der Aufbau des Kapitels ist dabei im Wesentlichen analog zur Vorstellung der skalaren Uhren und der Vektor-Uhren in Kapitel 3.

4.1 Ziel

In Abschnitt 3.4 wurden die ersten logischen Uhren, Lamports skalare Uhren, beschrieben. Im Zuge dessen wurde erörtert, dass sie für die meisten Anwendungen unzureichend sind, weil sie die Kausalitätsbeziehungen der Ereignisse nur unvollständig erfassen.

In Abschnitt 3.5 wurden die sogenannten Vektor-Uhren erläutert. Diese stellen gegenüber den skalaren Uhren eine deutliche Verbesserung dar und weisen deren Nachteile nicht auf. Der Preis für die höhere Leistungsfähigkeit besteht in den Voraussetzungen, die ein System bzw. eine Anwendung für den Einsatz von Vektor-Uhren erfüllen muss: Die Anzahl der Prozesse im System muss konstant und bereits im Vorfeld bekannt sein. Diese Voraussetzungen sind für die Mehrzahl moderner Applikationen schlichtweg inakzeptabel, da der Verzicht auf die dynamische Erzeugung von Prozessen zur Laufzeit heutzutage nicht mehr zeitgemäß wäre.

Daher sind verbesserte logische Uhren erforderlich, welche die genannten Schwachpunkte der herkömmlichen Uhren nicht aufweisen. Das bedeutet insbesondere zweierlei:

- Die Uhren sollen in der Lage sein, die kausalen Abhängigkeiten unter den Ereignissen vollständig, d. h. mindestens* im selben Umfang zu erfassen, wie Vektor-Uhren.
- Die Uhren sollen in der Lage sein, neu erzeugte Prozesse in ihren Zeitstempeln zu berücksichtigen, ohne die Vergleichbarkeit der Zeitstempel für eine konsistente Auswertung zu verlieren.

Ein erster, besonders nahe liegender Ansatz zum Erreichen dieser Ziele, der im Rahmen der vorliegenden Arbeit verfolgt und ausgearbeitet wurde, wird im weiteren Verlauf dieses Kapitels vorgestellt: *dynamische Vektor-Uhren*.

4.2 Lösungsansatz und Funktionsweise

Das Problem von Vektor-Uhren mit dynamischen Prozesssystemen resultiert aus der Datenstruktur des Vektor-Zeitstempels. Der Zeitstempel besteht aus einem Vektor fester Dimension mit einem skalaren Eintrag pro Prozess. Soweit würde noch nichts dagegen sprechen, neu auftretende Prozesse einfach durch Inkrementierung der Dimension des Vektors als neuen Eintrag anzuhängen. Die Zuordnung zwischen Prozess und dem zugehörigen Vektoreintrag erfolgt jedoch über die (feste) Position des Eintrags im Vektor. Um diese Abbildung bei veränderlicher Dimension des Vektors nicht zu kompromittieren, dürfte man den Uhrwert von Prozess p_j, von dem Prozess p_i etwa mittels einer Nachricht Kenntnis erhält, nicht einfach an p_is Vektor anhängen (und damit dessen Dimension um 1 vergrößern). Dadurch ginge die Information verloren, dass dieser Uhrwert zu Prozess p_j gehört. Konsequenterweise müsste man den Eintrag, wie bei Standard-Vektor-Uhren auch, an derjenigen Indexposition vornehmen, die der systemweit eindeutigen Nummer des Prozesses entspricht, in diesem Fall j.† Angenommen, die erste Nachricht, die p_1 erhält, stammt von p_{337}. Dann müsste der Uhrvektor von p_1 zwischen der ersten und der 337. Position mit 335 Nullen aufgefüllt werden. Dies würde offensichtlich einen signifikanten Overhead bedeuten, und zwar nicht nur im Speicherbedarf, sondern insbesondere in der Netzwerklast: Dieser Vektor würde zusammen mit jeder Nachricht übertragen, die p_1 in Zukunft verschickt – und die Verschwendung wäre ebenso unbeschränkt wie die Anzahl der Prozesse. Darüber hinaus wäre man damit von vornherein jeder Möglichkeit beraubt, Vektorkomponenten

* Wünschenswert wäre natürlich noch eine uneingeschränkte Lückenerkennung.
† Eine solche ist in dem hier verwendeten Systemmodell vorhanden, siehe 4.3 bzw. 3.1.

später wieder zu löschen, die deshalb nicht mehr benötigt werden, weil der zugehörige Prozess in der Zwischenzeit beendet wurde. Genauer gesagt wäre eine derartige Reduktion der Dimension (und damit des Platzbedarfs) des Vektors immer dann unmöglich, wenn noch ein Prozess mit höherer ID (und damit höherer Indexposition) aktiv und im fraglichen Vektor vertreten ist.

Angesichts dieser Vorüberlegungen wird eine zwar kompliziertere, aber wesentlich flexiblere Lösung darin gesucht, die Zuordnung zwischen Prozess und Uhrwert auf andere Weise vorzunehmen, als über den Index der Vektorposition.

Der verwendete Ansatz ist nahe liegend. Es geht um eine bijektive Abbildung von $p_j \in P$ auf die entsprechenden skalaren Uhrwerte $v \in C_i(e_i^x)$. Bei Vektor-Uhren lautet diese Funktion folgendermaßen:

$$f_{vc} : \ P \rightarrow C_i(e_i^x), \ p_j \mapsto C_i(e_i^x)[j].$$

Für den Lösungsansatz der *dynamischen Vektor-Uhren* wird nun etwas Platz investiert und der Vektor um eine zweite Spalte zu einer Matrix erweitert.* Die zusätzliche Spalte referenziert den Prozess, welcher dem entsprechenden Eintrag in der ursprünglichen „Wertespalte“ zugeordnet ist. Die (immer noch bijektive) Abbildung lautet daher nun folgendermaßen:

$$f_{dvc} : \ P \rightarrow C_i(e_i^x)[*, 2], \ p_j \mapsto C_i(e_i^x)[k, 2], \ \text{wobei } k \text{ derart, dass } C_i(e_i^x)[k, 1] = j.$$

Auf diese Weise wird erreicht, dass die Zeitstempelgröße nicht mehr wie bei dem eingangs verworfenen Ansatz durch die (ihrerseits unbeschränkte) Anzahl der Prozesse $|P|$ beschränkt wird, sondern durch zweimal die Anzahl von Prozessen, von denen der den Vektor unterhaltende Prozess direkt oder indirekt Uhrwerte empfangen hat. Es kann davon ausgegangen werden, dass diese Schranke in den meisten realistischen Anwendungsszenarien signifikant kleiner ist als $|P|$ (obwohl sie letztendlich ebenfalls unbeschränkt ist). Darüber hinaus ist prinzipiell eine „garbage collection“ zur Entfernung nicht länger benötigter Matrix-Einträge möglich. Dieses Thema wird in Abschnitt 4.5 ausführlich behandelt.

Abgesehen von dem durch die Einführung einer zweiten Spalte modifizierten Prozess-Uhrwert-Mapping ist die Funktionsweise der dynamischen Vektor-Uhren identisch zu der von den im letzten Kapitel vorgestellten Standard-Vektor-Uhren. Die entsprechend angepassten Algorithmen zu Aktualisierung, Auswertung und Lückenerkennung werden in Abschnitt 4.4 angegeben.

* Der Grund, diese Uhren als „dynamische Vektor-Uhren“ und nicht als „Matrix-Uhren“ zu bezeichnen (was eigentlich nahe liegender gewesen wäre), ist der, dass letzterer Begriff bereits im Zusammenhang mit anderen Konzepten eingeführt worden ist, beispielsweise in [18].

4.3 Systemmodell

Für dynamische Vektor-Uhren wird genau dasselbe Systemmodell verwendet, wie für skalare und Vektor-Uhren auch. Dieses Modell ist in Abschnitt 3.1 beschrieben. Um der Dynamik hinsichtlich Prozesserzeugung und -terminierung Rechnung zu tragen, wird lediglich folgende Änderung vorgenommen:

Eine verteilte Berechnung besteht aus einer (nicht notwendigerweise endlichen) Menge von eindeutig nummerierten *potentiellen* Prozessen $\Pi = \{p_1, p_2, \ldots\}$, von denen jeder während eines gegebenen Berechnungsablaufs erzeugt werden kann aber nicht muss. Zu jedem gegebenen Zeitpunkt während eines Berechnungsablaufs ist eine endliche Teilmenge $P \subset \Pi$ von Prozessen im System *aktiv*. Ist P leer, befindet sich die Berechnung gerade nicht in Ausführung.

4.4 Algorithmen

Da ihre Funktionsweise, wie in Abschnitt 4.2 erklärt, weitgehend analog zu derjenigen von Standard-Vektor-Uhren ist, sind auch dynamische Vektor-Uhren prinzipiell in der Lage, die starke Uhrbedingung zu erfüllen. Dazu müssen allerdings noch passende, aufeinander abgestimmte Algorithmen zur Aktualisierung und zur Auswertung der Zeitstempel festgelegt werden. Diese sind naturgemäß ebenfalls ähnlich zu den Algorithmen für Standard-Vektor-Uhren, im Wesentlichen lediglich angepasst an die veränderte Struktur der Zeitstempel, und werden in den folgenden Unterabschnitten erklärt.

4.4.1 Aktualisierung

Am Anfang, das heißt beim ersten Ereignis eines Prozesses, bestehe die Uhr aus einer einzelnen Zeile mit (wenn mit 1 zu zählen begonnen wird):

$$C_i(e_i^1)[1,1] = i \text{ und } C_i(e_i^1)[1,2] = 1.$$

Damit die (starke) Uhrbedingung stets erfüllt ist, werden die dynamischen Vektor-Uhren C_i nach den folgenden zwei Regeln geführt:

- Wenn e_i^x das Empfangsereignis einer Nachricht ist, die mit e_j^y verschickt wurde, und n die Anzahl der Zeilen in $C_j(e_j^y)$ ist, dann

  $$\forall l \mid 0 < l < n :$$
  $$C_i(e_i^x)[k,2] := \max\left(C_i(e_i^{x-1})[k,2],\ C_j(e_j^y)[l,2]\right), \text{ falls } C_j(e_j^y)[l,1] = C_i(e_i^x)[k,1].$$

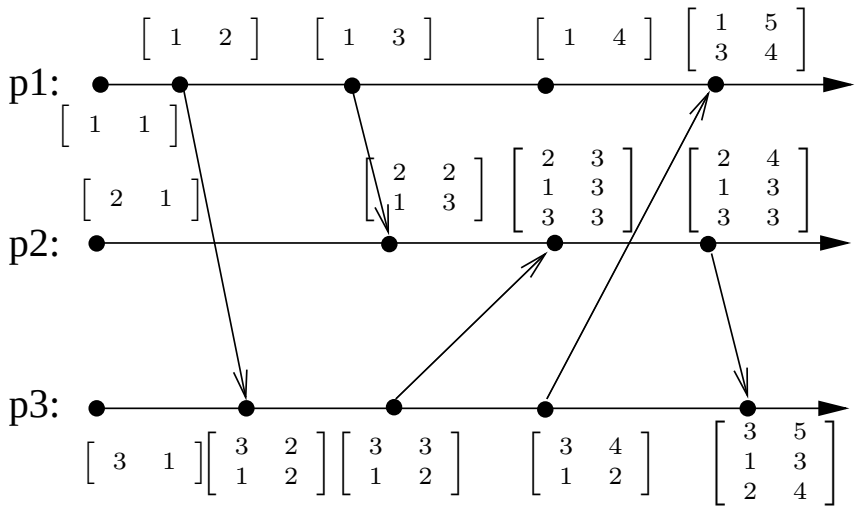

Abbildung 4.1: Dynamische Vektor-Uhren

Für jedes l, für das kein k existiert, so dass $C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1]$, wird eine neue Zeile an $C_i(e_i^x)$ angehängt mit

$$C_i(e_i^x)[m+1,1] = C_j(e_j^y)[l,1] \text{ und } C_i(e_i^x)[m+1,2] = C_j(e_j^y)[l,2],$$

wobei m die Anzahl der bereits vorhandenen Zeilen in $C_i(e_i^x)$ ist.

- Wenn e_i^x irgendein Ereignis ist (Empfangsereignisse eingeschlossen), dann inkrementiert der Prozess seinen eigenen skalaren Uhrwert:

$$C_i(e_i^x)[1,2] := C_i(e_i^{x-1})[1,2] + 1,$$

wobei, wie bereits postuliert, $C_i(e_i^x)[1,1] = i$ derjenige Eintrag ist, der den Prozess selbst repräsentiert. Im Falle eines Empfangsereignisses erfolgt dies *zusätzlich* zur ersten Regel.

Ein Beispiel zur Funktionsweise dynamischer Vektor-Uhren ist in Abbildung 4.1 gegeben.

Korollar 4.1. Die folgende wichtige Aussage ist immer wahr:

$$\nexists\, k \neq l :\ C_i(e_i^x)[k,1] = C_i(e_i^x)[l,1].$$

Dies bedeutet, dass es in jeder Uhrmatrix höchstens einen Eintrag für jeden beliebigen Prozess gibt, und folgt direkt aus den eben angegebenen Update-Regeln: Ein neuer Eintrag wird nur dann an eine Matrix angehängt, wenn dort nicht bereits ein passender Eintrag vorhanden ist. □

4.4.2 Auswertung

Die Auswertung dynamischer Vektor-Uhren erfolgt im Wesentlichen analog zu Standard-Vektor-Uhren. Dazu werden die in Abschnitt 3.5.3 für Vektor-Uhren definierten Relationen $=$ und $<$ folgendermaßen an die veränderte Struktur der Zeitstempel angepasst:

Definition 4.1. Die Relation $=$ wird für dynamische Vektor-Uhren folgendermaßen definiert. Dabei bezeichne $|C|$ die Länge eines Zeitstempels in Zeilen.

$$
\begin{aligned}
&C_i(e_i^x) = C_j(e_j^y) \text{ genau dann, wenn} \\
&\quad (|C_i(e_i^x)| = |C_j(e_j^y)|) \text{ und} \\
&\quad (\forall k \mid 0 < k \leq |C_i(e_i^x)| : \\
&\quad (\exists\, l :\ 0 < l \leq |C_j(e_j^y)| \text{ und } C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1] \text{ und } C_i(e_i^x)[k,2] = C_j(e_j^y)[l,2])).
\end{aligned}
$$

□

Es wird darauf hingewiesen, dass für eine Entscheidung über die Gleichheit zweier Zeitstempel die Reihenfolge der Einträge (Zeilen) keine Rolle spielt.

Definition 4.2. Die Relation $<$ wird für dynamische Vektor-Uhren folgendermaßen definiert. Dabei bezeichne $|C|$ die Länge eines Zeitstempels in Zeilen.

$$
\begin{aligned}
&C_i(e_i^x) < C_j(e_j^y) \text{ genau dann, wenn} \\
&\quad (C_i(e_i^x) \neq C_j(e_j^y)) \text{ und} \\
&\quad (\forall k \mid 0 < k \leq |C_i(e_i^x)| : \\
&\quad (\exists\, l :\ 0 < l \leq |C_j(e_j^y)| \text{ und } C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1] \text{ und } C_i(e_i^x)[k,2] \leq C_j(e_j^y)[l,2])).
\end{aligned}
$$

□

Auf Basis dieser angepassten Relationen kann nun, in Analogie zu Standard-Vektor-Uhren, eine Entscheidung über die kausale Abhängigkeit zweier Zeitstempel getroffen werden:

Korollar 4.2.

$$
\begin{aligned}
&e_i^x \Rightarrow e_j^y \text{ genau dann, wenn } \exists\, k, l : \\
&\quad C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1] = i \text{ und} \\
&\quad C_i(e_i^x)[k,2] \leq C_j(e_j^y)[l,2].
\end{aligned}
$$

□

Die grundsätzliche Nebenläufigkeitsbedingung bleibt (logischerweise) dieselbe wie bei allen anderen logischen Uhren, die die starke Uhrbedingung erfüllen (vergleiche Korollar 3.1 auf Seite 40):

$$e_i^x \parallel e_j^y \text{ genau dann, wenn } \begin{cases} C_j(e_j^y) \not\leq C_i(e_i^x) \text{ und} \\ C_i(e_i^x) \not\leq C_j(e_j^y). \end{cases}$$

Die Anzahl der Vergleiche, die dieser Test erfordert, entwickelt sich wie $O(|C_i(e_i^x)| \times |C_j(e_j^y)|)$. Hier jedoch kann dieser Nebenläufigkeitstest nicht auf den Vergleich zweier spezifischer Einträge reduziert werden, wie es für Standard-Vektor-Uhren mit Korollar 3.3 möglich war, weil bei dynamischen Vektor-Uhren naturgemäß keine festen Indizes zur Verfügung stehen, mittels derer man den gesuchten Eintrag direkt anspringen könnte, und daher (potentiell) alle Einträge durchsucht werden müssen.*

Wenn der Nebenläufigkeitstest angewendet wird und als Ergebnis $e_i^x \nparallel e_j^y$ liefert, dann kann jedoch nach wie vor sofort entschieden werden, je nachdem welche der beiden Bedingungen zutraf und welche nicht, ob $e_i^x \Rightarrow e_j^y$ oder $e_j^y \Rightarrow e_i^x$. Diese auf der starken Uhrbedingung basierende Eigenschaft ist der größte Vorteil von Vektor-Uhren und bleibt bei der hier vorgestellten Erweiterung für dynamische Systeme erhalten.

4.4.3 Lückenerkennung

Aufgrund der konzeptionell starken Analogie zu Standard-Vektor-Uhren, insbesondere da sie gleichermaßen die starke Uhrbedingung erfüllen, besitzen die dynamischen Vektor-Uhren auch hinsichtlich der Lückenerkennung dieselben Eigenschaften. Wie in Abschnitt 3.5.4 erläutert, handelt es sich dabei um eine eingeschränkte Eignung zur Feststellbarkeit der Existenz eines e_k^z mit $e_i^x \Rightarrow e_k^z \Rightarrow e_j^y$, nämlich nur im Falle $k = i$.

Unter Berücksichtigung der genannten Einschränkungen lautet das konkrete Entscheidungskriterium für dynamische Vektor-Uhren wie folgt.

Korollar 4.3. Für dynamische Vektor-Uhren gilt:

$\exists\, e_i^z \mid e_i^x \Rightarrow e_i^z \Rightarrow e_j^y$ genau dann, wenn
$C_i(e_i^x)[k,2] < C_j(e_j^y)[l,2]$ für beliebige k, l mit $C_i(e_i^x)[k,1] = C_j(e_j^y)[l,1]$ und $l \neq 1$.

□

Damit bestehen für dynamische Vektor-Uhren dieselben Anwendungsmöglichkeiten und -bedingungen, wie bereits in Abschnitt 3.5.4 für Standard-Vektor-Uhren beschrieben.

---

* Es soll nicht angenommen werden, dass eine zusätzliche Datenstruktur zur Verfügung steht, die die Suche etwa durch Indizierung verkürzt.

4.5 Garbage Collection

In den vorangegangenen Abschnitten wurde erläutert, wie dynamische Vektor-Uhren verwendet werden müssen, um einerseits die starke Uhrbedingung zu erfüllen, und andererseits einen mit dynamisch erzeugten Prozessen wachsenden Zeitstempelumfang zu realisieren. In der Praxis sind jedoch streng monoton wachsende Zeitstempel auch nicht wünschenswert. Je nach Laufzeit, Prozessanzahl und Kommunikationsaufkommen würden dabei die Systemkapazitäten zu stark belastet. Dies gilt sowohl für den Speicherplatz und das Netzwerk (da die Zeitstempel mit jeder Art von Kommunikation mitversendet werden müssen), als auch für die Rechenkapazität, da die Zeitstempel mittels der vorgestellten Algorithmen geführt und ausgewertet werden müssen. Es stellt sich daher zwangsläufig die Frage, ob und wie die Zeitstempel umgekehrt mit der dynamischen Beendigung von Prozessen auch wieder verkleinert werden können. Diese Frage wird in diesem Abschnitt und seinen Unterabschnitten ausführlich behandelt.

Im Gegensatz zu der am Anfang von Abschnitt 4.2 verworfenen Methode, Vektor-Uhren für dynamische Systeme zu erweitern, bietet die gewählte Vorgehensweise grundsätzlich durchaus die Möglichkeit, eine „garbage collection“ durchzuführen. Dies liegt daran, dass auch bei willkürlicher Entfernung von Matrixeinträgen (Zeilen) die Zuordnung der verbleibenden Einträge zu den jeweils repräsentierten Prozessen erhalten bleibt.

Die Grundidee der „garbage collection“ ist folgende: Wann immer ein Prozess p_i davon Kenntnis erhält, dass Prozess p_j beendet wurde, kann er, falls vorhanden, denjenigen Eintrag aus seiner aktuellen logischen Uhr löschen, welcher p_j repräsentiert. Dies ist der Eintrag mit $C_i(e_i^x)[k, 1] = j$. Im Allgemeinen wird eine derartige Benachrichtigung über die Beendigung von Prozessen von einem zu Grunde liegenden Systemmanagement propagiert werden, wie auch immer dieses geartet sei. Dies sei im Folgenden angenommen.

Auf diese Weise können signifikante Einsparungen an Speicherplatz, Netzlast und Rechenzeit erzielt werden. Damit dies ohne Einbußen an der Konsistenz der logischen Zeitführung funktionieren kann, müssen jedoch einige schwerwiegende Seiteneffekte berücksichtigt werden. Auf diese Aspekte wird in den folgenden beiden Unterabschnitten eingegangen.

4.5.1 Das Problem der Konsistenzerhaltung

Die erwähnten Seiteneffekte, die bei der „garbage collection“ unbedingt Berücksichtigung finden müssen, laufen darauf hinaus, dass sie die Konsistenz der logischen Zeitführung als solche gefährden. Das heißt, dass die unüberlegte Löschung von Einträgen aus Zeitstempeln im Allgemeinen dazu führt, dass die Zeitstempel nicht mehr vergleichbar sind: Jegliche Auswertung der Zeitstempel mittels der in diesem Kapitel dafür zur Verfügung gestellten

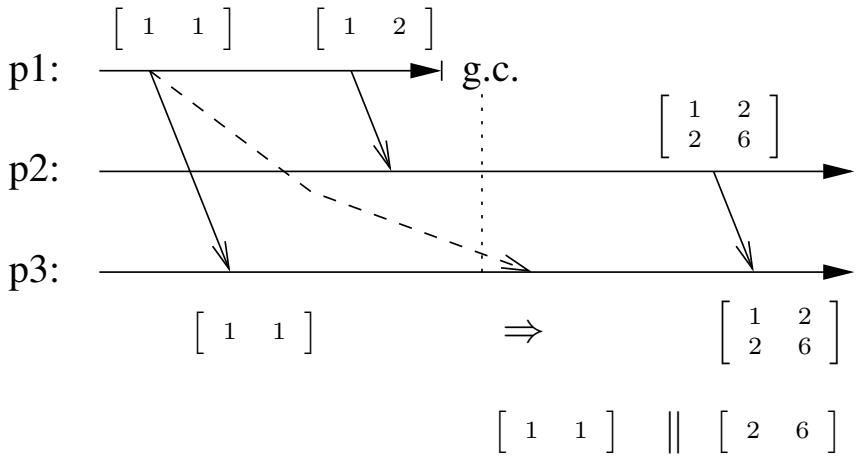

Abbildung 4.2: Fehlerhafter Zeitstempelvergleich infolge unachtsamer „garbage collection“

Algorithmen wird möglicherweise falsche Ergebnisse liefern und damit zu Trugschlüssen im Sinne einer fehlerbehafteten Sicht auf den Systemablauf führen.

Konkret bedeutet das für die „garbage collection“ zweierlei. Zum Einen muss die Löschung von Einträgen gleichermaßen für alle Uhren und Zeitstempel erfolgen, die noch in irgendeiner Form ausgewertet werden sollen. Wenn ein spezifischer Eintrag nur in einem von zwei Zeitstempeln gelöscht wurde, die verglichen werden (Nebenläufigkeits-/Abhängigkeitstest), wird der Vergleich möglicherweise ein falsches Ergebnis liefern.

Zum Zweiten dürfen Einträge grundsätzlich nicht gelöscht werden, die einen Prozess repräsentieren, welcher zwar schon beendet sein mag, von dem aber noch Nachrichten im System unterwegs sind, die möglicherweise noch für eine Uhrenauswertung benötigt werden. Ein jeder Nebenläufigkeits-/Abhängigkeitstest benötigt in beiden beteiligten Zeitstempeln alle Einträge, die einen der beiden beteiligten Prozesse repräsentieren. Wenn derartige Einträge fehlen, dann muss dies aufgrund der zu Grunde liegenden Kausalität so sein (keine Kommunikation zwischen den beiden Prozessen), keinesfalls aber aufgrund nachträglicher Löschung. Aus diesem Grund dürfen diese Einträge weder aus den Zeitstempeln auf „verwaisten“ Nachrichten gelöscht werden, noch aus Zeitstempeln oder lokalen Uhren, mit denen sie nach Ankunft der Nachrichten möglicherweise noch verglichen werden müssen. „Verwaiste“ Nachrichten können leicht durch Verzögerungen auf ihrem Weg durch das Netzwerk entstehen, über das im hier verwendeten Systemmodell keine entsprechenden Annahmen getroffen wurden (siehe Abschnitt 4.3 bzw. 3.1).

Ein Beispiel ist in Abbildung 4.2 angegeben. Die beiden Nachrichten, die bei p_3 ankommen, müssten korrekterweise als kausal geordnet betrachtet werden. Wenn wir jedoch annehmen, dass die Zustellung der ersten Nachricht verzögert erfolgt und inzwischen eine „garbage

collection“ stattgefunden hat, im Zuge derer der erste Eintrag aus der Prozessuhr von p_2 gelöscht wurde, zeigt ein noch ausstehender Vergleich der beiden Nachrichten-Zeitstempel unweigerlich Nebenläufigkeit an.

Es müssen also hinreichende Vorsichtsmaßnahmen getroffen werden, um sicherzustellen, dass bei einer „garbage collection“ keine Nachrichten eines terminierten Prozesses mehr unterwegs sind, deren Zeitstempel unter Umständen noch ausgewertet werden müssen und durch die „garbage collection“ beschädigt würden.*

Dass zu diesem Zweck ein gewisser Aufwand getrieben werden muss, liegt auf der Hand. Um sicherzustellen, dass alle Nachrichten beim Empfänger angekommen sind, muss eine systemweite Synchronisation der Prozesse erfolgen, die unter Umständen mit einem zeitweisen Stopp einiger oder gar aller Prozesse verbunden ist. Es ist also für die „garbage collection“ ein Protokoll notwendig, ähnlich beispielsweise den Protokollen, die für die Erstellung von Schnappschüssen oder Sicherungspunkten (breakpoints) zum Einsatz kommen. Ein Vorschlag für ein solches Protokoll, das speziell für die Anwendung einer „garbage collection“ bei dynamischen Vektor-Uhren erarbeitet wurde, wird im folgenden Unterabschnitt unterbreitet.

4.5.2 Protokollvorschlag

In diesem Unterabschnitt wird ein Protokoll vorgeschlagen, das eine „garbage collection“ bei dynamischen Vektor-Uhren ermöglichen soll, ohne die Konsistenz und Vergleichbarkeit der Uhren und Zeitstempel zu beeinträchtigen. Es ist anwendbar unter den Konditionen, die in [5] für die Implementierung von causal delivery mittels Vektor-Uhren angegeben werden (siehe dazu auch Abschnitt 3.5.4).

Es wird angenommen, dass ein Monitor-Prozess p_m existiert, der über alle relevanten Ereignisse im System benachrichtigt wird, d. h. über alle Ereignisse, bei denen logische Uhren aktualisiert werden. Dies ist eine notwendige Voraussetzung dafür, dass zumindest dieser eine Prozess in der Lage ist, die eingehenden Benachrichtigungen nach ihrer Kausalität zu ordnen (d. h. p_m benötigt causal delivery; siehe dazu Abschnitte 3.3 und 3.5.4 sowie [5]). Insbesondere werde p_m auch von jedem anderen Prozess p_j über dessen Terminierung informiert.

Aufgrund der causal delivery nimmt p_m die Benachrichtigung über die Terminierung von p_j erst nach allen anderen Ereignis-Benachrichtigungen desselben Prozesses entgegen. Daher

* Eine Anmerkung: In diesem speziellen Beispiel mag es vielleicht nahe liegend erscheinen, p_3 einfach beim Empfang der verspäteten Nachricht nachträglich den p_1-Eintrag verwerfen zu lassen. In anderen Fällen jedoch wäre dies keine Lösung, etwa in Beispielen mit umgekehrter Situation.

kann p_m an dieser Stelle sicher sein, alle Ereignisse von p_j bis zu dessen Terminierung bereits zu kennen. Dann werden die folgenden Schritte unternommen, um sicherzustellen, dass alle Zeitstempel zum selben logischen Zeitpunkt verkleinert werden.

Zunächst sendet p_m (oder auch ein anderes Management-Modul, welches von p_m die nötigen Informationen erhält) eine Nachricht m_1 an alle übrigen Prozesse,* die sie anweist, zwar weiter Nachrichten zu empfangen, aber das Versenden jedweder Nachrichten einzustellen, abgesehen von Nachrichtenempfangs-Benachrichtigungen an p_m. Alle Prozesse müssen den Empfang von m_1 an p_m quittieren, und p_m wartet den Empfang aller dieser Quittungen ab. Aufgrund der causal delivery, die sonst den Empfang aller Quittungen verhindert hätte, weiß p_m nun über alle Sendeereignisse Bescheid, die bis zum gegenwärtigen Zeitpunkt im System aufgetreten sind. Daher kann p_m sicherstellen, dass keine Nachrichten mehr unterwegs sind, indem er abwartet, bis die Anzahl der bei ihm eingegangenen Nachrichtenempfangs-Benachrichtigungen gleich der Anzahl der Sendeereignisse ist.

Anschließend versendet p_m eine Broadcast-Nachricht m_2, welche alle Empfänger anweist, bestimmte Einträge aus ihren Uhren zu löschen, und wartet dann wieder alle Quittungen für diese Nachricht ab.

Nun sind alle Uhren im System wieder konsistent und p_m kann eine Broadcast-Nachricht m_3 versenden, die allen Prozessen mitteilt, dass sie wieder Nachrichten versenden dürfen. Damit ist die „garbage collection“ sicher beendet.

Wenn n die Anzahl der gegenwärtig aktiven Prozesse ist, werden in diesem Protokoll $5n$ zusätzliche Nachrichten für die „garbage collection“ versendet, so dass eine lineare Komplexität vorliegt. Der einzige Prozess, für den causal delivery angenommen wurde, ist p_m.

Korrektheitsbeweis: Damit sichergestellt werden kann, dass keine Nachrichten des terminierten Prozesses p_j mehr im System unterwegs sind, muss das Protokoll zunächst dafür sorgen, dass eine zentrale Stelle, hier p_m, über alle Nachrichten Bescheid weiß, die p_j jemals versendet hat. Angenommen, p_j hat vor seiner Terminierung eine Nachricht versendet, von der p_m immer noch nichts weiß. Weil p_m über alle Ereignisse im System benachrichtigt wird (Annahme!), also auch über das Versenden der fraglichen Nachricht durch p_j, ist die einzige Erklärung, dass p_m die entsprechende Benachrichtigung noch nicht entgegengenommen hat. In diesem Fall jedoch kann p_m auch die Terminierungsbenachrichtigung von p_j noch nicht zur Kenntnis genommen haben, da diese kausal abhängig von allen Ereignissen p_js ist und bei p_m causal delivery vorliegt. Also ist die Annahme, es gäbe eine Nachricht von p_j, von der p_m nichts weiß, widersprüchlich.

* Es wird angenommen, dass die Prozesse zumindest einem Teil des Systemmanagements zur Laufzeit bekannt sind.

Nun muss jedes weitere Versenden von Nachrichten unterbunden werden. In Analogie zum oben für p_j Bewiesenen weiß p_m auch von allen Sendeereignissen aller aktiven Prozesse p_i bis zu dem Moment, an dem er von p_i die Quittung für m_i erhält.

Aufgrund dessen kann p_m die eingehenden Nachrichtenempfangs-Benachrichtigungen zählen (die laut Protokoll noch versendet werden dürfen) und mit der Anzahl versendeter Nachrichten vergleichen, welche aufgrund des Sendeverbots konstant bleiben muss. Da alle Nachrichten verlässlich und mit endlicher Verzögerung zugestellt werden (siehe Beschreibung des Systemmodells in Abschnitt 4.3 bzw. 3.1), muss die Anzahl der Nachrichtenempfangs-Benachrichtigungen letztendlich die Anzahl der Sendeereignisse erreichen.

Zu diesem Zeitpunkt kann p_m sicher sein, dass keine Nachrichten mehr unterwegs sind. Anderenfalls hätte p_m keine Nachrichtenempfangs-Benachrichtigung für die fragliche Nachricht erhalten können, und die Anzahlen versendeter und empfangener Nachrichten hätte nicht gleich sein können, es sei denn p_m hätte auch von einem Sendeereignis nichts gewusst. Dieser Fall wurde jedoch bereits widerlegt.

Zu dem Zeitpunkt, an dem die Matrixeinträge global gelöscht werden, die einen beendeten Prozess repräsentieren, sind daher keine Nachrichten mehr unterwegs. Es können auch keine Nachrichten mehr versendet werden, für die die gelöschten Einträge relevant wären, weil der Prozess, den diese Einträge repräsentieren, bereits terminiert ist. □

4.6 Effizienz

In diesem Abschnitt soll ein kurzer Überblick über den Speicher- und Laufzeitbedarf dynamischer Vektor-Uhren gegeben werden. Dabei orientiert sich der Laufzeitbedarf für die Aktualisierungs- und Auswertungs-Algorithmen naturgemäß am Platzbedarf der Datenstrukturen, auf denen sie arbeiten, also der Zeitstempel (hier zweispaltige Matrizen).

4.6.1 Platzbedarf

Der Platzbedarf für einen Vektor-Zeitstempel hängt linear von der Anzahl der Prozesse ab. Es existieren jeweils zwei skalare Einträge für jeden Prozess, so dass die Zeitstempelgröße bei n Prozessen immer genau $2n$ beträgt. Je nach der Häufigkeit, mit der eine „garbage collection“ durchgeführt wird (vergleiche Abschnitt 4.7), kann n dabei zwischen $|P|$ und $|\Pi|$ schwanken. Dies resultiert direkt aus der im Vergleich zu Vektor-Uhren dynamischen Natur der dynamischen Vektor-Uhren. Während bei Vektor-Uhren die Zeitstempelgröße über die gesamte Berechnungsdauer konstant bleibt, variiert sie bei dynamischen Vektor-Uhren, dem Design-Ziel entsprechend, über der Zeit.

4.6.2 Zeitbedarf

Für die Aktualisierung von Zeitstempeln sind, folgend aus der Fallunterscheidung in der Aktualisierungsregel, zwei Fälle zu unterscheiden. Die Aktualisierung von Empfangsereignissen benötigt Zeit in Größenordnung $O(n^2)$, da (im schlechtesten Fall) zwei ganze Zeitstempel komponentenweise durchsucht werden müssen. Je nach der Häufigkeit, mit der eine „garbage collection“ durchgeführt wird (vergleiche Abschnitt 4.7), kann n dabei zwischen $|P|$ und $|\Pi|$ schwanken (siehe auch Abschnitt 4.6.1). Bei allen anderen Ereignissen genügt das Inkrementieren eines einzigen Vektoreintrags. Dies ist immer der erste in der jeweiligen Matrix, so dass diese nicht durchsucht werden muss und ein konstanter Zeitaufwand vorliegt ($O(1)$).

Die Auswertung der Zeitstempel über den Nebenläufigkeitstest benötigt dagegen einen Zeitaufwand von $O(n^2)$, da einmal auf Gleichheit (n) und zweimal auf kleiner (n^2) geprüft werden muss.

Die Lückenerkennung benötigt einen Zeitaufwand von $O(n^2)$, da wie bei der Aktualisierung von Empfangsereignissen zwei Zeitstempel komponentenweise miteinander verglichen werden müssen.

4.7 Bewertung

In diesem Kapitel wurden *dynamische Vektor-Uhren* vorgestellt, eine im Rahmen der vorliegenden Arbeit entwickelte Erweiterung von Vektor-Uhren für Systeme mit dynamischer Prozesserzeugung und -terminierung. Es wurde gezeigt, wie die in diesem Uhrensystem verwendeten Zeitstempel, unter Beibehaltung des etablierten Konzepts von Vektor-Uhren, je nach aktuellem Prozessaufkommen wachsen und schrumpfen können.

Aufgrund der starken Analogie im Design besitzen dynamische Vektor-Uhren hinsichtlich ihrer Leistungsfähigkeit exakt dieselben Eigenschaften wie Standard-Vektor-Uhren. Sie weisen nicht wie skalare Uhren einen inhärenten kausalitätsbezogenen Informationsverlust auf; sie erfassen die kausalen Abhängigkeiten zwischen den Ereignissen einer verteilten Berechnung vollständig. Damit sind sie nicht nur geeignet zur Konstruktion einer konsistenten totalen Ereignisordnung (wie auch die skalaren Uhren), sondern auch für alle Anwendungen, die in irgendeiner Form die Analyse der Abhängigkeiten selbst erfordern. Insbesondere kann mit Hilfe dynamischer Vektor-Uhren beispielsweise die bereits in Abschnitt 3.4.6 angesprochene Frage beantwortet werden, ob zwei Ereignisse in einer konsistenten Ordnung vertauscht werden könnten, ohne die Konsistenz zu verletzen. Eine derartige Fragestellung tritt beispielsweise dann auf, wenn entschieden werden soll, ob zwei Programmteile parallel

ausgeführt werden könnten oder nicht, ob es also z. B. sinnvoll wäre, sie auf unterschiedlichen Knoten auszuführen (Platzierungsentscheidung).

Dynamische Vektor-Uhren weisen dieselbe Eigenschaft der eingeschränkten Lückenerkennung auf wie Standard-Vektor-Uhren. Im Vergleich zu letzteren stellen dynamische Vektor-Uhren jedoch nicht mehr die nachteiligen Anforderungen, die am Anfang dieses Kapitels als Motivation für die Entwicklung dynamischer Vektor-Uhren genannt wurden:

- Die Anzahl der Prozesse im System muss nicht mehr konstant sein, d. h. sie darf sich im Laufe der Berechnung ändern.
- Die Anzahl der Prozesse muss nicht mehr a priori bekannt sein.

Damit sind dynamische Vektor-Uhren für moderne verteilte und kooperierende Rechensysteme anwendbar, ohne auf Spezialfälle beschränkt zu sein wie Standard-Vektor-Uhren (denn ein System ohne dynamisches Erzeugen und Beenden von Prozessen wäre heutzutage ein Spezialfall).

Die im Laufe diese Kapitels angestellten Effizienzüberlegungen hinsichtlich des Konzepts dynamischer Vektor-Uhren und des vorgestellten „garbage collection“-Protokolls zeigen, dass die Führung logischer Zeit in dynamisch wachsenden/schrumpfenden Anwendungen mittels dynamischer Vektor-Uhren mit zusätzlichen Kosten verbunden ist (im Vergleich zu Standard-Vektor-Uhren). Der einzige Kostenpunkt, der als kritisch zu einzustufen ist, ist die Tatsache, dass die Zeitstempelverkleinerung (im schlechtesten Fall) eine Unterbrechung der gesamten verteilten Berechnung erfordert. Deshalb sollte der Tradeoff zwischen Zeitstempelgröße und Durchführungshäufigkeit der „garbage collection“ je nach Anwendung sorgfältig bedacht werden. Das „garbage collection“-Protokoll erlaubt es jedenfalls prinzipiell, Matrixeinträge für mehrere terminierte Prozesse in einer einzigen „garbage collection“ zu entfernen. Auf diese Weise ist es möglich, die „garbage collection“ zu verschieben, bis k Prozesse beendet wurden. Eine Erhöhung von k würde dann entsprechend weniger „garbage collections“ zur Folge haben; auf der anderen Seite müsste man in Kauf nehmen, dass die Zeitstempel bis zu $k - 1$ überflüssig gewordene Einträge enthalten.

Es bleibt zu beachten, dass sich die hier besprochenen Ergebnisse auf den „allgemeinen“ Fall beziehen. Es wurde eine minimale Menge von Annahmen in Bezug auf die Architektur und das Verhalten von Prozessen und Gesamtsystem getroffen. Dies geschah mit dem Ziel, dasselbe allgemein gehaltene Systemmodell verwenden zu können, das auch den skalaren Uhren und den Standard-Vektor-Uhren zu Grunde liegt. Auf diese Weise wird eine maximale Vergleichbarkeit der drei logischen Zeitsysteme gewährleistet.

Wie man durch die Beschränkung auf ein geringfügig genauer spezifiziertes Systemverhalten deutlich bessere Voraussetzungen für die logische Zeitführung schaffen kann, wird aus

dem nächsten Kapitel hervorgehen. Dort wird ein System logischer Uhren entwickelt, das in seinen Zeitstempeln „automatisch“ mit dem dynamischen Prozessaufkommen zu wachsen und zu schrumpfen in der Lage ist, ohne eine explizite „garbage collection“ zu benötigen.

Kapitel 5

Baum-Uhren

In Kapitel 3 wurden die beiden gängigen Arten logischer Uhren vorgestellt. Außer ihrer Funktionsweise und den entsprechenden Algorithmen wurde insbesondere ihre jeweilige Leistungsfähigkeit und Eignung für verschiedene Applikationen beleuchtet. Die Defizite dieser Uhren im Hinblick auf Systeme mit dynamischer Prozesserzeugung war die Motivation zur Entwicklung zweier verbesserter Uhrensysteme, von denen eines, die dynamischen Vektor-Uhren, in Kapitel 4 bereits vorgestellt wurde.

In diesem Kapitel wird nun das zweite der beiden in dieser Arbeit entwickelten Systeme erklärt, die *Baum-Uhren* oder *tree clocks*. Diese besitzen die volle Leistungsfähigkeit der Vektor-Uhren, können aber effizient in Systemen mit dynamischer Prozesserzeugung eingesetzt werden und bieten darüberhinaus noch einige weitere Vorteile. Im folgenden Abschnitt 5.1 wird die zu Grunde liegende Zielsetzung besprochen, wobei auch die wichtigsten Punkte der Leistungsmerkmale der drei bisher erläuterten Uhrensysteme noch einmal zusammengefasst werden. Danach erfolgt die Erläuterung der Baum-Uhren. Der Aufbau des Kapitels ist dabei im Wesentlichen analog zu den beiden vorangegangenen Kapiteln, wobei sich hier jedoch zwei Abschnitte mit Algorithmen beschäftigen. Grund dafür ist, dass zwei verschiedene Systemmodelle behandelt werden, wie in Abschnitt 5.3 erklärt. Zusätzlich gibt es am Schluss noch einen Abschnitt, der auf weitere Vorteile von Baum-Uhren hinweist, die über die Zielsetzung und die bloße Analogie zu den bisher besprochenen Uhrentypen hinausgehen.

5.1 Ziel

In Abschnitt 3.4 wurden die ersten logischen Uhren, Lamports skalare Uhren, beschrieben. Im Zuge dessen wurde erörtert, warum sie für die meisten Anwendungen unzureichend

sind: Die skalaren Uhren erfassen die Kausalitätsbeziehungen der Ereignisse, um die es ja eigentlich geht, nur unvollständig.

In Abschnitt 3.5 wurden die sogenannten Vektor-Uhren erläutert. Diese wurden mit dem Ziel entwickelt, die Unzulänglichkeiten der ursprünglichen skalaren Uhren zu beseitigen. Dies ist auch gelungen, jedoch zu einem hohen Preis. Dieser Preis besteht in den Voraussetzungen, die eine Anwendung für den Einsatz von Vektor-Uhren erfüllen muss:

- Die Anzahl der Prozesse im System ist konstant, d. h. sie ändert sich nicht im Laufe der Berechnung.
- Die Anzahl der Prozesse ist a priori bekannt.

Diese Voraussetzungen sind für die Mehrzahl moderner Applikationen schlichtweg inakzeptabel, da diese von der grundlegenden Fähigkeit aktueller Betriebssysteme Gebrauch zu machen gewohnt sind, nach Belieben oder Bedarf Prozesse erzeugen und beenden zu können.

Um solchen dynamischen Anwendungen Rechnung zu tragen, sind verbesserte logische Uhren erforderlich, welche die oben genannten (und in den Abschnitten 3.4.6 und 3.5.6 erläuterten) Schwachpunkte der herkömmlichen Uhren nicht aufweisen. Das bedeutet insbesondere zweierlei:

- Die Uhren sollen in der Lage sein, die kausalen Abhängigkeiten unter den Ereignissen vollständig, d. h. mindestens* im selben Umfang zu erfassen, wie Vektor-Uhren.
- Die Uhren sollen in der Lage sein, neu erzeugte Prozesse in ihren Zeitstempeln zu berücksichtigen, ohne die Vergleichbarkeit der Zeitstempel für eine konsistente Auswertung zu verlieren.

Die erste im Rahmen der vorliegenden Arbeit entstandene Neuentwicklung eines logischen Zeiterfassungssystems wurde in Kapitel 4 vorgestellt. Die dynamischen Vektor-Uhren erfüllen die oben genannten Kriterien, sind dabei jedoch aufgrund des bewusst so allgemein wie nur möglich gehaltenen Systemmodells gewissen Zugeständnissen unterworfen. Diese betreffen im Wesentlichen die Komplexität der Aktualisierungs- und Auswertungsalgorithmen, sowie insbesondere die Notwendigkeit einer expliziten und teuren „garbage collection“ zur systemweiten Entfernung nicht mehr benötigter Uhrinformationen. Für diese ist ein Protokoll vonnöten, das die Berechnung vorübergehend ganz oder teilweise anhalten muss (siehe Abschnitt 4.5.2).

* Wünschenswert wäre natürlich noch eine uneingeschränkte Lückenerkennung.

Besser wäre natürlich eine effiziente logische Uhr, die „automatisch“ mit dem dynamischen Prozessaufkommen zu wachsen und zu schrumpfen in der Lage ist, ohne eine explizite „garbage collection“ zu benötigen. Dies war eigentlich auch das ursprüngliche Ziel; es wurde lediglich im ersten Anlauf nicht ganz erreicht und lieferte stattdessen die dynamischen Vektor-Uhren. In diesem Kapitel werden nun die sogenannten *Baum-Uhren* vorgestellt, mit deren Entwicklung das genannte Ziel letztendlich erreicht wurde.

5.2 Lösungsansatz und Funktionsweise

Der Ansatz, um ein echt dynamisches Wachsen und Schrumpfen der Zeitstempel mit dem tatsächlichen Prozessaufkommen zu erreichen, liegt in der Überlegung, dazu die Struktur heranzuziehen, der die aktiven Prozesse unterliegen. Bisher wurde eine Struktur, die die Erzeugung der Prozesse bestimmt, seitens des Systemmodells vollständig außer Acht gelassen. Das bislang zu Grunde gelegte Systemmodell (siehe Abschnitte 3.1 und 4.3) postuliert einfach eine Menge von Prozessen, ohne eine Aussage darüber zu treffen, wie die Prozesse entstehen. Ausgehend von üblichen praktischen Gegebenheiten lässt sich diesbezüglich zunächst die einfache Feststellung machen, dass Prozesse üblicherweise nicht vom Himmel fallen (wie das traditionell für logische Uhren verwendete Systemmodell bei näherer Betrachtung zu suggerieren scheint), sondern jeweils von einem anderen Prozess erzeugt werden. An dieser Stelle kommt die Struktur ins Spiel, die zwischen den Prozessen untereinander besteht und sich für die logische Zeitführung ausnutzen lassen sollte. Und diese Struktur ist, wie die obige rekursive Feststellung (Prozess erzeugt Prozess) bereits nahe legt, üblicherweise ein Baum.*

Die Grundidee für die logische Zeitführung ist nun, die besagte Interprozess-Struktur auf die Zeitstempel abzubilden beziehungsweise in den Zeitstempeln widerzuspiegeln. Da diese Struktur naturgemäß dynamisch ist, sollte sich die Dynamik auf diese Weise auf die logischen Uhren übertragen lassen.

Das Konzept von *Baum-Uhren* beruht darauf, die Ereignisse eines einzelnen Prozesses in einem Knoten eines Baumes zu zählen. Wenn dieser Prozess einen weiteren abspaltet, wird das Zählen für den alten und für den neuen Prozess in zwei Knoten fortgesetzt, die als Kinder an den bisherigen Zählerknoten angehängt werden. Auf diese Weise bleibt der Zählerwert des Vaterprozesses im Vaterknoten erhalten und kann reaktiviert werden, wenn der Kindprozess terminiert und die beiden Kindknoten entfernt werden. Die Struktur der Zeitstempel passt sich der Erzeugungsstruktur der Prozesse dynamisch an.

* Das UNIX-Kommando *pstree* kann schnell und einfach zur Veranschaulichung verwendet werden.

Bevor in den Abschnitten 5.5 und 5.6 die genauen Algorithmen angegeben werden können, die zur Führung von Baum-Uhren notwendig sind, muss zunächst noch das Systemmodell angepasst werden. Anschließend werden noch einige notwendige Notationen sowie die entsprechende Terminologie eingeführt.

5.3 Systemmodell

Dieser Abschnitt beschreibt das Systemmodell, anhand dessen die folgenden Abschnitte Baum-Uhren behandeln werden. Dieses Modell basiert auf dem Experimentalsystem MoDiS [12, 13], wurde aber so weit verallgemeinert, dass es sich so wenig wie möglich von den in den Kapiteln 3 und 4 verwendeten Modell unterscheidet und damit für die meisten verteilten Systeme anwendbar bleibt. Dieser Abschnitt hätte sich auch auf die Erläuterung der Unterschiede beschränken und ansonsten auf die ursprüngliche Vorstellung des Modells in Abschnitt 3.1 verweisen können, wie es im Zusammenhang mit dynamischen Vektor-Uhren der Fall war (Abschnitt 4.3). Um einer möglichen Verwirrung durch allzu viel Verweise und eine auf mehrere Abschnitte verteilte Modellbeschreibung vorzubeugen, soll an dieser Stelle jedoch das (modifizierte) Modell in seiner Gesamtheit noch einmal ausführlich beschrieben werden.

Genaugenommen werden in diesem Abschnitt, bzw. in den folgenden beiden Unterabschnitten, zwei verschiedene Systemmodelle eingeführt, die im Folgenden als das *elementare Systemmodell* und das *erweiterte Systemmodell (mit Nachrichtenaustausch)* bezeichnet werden. Die Baum-Uhren werden in den Abschnitten 5.5 und 5.6 für jedes dieser Modelle erklärt, wobei sie jeweils unterschiedliche Eigenschaften aufweisen, und verschiedene Algorithmen zum Einsatz kommen müssen.

5.3.1 Das elementare Systemmodell

Da logische Uhren zur Erfassung dynamischer Berechnungsabläufe dienen sollen, ist die Grundlage des Systemmodells konsequenterweise der *Prozess*. Die Berechnung selbst besteht aus einer (nicht notwendigerweise endlichen) Menge von eindeutig nummerierten *potentiellen* Prozessen $\Pi = \{p_1, p_2, \ldots\}$, von denen jeder während eines gegebenen Berechnungsablaufs erzeugt werden kann aber nicht muss. Zu jedem gegebenen Zeitpunkt während eines Berechnungsablaufs ist eine endliche Teilmenge $P \subset \Pi$ von Prozessen im System *aktiv*. Ist P leer, befindet sich die Berechnung gerade nicht in Ausführung.

Jeder Prozess p_i zerfällt in eine Sequenz von *Ereignissen* $E_i = \{e_i^1, e_i^2, \ldots\}$, welche einer totalen Ordnung unterliegen, und zwar der *Prozessordnung* $\rightarrow$. Diese Ereignisse stellen die

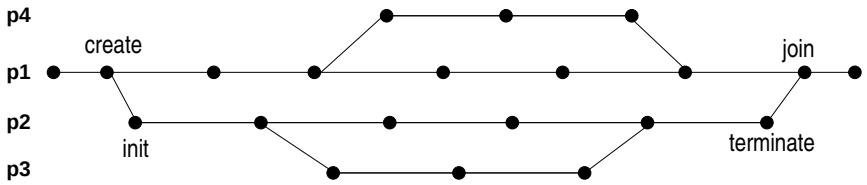

Abbildung 5.1: Ereignisverband

Berechnungsschritte dar, deren ordnungsgemäße Ausführung schließlich das Ergebnis liefert. Jedes Ereignis ist atomar und verändert den *Zustand* oder *Status* des Prozesses. Was genau ein solches Ereignis umfasst wird offen gelassen, da es von der betrachteten Abstraktionsebene abhängt. Ein Ereignis kann eine einzelne Programmanweisung oder auch eine ganze Prozedur umfassen. Wichtig ist nur, dass es auf der betrachteten Abstraktionsebene als unteilbare Einheit gesehen wird. E sei die Menge aller Ereignisse der Berechnung.

Von besonderem Interesse im Hinblick auf das globale Verhalten von Systemen mit dynamisch erzeugten Prozessen sind diejenigen Ereignisse, die die Erzeugung oder die Terminierung eines Prozesses repräsentieren. Diese werden im Folgenden als *Erzeugungs-*, *Initialisierungs-*, *Terminierungs-*, und *Vereinigungsereignisse* bezeichnet.*

Wenn ein Prozess ein Erzeugungsereignis ausführt, generiert er einen neuen (nebenläufigen) Prozess, dessen erstes Ereignis ein Initialisierungsereignis ist. Dies impliziert, dass ein Initialisierungsereignis immer *nach* dem entsprechenden Erzeugungsereignis ausgeführt wird, so dass hier eine kausale Abhängigkeit vorliegt. Analog dazu wartet ein Prozess, der ein Vereinigungsereignis ausführt, bis das entsprechende Terminierungsereignis eines zuvor erzeugten Kindprozesses ausgeführt worden ist.† Aufgrund der Abhängigkeiten zwischen diesen Ereignissen, wird die Prozessordnung zu einer partiellen globalen Ordnung $\Rightarrow$ erweitert, die im Unterabschnitt 5.4.1 formal definiert wird. Dabei handelt es sich um nichts anderes als die entsprechend angepasste happened-before-Relation.

Aus dem oben Gesagten folgt, dass jeder Prozess im System mit einem Initialisierungsereignis beginnt und mit einem Terminierungsereignis endet. Im Zuge des Initialisierungsereignisses wird der Prozess zu P hinzugefügt und beim Terminierungsereignis wieder aus P entfernt. Ein Prozess kann erst dann terminieren, wenn er für jeden seiner Kindprozesse

* Die englischen Bezeichnungen sind *create*, *init*, *term(inate)*, und *join events*. Diese werden später im Zusammenhang mit der (im Code vollständig englisch gehaltenen) Implementierung verwendet, insbesondere in den Abbildungen und Code-Listings.

† In den meisten Systemen ist ein derartiger Mechanismus bereits vorhanden; entweder weil der Vaterprozess darauf wartet, dass ihm der Kindprozess ein Ergebnis zurückliefert, oder einfach weil es in UNIX-artigen Systemen Standard ist, den Rückgabewert von Kindprozessen explizit abzufragen (*wait* oder *waitpid*-Funktion), um Zombies zu verhindern.

ein Vereinigungsereignis ausgeführt hat. Daher kann o. B. d. A. angenommen werden, dass die Berechnung mit genau einem aktiven Prozess beginnt und endet. Die Ereignisse jeder derartigen Berechnung bilden dann einen *Ereignisverband*, wie von Spies in [14] formal erklärt und in Abbildung 5.1 exemplarisch dargestellt (die Zeit vergeht von links nach rechts). Es handelt sich dabei um einen im üblichen mathematischen Sinne vollständigen Verband, dessen Abgeschlossenheit durch die obigen Annahmen postuliert wird.

5.3.2 Das erweiterte Systemmodell

In dem elementaren Systemmodell, wie es im letzten Unterabschnitt beschrieben wurde, wird der einzige Informationsfluss zwischen verschiedenen Prozessen durch die Abhängigkeiten von Erzeugungs- und Initialisierungsereignis sowie von Terminierungs- und Vereinigungsereignis etabliert. Ein derartiges Modell wäre adäquat etwa für reine RPC-Systeme oder dergleichen. In Abschnitt 5.6 sollen Baum-Uhren jedoch für eine Klasse von Systemen erklärt werden, in denen die Prozesse auch mittels explizitem Nachrichtenaustausch kommunizieren können. Dazu wird das im letzten Unterabschnitt beschriebene elementare Systemmodell zum *erweiterten Systemmodell mit Nachrichtenaustausch* erweitert, welches wieder weitgehend äquivalent zum bisher verwendeten Modell ist. Die Nachrichten seien beliebige (temporäre) Datenobjekte, die vom zu Grunde liegenden System *zuverlässig* vom Absender zum Empfänger (beides Prozesse aus P) transportiert werden. Über die zwischen dem Versand und dem Empfang einer Nachricht auftretende Verzögerung sei nichts weiter angenommen, als dass sie endlich sei.

Für viele Anwendungen ist es unabdingbar, dass die Reihenfolge der Nachrichten beim Transport erhalten bleibt, dass die Nachrichten also in der selben Reihenfolge zugestellt werden, in der sie auch abgeschickt wurden. Diese Funktionalität des zu Grunde liegenden Transportsystems wird im Allgemeinen als *FIFO-Kanäle* oder *FIFO channels* bezeichnet. Es liegt auf der Hand, dass solche FIFO-Kanäle in engem Zusammenhang mit logischen Uhren stehen. Da logische Uhren eben dazu dienen können, die (notwendige) Reihenfolge des Versands der Nachrichten zu rekonstruieren, sollen die FIFO-Kanäle auch in diesem Kapitel nicht als gegeben vorausgesetzt werden. Sie stellen vielmehr ein Ziel dar, das mit Hilfe der logischen Uhren erreicht bzw. implementiert werden kann.

Von Bedeutung für alle Betrachtungen des globalen Verhaltens von Systemen mit interagierenden/kooperierenden Prozessen sind, nach dem erweiterten Systemmodell, auch diejenigen Ereignisse, die den Versand oder den Empfang einer Nachricht repräsentieren. Diese Ereignisse werden weiterhin als *Sendeereignisse* bzw. *Empfangsereignisse* bezeichnet. Da eine Nachricht erst dann empfangen werden kann, wenn sie versendet worden ist, etablieren diese Ereignisse weitere Abhängigkeiten zwischen den Ereignissen, zusätzlich

zu den bereits im elementaren Systemmodell vorhandenen. Die entsprechend angepasste happened-before-Relation wird in Unterabschnitt 5.4.1 formal definiert.

5.4 Terminologie und Notationen

Bevor die Algorithmen besprochen werden können, die zur Aktualisierung und Auswertung von Baum-Uhren benötigt werden, ist es notwendig, die Terminologie und die Notationen einzuführen, die zu ihrer Beschreibung verwendet werden – was an dieser Stelle geschehen soll.

Der *Zeitstempel* eines Ereignisses e_i^x wird als $t(e_i^x)$ bezeichnet und hat die Struktur eines *Baumes*.*

Ein Baum $t(e_i^x)$ ist *größer* als ein anderer Baum $t(e_j^y)$ genau dann, wenn $e_j^y \Rightarrow e_i^x$ nach der in Unterabschnitt 5.4.1 definierten Relation $\Rightarrow$. Dies ist lediglich eine abkürzende Ausdrucksweise für den Umstand, dass der durch den Baum $t(e_i^x)$ bezeichnete logische Zeitpunkt dem durch $t(e_j^y)$ bezeichneten nachgeordnet ist.

Jeder *Knoten* n in einem solchen Baum besteht aus zwei skalaren Werten. Einer davon ist der *Wert* $v(n)$ des Knotens, der andere identifiziert den mit diesem Wert assoziierten Prozess und wird *Label* $l(n)$ des Knotens genannt.† Wann immer der Einfachheit halber davon die Rede ist, dass ein Knoten n inkrementiert wird, ist damit gemeint, dass $v(n)$ inkrementiert wird.

Zum Zeitpunkt eines jeden Ereignisses e_i^x existiert im Baum $t(e_i^x)$ genau ein Knoten, der das Prädikat $c(i)$ besitzt, was bedeutet, dass p_i augenblicklich in diesem Knoten seine Ereignisse zählt. Es wird angenommen, dass dieses Flag zusammen mit dem Knoten gespeichert ist.

Der Vaterknoten eines Knotens n wird mit $f(n)$ bezeichnet.

Wir müssen in der Lage sein, bestimmte Knoten verschiedener Bäume vergleichen zu können. Zu diesem Zweck wird Folgendes festgelegt. Zwei Knoten $n \in t(e_i^x)$ und $\overline{n} \in t(e_j^y)$ heißen *zueinander passend* genau dann, wenn sie sich in ihrem jeweiligen Baum auf derselben Ebene befinden und $l(n) = l(\overline{n})$. Wenn n und $\overline{n}$ zueinander passen, schreiben wir

---

* Hier weicht die Notation geringfügig von der bisherigen Bezeichnung $C_i(e_i^x)$ ab. Gemeint ist letztendlich exakt dasselbe, nämlich der dem Ereignis e_i^x zugeordnete Zeitstempel. Die neue Bezeichnung soll die stark veränderte Datenstruktur des Zeitstempels zum Ausdruck bringen, die nun eben ein Baum ist und nicht mehr eine ein- oder zweispaltige Matrix, deren Einträge mit einem Index bezeichnet werden können, wie bisher etwa mit $C_i(e_i^x)[k, 2]$.

† O. B. d. A. wird festgelegt, dass die Wurzel des Baumes sich immer auf Prozess p_1 bezieht, so dass für den Wurzelknoten nicht unbedingt ein Label benötigt wird.

$n \approx \overline{n}$. Es wird darauf hingewiesen, dass diese Relation nicht von den Werten $v(n)$ und $v(\overline{n})$ der Knoten abhängt.

Aus den in den Abschnitten 5.5 und 5.6 angegebenen Algorithmen geht folgende Feststellung hervor:

Korollar 5.1. Jeder Knoten $n \in t(e_i^x)$ hat höchstens einen passenden Knoten $\overline{n}$ in jedem anderen Baum $t(e_j^y)$. □

5.4.1 Die happened-before-Relation

In Abschnitt 3.2 wurde die happened-before-Relation definiert (Definition 3.1), um alle kausalen Abhängigkeiten zwischen den Ereignissen beschreiben zu können. Diese Relation ist die transitive Hülle der Prozessordnung und der natürlichen kausalen Abhängigkeiten zwischen Ereignissen verschiedener Prozesse. Durch die Einführung der Prozesserzeugungs- und -terminierungsabhängigkeiten im hier verwendeten Systemmodell (siehe Abschnitt 5.3) ist die Notwendigkeit entstanden, die happened-before-Relation für den Kontext dieses Kapitels neu zu definieren:

Definition 5.1. Die *happened-before-Relation* $\Rightarrow$ ist die kleinste Relation (im Sinne der Mächtigkeit), welche die folgenden Bedingungen erfüllt:

- Wenn $e_i^x \rightarrow e_i^y$, dann $e_i^x \Rightarrow e_i^y$.
- Wenn e_i^x ein Erzeugungsereignis und e_j^y das dazugehörige Initialisierungsereignis ist, dann $e_i^x \Rightarrow e_j^y$.
- Wenn e_i^x ein Terminierungsereignis und e_j^y das dazugehörige Vereinigungsereignis ist, dann $e_i^x \Rightarrow e_j^y$.
- Falls das erweiterte Systemmodell mit Nachrichtenaustausch zur Anwendung kommt (siehe Abschnitt 5.3.2):
  Wenn e_i^x ein Sendeereignis und e_j^y das Empfangsereignis der dazugehörigen Nachricht ist, dann $e_i^x \Rightarrow e_j^y$.
- Wenn $e_i^x \Rightarrow e_j^y$ und $e_j^y \Rightarrow e_k^z$, dann $e_i^x \Rightarrow e_k^z$.

□

5.5 Algorithmen für Systeme ohne Nachrichtenaustausch

In diesem Abschnitt werden die Algorithmen vorgestellt, mittels derer die Baum-Uhren derart geführt und ausgewertet werden können, dass die starke Uhrbedingung stets erfüllt ist.

Bei Baum-Uhren haben die Zeitstempel eine wesentlich andere Struktur als bei Vektor-Uhren oder dynamischen Vektor-Uhren. Dies führt unweigerlich dazu, dass auch die Algorithmen, die die Zeitstempel be- bzw. verarbeiten müssen, deutlich an Ähnlichkeit zu den entsprechenden Algorithmen für die bislang besprochenen Uhrentypen verlieren. Während die Algorithmen für dynamische Vektor-Uhren weitgehend analog angepasste Varianten der Algorithmen für Vektor-Uhren waren, liegt bei Baum-Uhren neben einem geringfügig veränderten Systemmodell eine grundsätzlich andere Datenstruktur vor, so dass auch mit grundsätzlich anderen Algorithmen gerechnet werden muss.

Allen in diesem Abschnitt vorgestellten Algorithmen liegt das elementare Systemmodell aus Abschnitt 5.3.1 zu Grunde. Sie unterscheiden sich dadurch von den Algorithmen, die für das erweiterte Systemmodell mit Nachrichtenaustausch aus Abschnitt 5.3.2 notwendig sind, und die in Abschnitt 5.6 behandelt werden.

Der Grund warum den Systemen ohne Nachrichtenaustausch hier in aller Ausführlichkeit ein ganzer Abschnitt gewidmet wird, obwohl sie eigentlich gegenüber dem erweiterten Systemmodell lediglich einen Spezialfall darstellen, ist darin zu suchen, dass sie für die Baum-Uhren eine Art „natürliche Umgebung“ bilden. Die kausalen Abhängigkeiten, die die Basis jeder logischen Zeitführung sind, werden durch den Informationsfluss etabliert, wie auch immer dieser in einem speziellen Systemmodell modelliert wird. Im elementaren Systemmodell ist ein Informationsfluss über die Grenzen eines einzelnen Ausführungsfadens hinaus ausschließlich durch die Erzeugung und Terminierung von Prozessen gegeben. Design und Konzept von Baum-Uhren zielen genau darauf ab, die Struktur der Prozesse untereinander zu reflektieren, was sich in diesem Fall genau mit dem Informationsfluss deckt, so dass sie ihn auf sehr natürliche Weise abbilden bzw. einfangen können. Dies führt unter anderem dazu, dass Baum-Uhren in Kombination mit dem elementaren Systemmodell als einzige der hier besprochenen logischen Uhren in der Lage sind, eine vollständige, uneingeschränkte Lückenerkennung zu ermöglichen.

Mit der Einführung von Nachrichtenaustausch gehen die Möglichkeiten für den Informationsfluss über die Prozessstruktur weit hinaus, so dass für die Baum-Uhren einige Schwierigkeiten auftreten und signifikante Anpassungen an den Algorithmen notwendig werden. Für die Nachrichtenaustausch-fähigen Algorithmen siehe, wie gesagt, Abschnitt 5.6.

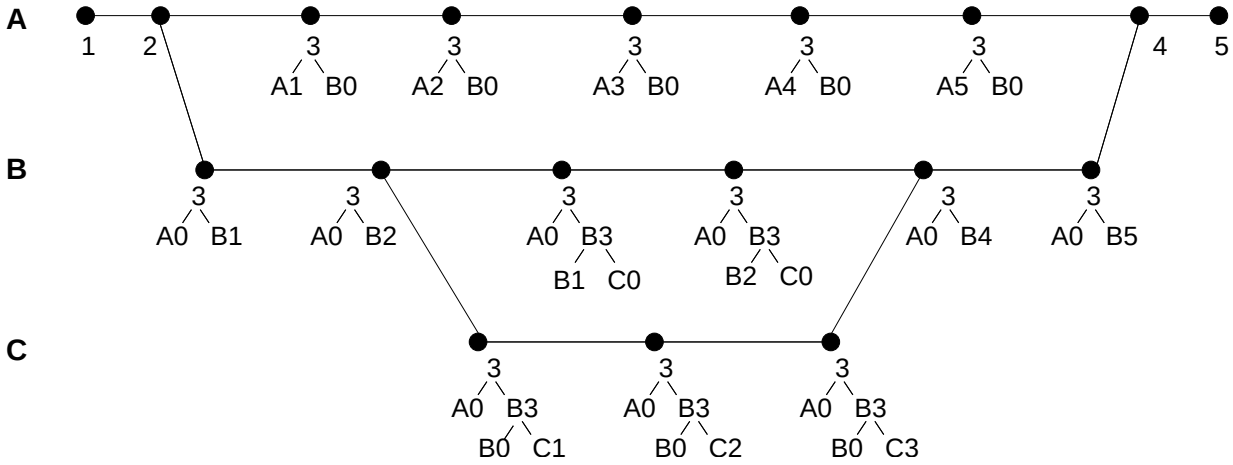

Abbildung 5.2: Baum-Uhren

5.5.1 Aktualisierung

Weil die Berechnung mit genau einem Prozess p_1 gestartet wird (siehe Abschnitt 5.3.1), wird zunächst lediglich ein einziger skalarer Zähler benötigt. Dieser Zähler wird bei jedem Ereignis um 1 erhöht, das in der streng sequentiellen Folge der Ereignisse von p_1 auftritt, und ist die Wurzel des Baumes. Am Anfang hat p_1 also einen zugeordneten Baum $t(e_i^1)$, der aus einem einzigen skalaren Knoten besteht, welcher das Prädikat $c(1)$ besitzt und mit jedem Ereignis aus der Prozessordnung inkrementiert wird. Falls kein weiterer Prozess erzeugt werden sollte, bleibt es dabei. Anderenfalls treten Erzeugungsereignisse und Vereinigungsereignisse auf, und bei diesen müssen, im Gegensatz auch zu Initialisierungs- und Terminierungsereignissen, besondere Maßnahmen ergriffen werden. Diese werden im Folgenden erläutert.

Bei jedem Erzeugungsereignis e_i^x, bei dem ein neuer Prozess p_j erzeugt wird, werden die folgenden Schritte ausgeführt:

1. $c(i) \in t(e_i^x)$ wird ganz normal inkrementiert.
2. Es wird ein Baum $t(e_i^{x+1})$ als Kopie von $t(e_i^x)$ erzeugt.
3. Zwei Kindknoten werden an $c(i) \in t(e_i^{x+1})$ angehängt. Einer erhält das Label i und das Prädikat $c(i)$,* der andere erhält das Label j und das Prädikat $c(j)$. Die Werte beider Knoten werden mit 0 initialisiert.

* Es ist wichtig, daran zu denken, dass nur ein Knoten im Baum dieses Prädikat haben kann. Der bisherige Zähler (nun Vaterknoten) muss es darum an dieser Stelle verlieren!

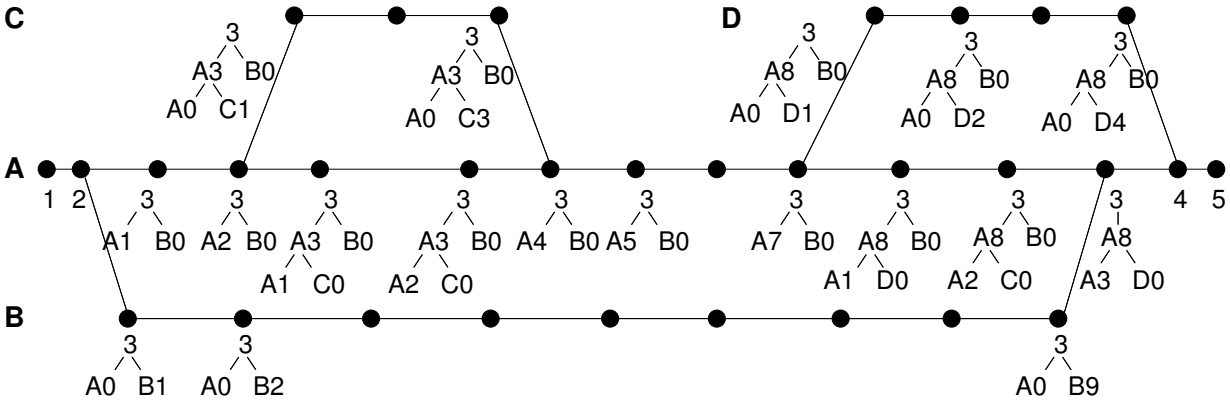

Abbildung 5.3: Baum-Uhren

4. Der komplette Baum $t(e_i^{x+1})$ wird nach $t(e_j^1)$ kopiert, so dass sowohl p_i als auch p_j jeweils einen eigenen Baum für ihre weitere Ereigniszählung besitzen.
5. p_i inkrementiert seinen neuen Zähler $c(i) \in t(e_i^x)$, p_j inkrementiert $c(j) \in t(e_j^1)$.
6. $t(e_i^{x+1})$ wird dem Ereignis e_i^{x+1} zugeordnet, nicht e_i^x!

Bei jedem Vereinigungsereignis e_i^x, bei dem ein terminierter Prozess p_j zu seinem Vater p_i zurückkehrt, wird $t(e_i^x)$ folgendermaßen aktualisiert:

1. $c(i) \in t(e_i^x)$ wird ganz normal inkrementiert.
2. Der Knoten $n \in t(e_i^x)$ mit $n \approx c(j)$ wird entfernt.
3. Falls $t(e_i^x)$ nun ein Blatt m ohne Geschwisterknoten aufweist, wird der Baum weiter beschnitten, indem m entfernt und $f(m)$ inkrementiert wird. Falls m gerade das Prädikat $c(i)$ besaß, wird dieses auf $f(m)$ übertragen.
   Schritt 3 wird wiederholt, bis kein Blatt ohne Geschwisterknoten mehr vorliegt.*

Die Abbildungen 5.2 und 5.3 illustrieren den Aktualisierungsalgorithmus. Zugunsten der besseren Lesbarkeit der Bäume werden die Prozessidentifikatoren in Form von Buchstaben anstelle von Nummern dargestellt, und in Abbildung 5.3 wurden einige weniger signifikante Zeitstempel ausgelassen. Das Vereinigungsereignis mit dem Zeitstempel „4“ in Abbildung 5.3 ist ein Beispiel für die wiederholte Anwendung der Regel für Blätter ohne Geschwister.

* Diese Regel ist der Grund dafür, dass die scheinbar schlaue und nahe liegende Optimierungsidee nicht durchführbar ist, einfach alle Knoten wegzulassen, deren Wert Null ist. Die Geschwisterknoten, ob Null oder nicht, werden für die Entscheidung benötigt, welche Knoten bei einem Vereinigungsereignis verworfen werden dürfen und welche nicht.

5.5.2 Auswertung

Die zentrale Auswertungsoperation für logische Uhren ist der Vergleich zweier Zeitstempel zum Zweck der Entscheidung, ob sie kausal geordnet oder unabhängig voneinander sind (und falls ersteres, welcher Zeitstempel der spätere ist).

Um zwei Bäume $t(e_i^x)$ und $t(e_j^y)$ zu vergleichen, wird eine einfache Breitensuche nach dem ersten Knotenpaar $n \in t(e_i^x) \approx \overline{n} \in t(e_j^y)$ durchgeführt, für das gilt: $v(n) \neq v(\overline{n})$.

Wenn $v(n) < v(\overline{n})$, dann ist $t(e_j^y)$ größer als $t(e_i^x)$, was impliziert, dass $e_i^x \Rightarrow e_j^y$.

Falls jedoch entweder $v(n) = 0$ oder $v(\overline{n}) = 0$, dann ist keiner der beiden Bäume größer als der andere, woraus folgt, dass $e_i^x \parallel e_j^y$. Von dieser Regel gibt es allerdings eine Ausnahme: Ein Knoten ohne Geschwister kann durchaus größer sein als ein passender Nullwert-Knoten.

Falls kein zu n passender Knoten $\overline{n} \in t(e_j^y)$ existiert, dann wird n nicht berücksichtigt und die Breitensuche fortgesetzt.

Zur Veranschaulichung können die Abbildungen 5.2 und 5.3 verwendet werden. Wenn ein Pfad zwischen zwei Ereignissen existiert, dann muss der Baum des weiter rechts liegenden Ereignisses nach obigem Algorithmus größer sein.

Nach dieser etwas informellen Definition des Auswertungsalgorithmus sollen nun auch noch formal die Relationen = und < für Baum-Zeitstempel definiert werden.

Definition 5.2. Die Relation = wird für Baum-Uhren folgendermaßen definiert. Dabei bezeichne $|t|$ die Anzahl der Knoten im Baum.

$$
\begin{aligned}
&t(e_i^x) = t(e_j^y) \text{ genau dann, wenn} \\
&\quad |t(e_i^x)| = |t(e_j^y)| \text{ und} \\
&\quad \forall n \in t(e_i^x)\ \exists\, \overline{n} \in t(e_j^y) \text{ mit } n \approx \overline{n} \text{ und } v(n) = v(\overline{n}).
\end{aligned}
$$

□

Definition 5.3. Die Relation < wird für Baum-Uhren folgendermaßen definiert:

$$
\begin{aligned}
&t(e_i^x) < t(e_j^y) \text{ genau dann, wenn} \\
&\quad t(e_i^x) \neq t(e_j^y) \\
&\quad \text{und für das erste durch Breitensuche gefundene Knotenpaar} \\
&\quad n \in t(e_i^x) \approx \overline{n} \in t(e_j^y) \text{ mit } v(n) \neq v(\overline{n}) \text{ gilt:} \\
&\qquad v(n) < v(\overline{n}) \text{ und} \\
&\qquad (v(n) \neq 0 \text{ oder } \overline{n} \text{ hat keine Geschwisterknoten}).
\end{aligned}
$$

□

Wie auch schon aus der oben gelieferten verbalen Beschreibung des Auswertungsalgorithmus hervorgeht, kann über die Richtung einer eventuellen kausalen Abhängigkeit zwischen den beiden gegebenen Zeitstempeln in ein und derselben Breitensuche entschieden werden wie über ihre Nebenläufigkeit. Für die Nebenläufigkeitsrelation heißt das, dass sie erstmals ohne Abhängigkeit von der Relation $<$ angegeben werden kann:

Korollar 5.2. Unter Verwendung von Baum-Uhren und des elementaren Systemmodells (siehe Abschnitt 5.3.1) gilt:

$$
\begin{aligned}
&e_i^x \parallel e_j^y \text{ genau dann, wenn} \\
&\quad t(e_i^x) \neq t(e_j^y) \\
&\quad \text{und für das erste durch Breitensuche gefundene Knotenpaar} \\
&\quad n \in t(e_i^x) \approx \overline{n} \in t(e_j^y) \text{ mit } v(n) \neq v(\overline{n}) \text{ gilt:} \\
&\qquad (v(n) = 0 \text{ und } \overline{n} \text{ hat Geschwisterknoten}) \text{ oder} \\
&\qquad (v(\overline{n}) = 0 \text{ und } n \text{ hat Geschwisterknoten}).
\end{aligned}
$$

□

Beweis: Folgt direkt aus Korollar 3.1 auf Seite 40 und der soeben angegebenen Definition 5.3. □

5.5.3 Lückenerkennung

Baum-Uhren in einem verteilten System ohne Nachrichtenaustausch (z. B. in einem reinen RPC-System) sind die ersten der hier vorgestellten Uhren, die eine vollständige, uneingeschränkte Lückenerkennung erlauben. Wenn zwei kausal voneinander abhängige Bäume gegeben sind, kann man ohne weiteres entscheiden, ob zwischen ihnen ein dritter Baum in der kausalen Kette fehlt oder nicht.

Korollar 5.3. Für Baum-Uhren gilt:

$$
\begin{aligned}
&\exists\, e_i^z \mid e_i^x \Rightarrow e_i^z \Rightarrow e_j^y \text{ genau dann, wenn} \\
&\quad \forall n \in t(e_i^x) : \ (v(n) \leq 1 \text{ und } \nexists\, \overline{n} \in t(e_j^y) \text{ mit } n \approx \overline{n}) \text{ oder } |v(n) - v(\overline{n})| \leq 1 \text{ und} \\
&\quad \forall m \in t(e_j^y) : \ (v(m) \leq 1 \text{ und } \nexists\, \overline{m} \in t(e_i^x) \text{ mit } m \approx \overline{m}) \text{ oder } |v(m) - v(\overline{m})| \leq 1.
\end{aligned}
$$

□

Einfacher ausgedrückt heißt das: Es gibt genau dann keine Lücke zwischen zwei Bäumen, wenn sich die Werte aller zueinander passenden Knoten um höchstens eins unterscheiden, wobei für fehlende Knoten ein Wert von Null angenommen wird. Diese Aussage wird direkt

durch die Aktualisierungsregeln begründet: Bei keinem Ereignis wird irgendein Knoten um mehr als 1 inkrementiert, und neu angehängte Knoten erhalten immer einen Wert von entweder 0 oder 1. Daher bedeutet eine Differenz von 2 oder mehr eine Lücke (und umgekehrt). Zur Veranschaulichung können wieder die Abbildungen 5.2 und 5.3 betrachtet werden.

Eine Schwierigkeit tritt dabei jedoch noch auf. Das Entfernen von Knoten bei Vereinigungsereignissen führt inhärent zu einem gewissen Informationsverlust. Für die bereits angegebenen Aktualisierungs- und Vergleichsalgorithmen ist dies absolut unproblematisch. Leider nicht jedoch für die Lückenerkennung. Dies ist zwar nicht überraschend, führt aber dazu, dass hier ein Sonderfall auftritt, der eine besondere Lösung erfordert.

Wenn der größere der beiden Zeitstempel, für die eine Lückenerkennung durchgeführt werden soll, zu einem Vereinigungsereignis gehört, ist die Lückenerkennung nur dadurch möglich, dass der oben erwähnte Informationsverlust ausgeglichen wird. Dies ist recht einfach: Vereinigungsereignisse müssen zusätzlich zu ihrem eigenen (reduzierten) Baum auch die Bäume ihrer zwei unmittelbaren Vorgängerereignisse an diejenige Instanz mitliefern, die die Lückenerkennung durchführen soll. Typischerweise könnte dies etwa ein Monitorprozess sein, der die fortschreitende Berechnung visualisieren soll, oder auch ein Debugger.

Die einfache Regel, um den Datenverlust bei Vereinigungsereignissen zum Zweck einer Lückenerkennung zu überbrücken, besteht darin, dass jeder Baum, der kleiner ist als der eines Vereinigungsereignisses, ein direkter Vorgänger (ohne Lücke) dann und nur dann ist, wenn er in allen Knoten und Werten identisch ist mit einem der beiden Vorgängerbäume, die das Vereinigungsereignis zusätzlich zu seinem eigenen Baum zur Verfügung stellt.

In der Praxis heißt das lediglich, dass, wenn Lückenerkennung erforderlich ist, Vereinigungsereignisse (und nur diese) drei Zeitstempel an eventuelle Logs oder Benachrichtigungen über das Auftreten des Ereignisses anhängen müssen statt einem. Die Referenzimplementierung (siehe Kapitel 6) zeigt, dass dies problemlos möglich ist.

5.6 Algorithmen für Systeme mit Nachrichtenaustausch

In Abschnitt 5.5 wurden die Algorithmen für Baum-Uhren behandelt, die in Systemumgebungen ohne Nachrichtenaustausch gelten, also dem elementaren Systemmodell aus Abschnitt 5.3.1 entsprechen. In diesem Abschnitt werden nun die entsprechenden Algorithmen für Systeme mit explizitem Nachrichtenaustausch vorgestellt, d. h. allen hier angegebenen Überlegungen und Algorithmen liegt das erweiterte Systemmodell mit Nachrichtenaustausch aus Abschnitt 5.3.2 zu Grunde.

Wie in der Einleitung zu Abschnitt 5.5 bereits erklärt, wird damit die für Baum-Uhren „natürliche“ Situation verlassen, in der der Informationsfluss durch die Prozessstruktur, und damit auch die Datenstruktur der Zeitstempel, vollständig abgedeckt wird. Daher ist es nur nahe liegend, dass dabei einige Schwierigkeiten auftreten, die signifikante Anpassungen an den Aktualisierungs- und Auswertungsalgorithmen notwendig machen. Die derart modifizierten Versionen werden in den folgenden Unterabschnitten vorgestellt.

5.6.1 Aktualisierung

Alles, was in Abschnitt 5.5.1 bezüglich Erzeugungsereignissen gesagt wurde, bleibt unverändert. Im erweiterten Systemmodell mit Nachrichtenaustausch kommen jedoch noch Sende- und Empfangsereignisse hinzu. Außerdem wird der Aktualisierungsalgorithmus für Vereinigungsereignisse geringfügig modifiziert.

Die Aktualisierungsregel für Sendeereignisse ist trivial. p_i sei der Prozess, in dem das Ereignis auftritt. Dann muss lediglich der Zählerknoten $c(i)$ inkrementiert werden, wie bei jedem anderen Ereignis auch (insbesondere auch bei Initialisierungs- und Terminierungsereignissen).

Interessanter sind Empfangsereignisse. Hierbei nämlich erhält der Prozess neue Information, was entsprechend in seiner logischen Uhr, hier dem Baum, festgehalten werden muss. Zu diesem Zweck aktualisiert ein Prozess p_i, der mit einem Empfangsereignis e_i^x eine Nachricht empfängt, welche mit einem Sendeereignis e_j^y verschickt wurde, seinen Baum folgendermaßen:

1. $c(i) \in t(e_i^x)$ wird ganz normal inkrementiert.
2. $\forall$ Knoten $n \in t(e_i^x)$: Wenn ein Knoten $\overline{n} \in t(e_j^y)$ mit $n \approx \overline{n}$ und $v(n) < v(\overline{n})$ existiert, dann $v(n) := v(\overline{n})$.
3. Zusätzlich werden alle Knoten $m \in t(e_j^y)$ ohne passendes $\overline{m} \in t(e_i^x)$ nach $t(e_i^x)$ kopiert, wenn (und nur wenn) $l(f(m)) \neq i$. Anderenfalls wird der gesamte Unterbaum mit der Wurzel m *nicht* kopiert.*

Bei jedem Vereinigungsereignis e_i^x, bei dem ein terminierter Prozess p_j zu seinem Vater p_i zurückkehrt, wird $t(e_i^x)$ folgendermaßen aktualisiert:

* Kurze Erklärung dazu: Das Übernehmen des Unterbaumes unter den beschriebenen Umständen könnte die Regeln zur Reduktion des Baumes bei Vereinigungsereignissen kompromittieren, weil ein Blatt, das eigentlich keine Geschwister mehr besitzt, dadurch wieder welche bekommen könnte.

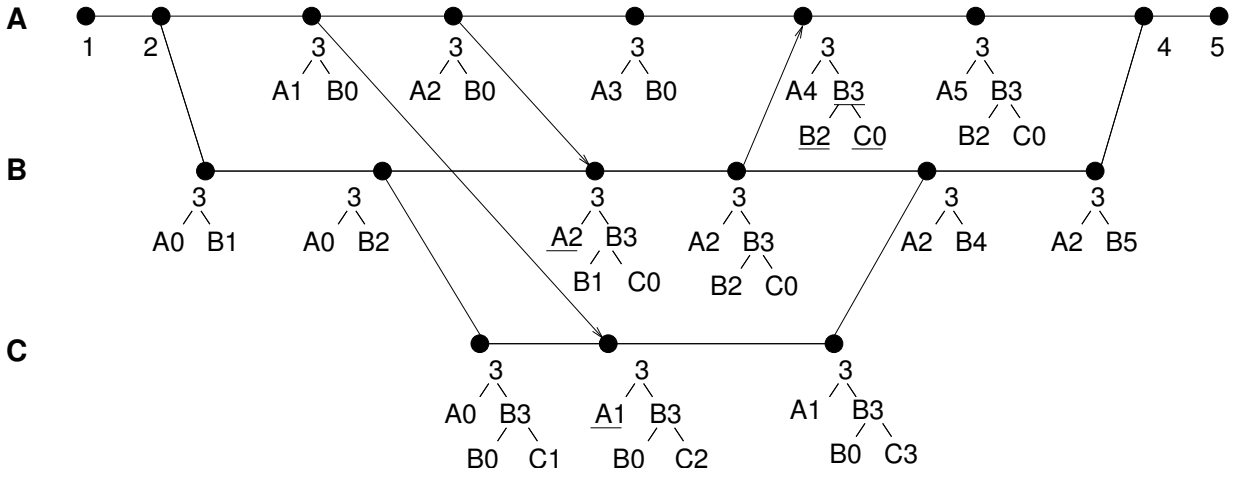

Abbildung 5.4: Baum-Uhren mit Nachrichtenaustausch

1. $c(i) \in t(e_i^x)$ wird ganz normal inkrementiert.
2. $\forall$ Knoten $n \in t(e_i^x)$: Wenn ein Knoten $\overline{n} \in t(e_j^y)$ mit $n \approx \overline{n}$ und $v(n) < v(\overline{n})$ existiert, dann $v(n) := v(\overline{n})$.
3. Der Knoten $n \in t(e_i^x)$ mit $n \approx c(j)$ wird entfernt.
4. Falls $t(e_i^x)$ nun ein Blatt m ohne Geschwisterknoten aufweist, wird der Baum weiter beschnitten, indem m entfernt und $f(m)$ inkrementiert wird. Falls m gerade das Prädikat $c(i)$ besaß, wird dieses auf $f(m)$ übertragen.
   Schritt 3 wird so oft wiederholt, bis kein Blatt ohne Geschwisterknoten mehr vorliegt.*

Es wird darauf hingewiesen, dass bei Vereinigungsereignissen, im Gegensatz zu Empfangsereignissen, keine Knoten an den Baum des Vaterprozesses angehängt werden.

Abbildung 5.4 illustriert die modifizierten Aktualisierungsalgorithmen. Knoten, die durch Nachrichtenaustausch neu gewonnene Kausalitätsinformationen beinhalten, sind dort unterstrichen dargestellt.

* Diese Regel ist der Grund dafür, dass die scheinbar schlaue und nahe liegende Optimierungsidee nicht durchführbar ist, einfach alle Knoten wegzulassen, deren Wert Null ist. Die Geschwisterknoten, ob Null oder nicht, werden für die Entscheidung benötigt, welche Knoten bei einem Vereinigungsereignis verworfen werden dürfen und welche nicht.

5.6.2 Auswertung

In diesem Unterabschnitt werden die Auswertungsrelationen für das erweiterte Systemmodell mit Nachrichtenaustausch angepasst. Die zentrale Auswertungsoperation für logische Uhren ist auch hier der Vergleich zweier Zeitstempel zum Zweck der Entscheidung, ob sie unabhängig voneinander sind oder einer kausalen Ordnung unterliegen (und wenn ja, welcher).

Die Relation $=$ ändert sich gegenüber dem elementaren Systemmodell nicht. Sie lautet immer noch wie in Definition 5.2 definiert (siehe Abschnitt 5.5.2):

$$\begin{aligned} &t(e_i^x) = t(e_j^y) \text{ genau dann, wenn} \\ &\quad |t(e_i^x)| = |t(e_j^y)| \text{ und} \\ &\quad \forall n \in t(e_i^x) : \ \exists\, \overline{n} \in t(e_j^y) \text{ mit } n \approx \overline{n} \text{ und } v(n) = v(\overline{n}). \end{aligned}$$

Wohl aber muss die Relation $<$ für das erweiterte Systemmodell mit Nachrichtenaustausch umdefiniert werden:

Definition 5.4. Die Relation $<$ wird für Baum-Uhren mit Nachrichtenaustausch folgendermaßen definiert:

$$\begin{aligned} &t(e_i^x) < t(e_j^y) \text{ genau dann, wenn} \\ &\quad t(e_i^x) \neq t(e_j^y) \text{ und} \\ &\quad \forall n \in t(e_j^y) : \ (\nexists\, \overline{n} \in t(e_i^x) \text{ mit } n \approx \overline{n}) \text{ oder } v(n) \geq v(\overline{n}). \end{aligned}$$

□

Einfacher ausgedrückt heißt das: Ein Baum ist genau dann größer als ein anderer, wenn der Wert aller seiner Knoten größer oder gleich dem Wert des jeweils passenden Knotens im anderen Baum ist. Existiert kein passender Knoten, wird für diesen ein Wert von Null angenommen.

Die Nebenläufigkeitsrelation $\parallel$ kann hier nicht mehr wie bei Baum-Uhren im elementaren Systemmodell „direkt“ angegeben werden. Wie bei allen anderen logischen Uhren auch, müssen wir sie hier auf der starken Uhrbedingung und der Relation $<$ abstützen (vergleiche Korollar 3.1 in Abschnitt 3.5.3):

$$e_i^x \parallel e_j^y \text{ genau dann, wenn } \begin{cases} t(e_j^y) \not\leq t(e_i^x) \text{ und} \\ t(e_i^x) \not\leq t(e_j^y). \end{cases}$$

Zur Veranschaulichung kann Abbildung 5.4 verwendet werden. Wenn ein Pfad zwischen zwei Ereignissen existiert, dann muss der Baum des weiter rechts liegenden Ereignisses nach obigem Algorithmus größer sein.

5.6.3 Lückenerkennung

Im erweiterten Systemmodell mit Nachrichtenaustausch befinden sich Baum-Uhren, wie in der Einleitung zu Abschnitt 5.5 erklärt, nicht mehr in ihrer „natürlichen Umgebung“ und können daher auch nicht mehr eine vollständige Lückenerkennung unterstützen, wie es im elementaren Systemmodell der Fall war. Sie leisten jedoch immer noch genau dasselbe wie Vektor-Uhren oder dynamische Vektor-Uhren: Bei zwei gegebenen Ereignissen $e_i^x \Rightarrow e_j^y$ können wir entscheiden, ob ein drittes Ereignis existiert, so dass $e_i^x \Rightarrow e_k^z \Rightarrow e_j^y$, aber nur im Spezialfall $i = k$. Für weitere Informationen zu dieser eingeschränkten Form der Lückenerkennung siehe 3.5.4 bzw. [5].

Korollar 5.4. Die Entscheidungsregel für zwei Bäume $t(e_i^x)$ und $t(e_j^y)$ mit $e_i^x \Rightarrow e_j^y$ wird durch die folgende Fallunterscheidung* angegeben:

- $i = j$ oder e_j^y ist ein Initialisierungsereignis:

$$\begin{aligned}&\nexists\, e_i^z \mid e_i^x \Rightarrow e_i^z \Rightarrow e_j^y \text{ genau dann, wenn}\\ &\quad \forall n \in t(e_j^y) \text{ auf dem Pfad von der Wurzel zu } c(j):\\ &\qquad v(n) \leq 1 \text{ oder}\\ &\qquad \exists\, \overline{n} \in t(e_i^x):\ n \approx \overline{n} \text{ und } v(n) - v(\overline{n}) \leq 1.\end{aligned}$$

- $i \neq j$ und e_j^y ist kein Initialisierungsereignis:

$$\begin{aligned}&\nexists\, e_i^z \mid e_i^x \Rightarrow e_i^z \Rightarrow e_j^y \text{ genau dann, wenn}\\ &\quad \forall n \in t(e_i^x) \text{ auf dem Pfad von der Wurzel zu } c(i):\\ &\qquad v(n) = v(\overline{n}) \text{ mit } \overline{n} \in t(e_j^y) \text{ und } n \approx \overline{n}.\end{aligned}$$

□

Korollar 5.5. Aufgrund von $e_i^x \Rightarrow e_j^y$ und der Aktualisierungsregeln für Baum-Uhren im erweiterten Systemmodell mit Nachrichtenaustausch ist es nicht möglich, dass im Fall $i \neq j$ kein passendes $\overline{n}$ in $t(e_j^y)$ existiert. □

Zur Veranschaulichung kann wieder Abbildung 5.4 verwendet werden.

---

* Die Fallunterscheidung nach i und j ist hier wegen der erwähnten Einschränkung in den Lückenerkennungsmöglichkeiten nötig. Die weitere Unterscheidung nach dem Initialisierungsereignis ist darin begründet, dass bei einem solchen nur der neu eingeführte Zähler $c(j)$ inkrementiert wird. Daher würde die Betrachtung von $c(i)$ hier nicht ausreichen.

5.7 Effizienz

In diesem Abschnitt wird eine kurze Abschätzung der Effizienz von Baum-Uhren hinsichtlich Speicher- und Zeitbedarf gegeben. Der Speicherbedarf bezieht sich hier, wie auch schon bei den anderen vorgestellten Uhrentypen, auf die Zeitstempelgröße, der Zeitbedarf auf die Komplexität der darauf operierenden Aktualisierungs- und Auswertungsalgorithmen. Es wird auch gleich ein direkter Vergleich zu den jeweiligen Abschätzungen für Vektor-Uhren als dem bisher gängigen Standard geliefert, auch wenn das redundant zum Inhalt von Abschnitt 3.5.5 ist.

Es ist wichtig, sich darüber im Klaren zu sein, dass die Zahlen für Baum-Uhren lediglich *für einen bestimmten Zeitpunkt* während des Berechnungsablaufs angegeben werden können. Dies resultiert direkt aus ihrer im Vergleich zu Vektor-Uhren stark dynamischen Natur. Während bei Vektor-Uhren beispielsweise die Zeitstempelgröße über die gesamte Berechnungsdauer konstant bleibt, variiert sie bei Baum-Uhren, dem Design-Ziel entsprechend, über der Zeit. Die Gesamtperformanz könnte daher nur auf der Basis von Informationen über das Verhalten der Berechnung abgeschätzt werden. Bezüglich dieses Verhaltens wurden jedoch keinerlei Annahmen getroffen. Insbesondere wäre hier das Schachtelungsschema der Prozesse von Bedeutung, d. h. die Entwicklung von $|P|$ über der Zeit.

5.7.1 Platzbedarf

Im Gegensatz zu Vektor-Uhren ist bei Baum-Uhren die Zeitstempelgröße nicht konstant sondern variabel über der gesamten Ausführungsdauer der verteilten Berechnung. Mit der Erzeugung von Prozessen werden Knoten zu Bäumen hinzugefügt, und mit der Beendigung von Prozessen wieder entfernt.

Wie in Abschnitt 5.3.1 bereits definiert, sei P die Menge der aktiven Prozesse. Das Verhalten von $|P|$ über der Zeit ist unbekannt. Daher können wir hier lediglich einige interessante Fälle herausgreifen.

Im Best-Case-Szenario werden alle dynamisch erzeugten Prozesse vom Initialprozess p_1 abgespalten, und zwar derart, dass stets $|P| \leq 2$ (d. h. ein neuer Kindprozess wird erst erzeugt, wenn der vorige bereits terminiert ist). In diesem Fall beträgt die Größe eines jeden Baumes zu jedem beliebigen Zeitpunkt genau entweder 1 oder 3 – und das obwohl $|\Pi|$, also die Anzahl der an der Berechnung potentiell beteiligten Prozesse, beliebig hoch sein darf!

Im schlechtesten Fall erzeugt jeder Prozess nur einen Kindprozess, der wiederum den nächsten Prozess erzeugt, d. h. es liegt gewissermaßen eine „maximale Schachtelung“ vor.

Hier ist die Baumgröße zu einem gegebenen Zeitpunkt $\in O(|P|)$, da (grob gesagt) für jeden neuen Prozess zwei Knoten hinzugefügt werden.

Es wird noch einmal auf die Dynamik hingewiesen: Alle diese Zahlen basieren auf P, also der Menge der gerade aktiven Prozesse. Dies reflektiert direkt und logischerweise das Design-Ziel von Baum-Uhren, linear-proportional mit dem Prozessaufkommen zu wachsen und zu schrumpfen. Aus diesem Grund ist der tatsächliche Platzbedarf stark abhängig vom Verhalten der Berechnung. Im Gegensatz dazu sind Vektor-Uhren in ihrer Zeitstempelgröße unabhängig vom Verhalten der Berechnung, und eine entsprechende Abschätzung muss auf der Basis von Π angegeben werden, der Menge der potentiell auftretenden Prozesse. Wenn Π endlich ist (anderenfalls wären Vektor-Uhren naturgemäß gar nicht anwendbar), ist die Größe eines Vektor-Zeitstempels immer $\in O(|\Pi|)$.

Allgemein gilt: $|P| \leq |\Pi|$. Im schlechtesten Fall, so unwahrscheinlich er für die Praxis auch sein mag, ist daher $|P| = |\Pi|$, so dass Baum-Uhren dasselbe Verhalten zeigen wie Vektor-Uhren.

Im (genauso unwahrscheinlichen) besten Fall dagegen sind die Baum-Uhren den Vektor-Uhren mit einer Zeitstempelgröße von ≤ 3 im Vergleich zu $|\Pi|$ buchstäblich beliebig weit überlegen.

In der Praxis wird der Normalfall mit $|P| \ll |\Pi|$ irgendwo zwischen diesen beiden Extremfällen liegen, was jedenfalls einen deutlichen Effizienzvorteil für die Baum-Uhren bedeutet.

Außer dass Baum-Uhren kein Wissen über $|\Pi|$ benötigen und damit im Gegensatz zu Vektor-Uhren für dynamische Systeme geeignet sind, sind sie also sogar noch effizienter.*

5.7.2 Zeitbedarf

Die Zeiteffizienz von Baum-Uhren wird in Form von Laufzeitabschätzungen für die Aktualisierungs- und Auswertungsalgorithmen angegeben. Weil diese direkt auf den Zeitstempeln arbeiten, korreliert ihre Komplexität zum Teil stark mit der Zeitstempelgröße, die im Abschnitt über den Platzbedarf abgeschätzt wurde.

---

* In einem System mit Nachrichtenaustausch kann ein nachrichtenempfangender Prozess Knoten von den Absenderprozessen ansammeln, bis sie schließlich bei der Terminierung des Empfängerprozesses wieder entfernt werden. In den meisten Fällen werden die Knoten beim Nachrichtenempfang nicht akkumuliert sondern überschrieben. In einem bestimmten schwer zu konstruierenden Szenario jedoch kann dies zu einer temporären Baumgröße von $O(|\overline{P}|)$ führen, wobei $\overline{P}$ die Menge von Prozessen ist, die vom Beginn der Berechnung bis zum gegenwärtigen Zeitpunkt aktiv waren oder sind. Trotzdem gilt: $|\overline{P}| \leq |\Pi|$.

Bei den meisten Ereignissen, die im Laufe einer Berechnung auftreten können, muss zur Aktualisierung lediglich ein einziger Knoten im Baum verändert werden. Theoretisch kann der benötigte Zeitaufwand hier also konstant sein, nämlich genau dann, wenn der Knoten direkt angesprungen werden kann, ohne erst den ganzen Baum durchsuchen zu müssen. Anderenfalls müsste der Aufwand mit $O(|P|)$ angegeben werden, entsprechend der Baumgröße im schlechtesten Fall (siehe Platzbedarf). Der eine Knoten, um den es hier geht, ist immer der aktuelle Zähler, d. h. derjenige Knoten im Baum von Prozess i, für den augenblicklich das Prädikat $c(i)$ gilt. Es ist daher ohne weiteres möglich, den Zeitaufwand konstant ($O(1)$) zu halten, indem man zusammen mit dem Baum einen Verweis auf seinen jeweiligen Zählerknoten speichert (welcher stets eindeutig bestimmt ist). Dieser Ansatz wurde in der Referenzimplementierung (siehe Kapitel 6) erfolgreich umgesetzt.

Vereinigungs- und Empfangsereignisse sind die Ausnahme von diesem konstanten Zeitbedarf. Hier bleibt eine Betrachtung des ganzen Baumes nicht aus, so dass eine Zeitkomplexität von $O(|P|)$ vorliegt.

Ebenfalls $O(|P|)$, und aus demselben Grund, gilt für die Auswertung von zwei Zeitstempeln im Sinne einer Abhängigkeits-/Nebenläufigkeitsprüfung, sowie für die Lückenerkennung.

Vektor-Uhren sind lediglich hinsichtlich der Abhängigkeits-/Nebenläufigkeitsprüfung effizienter als Baum-Uhren, da sie dafür einen konstanten Zeitaufwand ($O(1)$) benötigen. Für die Lückenerkennung und die Aktualisierung von Vereinigungs- und Empfangsereignissen hingegen benötigen Vektor-Uhren Zeit in Höhe von $O(|\Pi|)$, und sind daher weniger effizient als Baum-Uhren (vergleiche auch Abschnitt 5.7.1). In allen anderen Algorithmen besteht kein Unterschied.

5.8 Bewertung

In diesem Kapitel wurden *Baum-Uhren* vorgestellt und erläutert, logische Uhren für Systeme mit dynamischer Prozesserzeugung und -terminierung. Es wurde gezeigt, wie die in diesem Uhrensystem verwendeten Zeitstempel je nach aktuellem Prozessaufkommen in ihrem Umfang wachsen und schrumpfen können.

Im Gegensatz zu den in Kapitel 4 vorgestellten dynamischen Vektor-Uhren besteht im Konzept von Baum-Uhren bewusst wenig Analogie zu Vektor-Uhren. Es wird ein geringfügig verändertes Systemmodell zu Grunde gelegt und eine grundlegend andere Datenstruktur für die Zeitstempel verwendet. Auf diese Weise wird die in der Zielsetzung spezifizierte Dynamik erreicht, ohne Zugeständnisse in Leistungsfähigkeit oder Effizienz machen zu müssen. Baum-Uhren leisten in Systemen mit explizitem Nachrichtenaustausch exakt dasselbe wie Vektor-Uhren hinsichtlich der erfassten Kausalität, Lückenerkennung usw. In Systemen ohne expliziten Nachrichtenaustausch leisten sie sogar noch mehr, da sie hier

eine vollständige, uneingeschränkte Lückenerkennung ermöglichen. Im Folgenden wird der Leistungsumfang analog zu den Bewertungs-Abschnitten für Vektor-Uhren und dynamische Vektor-Uhren noch einmal zusammengefasst.

Baum-Uhren weisen nicht wie skalare Uhren einen inhärenten kausalitätsbezogenen Informationsverlust auf; sie erfassen die kausalen Abhängigkeiten zwischen den Ereignissen einer verteilten Berechnung vollständig. Damit sind sie nicht nur geeignet zur Konstruktion einer konsistenten totalen Ereignisordnung (wie auch die skalaren Uhren), sondern auch für alle Anwendungen, die in irgendeiner Form die Analyse der Abhängigkeiten selbst erfordern. Insbesondere kann mit Hilfe von Baum-Uhren beispielsweise die Frage beantwortet werden, ob zwei Ereignisse in einer konsistenten Ordnung vertauscht werden könnten, ohne die Konsistenz zu verletzen. Derartige Fragestellungen treten beispielsweise bei Platzierungsentscheidungen auf, d. h. wenn entschieden werden soll, ob zwei Programmteile parallel auf unterschiedlichen Knoten ausgeführt werden können.

Die Eigenschaft der eingeschränkten Lückenerkennung macht Baum-Uhren, wie Vektor-Uhren, außerdem anwendbar zur Implementierung von FIFO delivery oder causal delivery, etwa zu Monitoring- oder Debuggingzwecken – wenn auch unter einschränkenden Voraussetzungen, wie in Abschnitt 3.5.4 beschrieben. In Systemen ohne expliziten Nachrichtenaustausch, wie etwa rein RPC-basierten Systemen, ermöglichen Baum-Uhren sogar eine vollständige, uneingeschränkte Lückenerkennung, was sie unter den im Rahmen dieser Arbeit vorgestellten logischen Uhren einzigartig macht.

Im Vergleich zu Standard-Vektor-Uhren stellen Baum-Uhren nicht mehr die nachteiligen Anforderungen, die am Anfang dieses Kapitels als Motivation für die Entwicklung von Baum-Uhren genannt wurden: Dass die Anzahl der Prozesse im System durch die ganze Berechnung hindurch konstant und darüber hinaus bereits im Vorfeld bekannt sein muss.

Damit sind Baum-Uhren für moderne verteilte und kooperierende Rechensysteme anwendbar, ohne auf Spezialfälle beschränkt zu sein wie Standard-Vektor-Uhren, denn selbst in herkömmlichen Desktop- oder sogar Single-User-Systemen ist das Erzeugen und Beenden von Prozessen bereits seit geraumer Zeit Standard.

Die im Laufe diese Kapitels angestellten Effizienzüberlegungen (siehe Abschnitt 5.7) zeigen, dass Baum-Uhren nicht nur über eine gleiche bis höhere Leistungsfähigkeit verfügen wie Vektor-Uhren, sondern dass sie, von einigen wenigen Fällen abgesehen, sogar effizienter sind als Vektor-Uhren.

Zusammenfassend kann man sagen, dass hier das Ziel erreicht wurde, ein System zu Erfassung logischer Zeit zu schaffen, das auf natürliche und effiziente Weise mit der dynamischen Erzeugung und Terminierung von Prozessen zu skalieren in der Lage ist, ohne dabei an Leistungsfähigkeit einzubüßen. Die einzige zusätzliche Voraussetzung ist ein primitiver Informationsfluss bei der Prozesserzeugung bzw. -terminierung, der in den meisten Systemen

ohnehin bereits in irgendeiner Form gegeben ist (vergleiche dazu die Erklärung des Systemmodells in Abschnitt 5.3.1).

5.9 Weitere Vorteile der Baum-Uhren

Dieser Abschnitt stellt gewissermaßen einen Ausblick auf eine mögliche Weiterführung der Arbeit an den in diesem Kapitel vorgestellten Baum-Uhren dar. Es soll hier kurz auf einen qualitativen Unterschied zwischen Baum-Uhren und Vektor-Uhren oder dynamischen Vektor-Uhren hingewiesen werden, der sich für verschiedenste Anwendungen ausnutzen lassen mag.

Das Konzept von Baum-Uhren sieht vor, dass die Struktur, die durch die gegenseitige Erzeugung zwischen den an der Berechnung beteiligten Prozessen etabliert wird, von den logischen Zeitstempeln widergespiegelt wird. Wann immer ein neuer Prozess erzeugt wird, erfolgt die weitere Zählung der Ereignisse auf einer neuen Ebene, während der alte Zählerstand gemerkt und mitgeführt wird, damit er wieder reaktiviert werden kann, wenn der neue Prozess terminiert. Das bedeutet, dass Baum-Uhren über eine Art von „Gedächtnis“ verfügen, was im Vergleich zu Vektor-Uhren und dynamischen Vektor-Uhren eine völlig neue Qualität darstellt.

Eine interessante Fragestellung, die im Rahmen der vorliegenden Arbeit leider nicht mehr erschöpfend verfolgt werden konnte, ist nun: Welche Vorteile bringt diese neue Qualität mit sich, bzw. wie lässt sie sich sinnvoll nutzen?

Offensichtlich kann diese Gedächtnisfunktion beispielsweise hilfreich dabei sein, korrespondierende Erzeugungs- und Vereinigungsereignisse aufzufinden bzw. einander zuzuordnen.

Eine mögliche Anwendung dieser Vorteile wird in Kapitel 7 behandelt, wobei gewissermaßen genau der eben besagte Punkt zum Tragen kommt. Bei der erwähnten Anwendung handelt es sich um Abstraktion von Teilen der verteilten Berechnung mit möglichst vollständiger Transparenz hinsichtlich der logischen Zeit.

Es scheint nahe liegend, dass sich noch viele weitere Anwendungsbereiche unterschiedlichster Zielsetzungen und Anforderungen finden lassen müssten, für die das „Gedächtnis“ der Baum-Uhren vorteilhaft ist.

Kapitel 6

Implementierung von Baum-Uhren

In den letzten beiden Kapiteln wurden zwei verbesserte Typen logischer Uhren vorgestellt, die im Rahmen der vorliegenden Arbeit entwickelt wurden. Der zweite dieser beiden Typen, die Baum-Uhren, bildet gewissermaßen den Kern der Arbeit und wurde daher nicht nur als theoretisches Konzept ausgearbeitet, sondern auch im Rahmen des MoDiS-Projektes (siehe Abschnitt 6.3) implementiert, um Korrektheit und praktische Realisierbarkeit nachzuweisen. Diese Referenzimplementierung ist Gegenstand dieses Kapitels. Insbesondere wird hier auf alle Aspekte der Umsetzung des theoretischen Konzepts von Baum-Uhren in eine praktische Realisierung eingegangen.

Das für Baum-Uhren verwendete Systemmodell (siehe Abschnitt 5.3) wurde so allgemein wie möglich gehalten, um Baum-Uhren für eine möglichst große Klasse von verteilten Systemen anwendbar zu machen. Daher werden auch bei der Besprechung der Referenzimplementierung in diesem Kapitel die spezifischen Details des der Realisierung zu Grunde liegenden Systems MoDiS so weit wie möglich außer Acht gelassen. Sie finden nur insoweit Berücksichtigung, wie sie für die eigentlich interessanten Aspekte unmittelbar von Bedeutung sind.

Der Aufbau des Kapitels orientiert sich eng am Ablauf der Dinge: In den nächsten beiden Abschnitten werden Ziel und Lösungsansatz formuliert. Danach wird das MoDiS-System vorgestellt, das für die Implementierung als Basis dient, und im Anschluss werden abschnittsweise die einzelnen Schritte auf dem Weg von der Baum-Uhr zum Ereignisverband besprochen: die Uhrenführung, die Ereignisprotokollierung, die Rekonstruktion des Ereignisverbands sowie seine graphische Visualisierung.

Um den Lesefluss nicht dadurch zu beeinträchtigen, dass dieses Kapitel mehr Code-Listings als Fließtext enthält, wurden alle Listings in den Anhang A ausgelagert. Um dem Leser das Auffinden dennoch so weit wie möglich zu erleichtern, werden die Listings unter Angabe der jeweiligen Seitenzahl referenziert.

6.1 Ziel

Ziel der Referenzimplementierung war eine praktische Untermauerung der theoretischen Konzepte, die in Kapitel 5 präsentiert wurden. Es sollte gezeigt werden, dass Baum-Uhren einerseits tatsächlich realisierbar sind und andererseits auch die richtigen Ergebnisse liefern, also insbesondere die Ereigniskausalität korrekt erfassen. Es geht bei dieser Implementierung also nicht um eine spezifische „sinnvolle“ Anwendung der Uhren, sondern um einen sogenannten „Proof of Concept“, einen praktischen Beleg für Machbarkeit und Funktionalität der ansonsten rein theoretischen Konzepte.

Aus der oben genannten Zielsetzung folgt auch, dass der Effizienz der Implementierung eine zweitrangige Bedeutung beigemessen wird. Es wurde zwar sehr sorgfältig und überlegt programmiert, immer auch unter besonderer Berücksichtigung der Effizienz, es wäre jedoch vermessen anzunehmen, dass nicht eine effizientere Implementierung möglich sei. An verschiedenen Stellen wurde sogar bewusst ein etwas ineffizienterer Weg eingeschlagen, um den Code übersichtlicher zu halten, bzw. um die Abbildung der Inhalte von Kapitel 5 auf konkreten Code deutlicher und nachvollziehbarer zu machen.

6.2 Lösungsansatz und Funktionsweise

Um den im letzten Abschnitt genannten Zielen gerecht zu werden, wurde der folgende Ansatz verfolgt.

Das Experimentalsystem MoDiS (siehe Abschnitt 6.3) soll verwendet werden, um eine nebenläufige Berechnung auszuführen. Dass die Berechnung einen Ereignisverband bildet, ist im Fall von MoDiS schon vom Grundkonzept her sichergestellt. Die in Abschnitt 5.3.1 für Baum-Uhren geforderten Voraussetzungen sind erfüllt.

Der verwendete Ansatz ist nun, jedem (relevanten, im in Abschnitt 6.6 erklärten Sinn) Ereignis einen Baum-Zeitstempel zuzuordnen, und anschließend aus der ungeordneten Menge der Zeitstempel durch deren Auswertung den Ereignisverband zu rekonstruieren, der die Berechnung charakterisiert. Es wäre schön gewesen, die Rekonstruktion des Ereignisverbandes bereits zur Laufzeit vorzunehmen, also etwa durch einen Beobachterprozess den Fortschritt der Berechnung visualisieren zu lassen. Angesichts der knappen noch zur Verfügung stehenden Zeit wurde jedoch darauf verzichtet, und stattdessen zur Laufzeit lediglich eine Protokollierung der Ereignisse vorgenommen. Aus diesem Protokoll wird dann nach Beendigung der Berechnung der entsprechende Ereignisverband rekonstruiert und visualisiert.

Im folgenden Abschnitt wird ein kurzer Überblick über die relevanten Eigenschaften des der Realisierung zu Grunde liegenden Experimentalsystems MoDiS gegeben. In den darauf folgenden Abschnitten wird näher auf die interessanteren Einzelheiten der Referenzimplementierung von Baum-Uhren eingegangen.

6.3 MoDiS

In diesem Abschnitt wird ein kurzer Überblick über die relevanten Eigenschaften des Experimentalsystems MoDiS gegeben. Durch Integration der in Kapitel 5 anhand eines Modellsystems erarbeiteten Konzepte in MoDiS soll gezeigt werden, wie diese Konzepte auf ein reales System übertragen werden können.

MoDiS (**M**odel **o**riented **Di**stributed **S**ystems) ist ein am Lehrstuhl für Betriebssysteme und Systemarchitektur der Technischen Universität München entwickeltes Experimentalsystem. Es kann charakterisiert werden als ein sprachbasierter, top-down-getriebener Ansatz zur Entwicklung verteilter nebenläufiger Systeme. Die Anweisungen, die eine verteilte Applikation beschreiben, werden in der objektbasierten High-Level-Programmiersprache INSEL (**In**tegration and **S**eparation Supporting **E**xperimental **L**anguage) spezifiziert. MoDiS verfolgt einen sogenannten Gesamtsystemansatz: Compiler, Laufzeitumgebung, DSM-Manager* und Kommunikator sind ebenso Teil des Systems wie Betriebssystemfunktionalitäten. Der gcc-basierte† Compiler gic (GNU INSEL Compiler, siehe [22]) übersetzt die abstrakte Anwendungsspezifikation in ein ausführbares Programm, welches sowohl die Applikation als auch Management-Komponenten enthält. Alle Mechanismen dieser Transformation und jegliche während der Transformation gesammelte Informationen sind Teil des Systems. Dieses Konzept stellt die größtmögliche Verfügbarkeit von Informationen sicher und unterstützt auf diese Weise ein automatisiertes Management für die anwendungsorientierte Nutzung der verteilten Hardware-Ressourcen.

Aufgrund des besagten Gesamtsystemansatzes kann man MoDiS durchaus als eine Art „Metasystem“ betrachten, aus dem man eine Klasse konkret spezifizierter verteilter Systeme „instantiieren“ kann (ähnlich wie Objekte aus einer Klassendefinition). Die Eigenschaften eines solcherart generierten verteilten Systems sind zum Teil fest durch MoDiS vorgegeben, da beispielsweise die Laufzeitumgebung mit allen ihren Features immer dieselbe ist. Andererseits kann aber das Verhalten des Systems, die eigentliche „Anwendung“,

* DSM steht für Distributed Shared Memory, also einen verteilten gemeinsam nutzbaren Speicher. Ein solcher steht den Prozessen in MoDiS zur Verfügung.

† gcc steht für die GNU Compiler Collection.

individuell in INSEL spezifiziert werden. Alle diese Teile und Komponenten sind dann gleichermaßen Teil des konkreten, fertig instantiierten verteilten Systems.

Eine detaillierte Erläuterung der MoDiS-Konzepte findet sich in [12] und [13].

6.3.1 INSEL

INSEL ist eine imperative, objektbasierte Programmiersprache, die in ihrer Syntax an Ada angelehnt ist. Sie stellt alle Sprachkonstrukte zur Verfügung, die auch in anderen imperativen Hochsprachen üblich sind, etwa Schleifen, Bedingungen, Strukturierung in Prozeduren/Funktionen usw.

Eine Besonderheit von INSEL ist, dass die einzelnen nebenläufigen Prozesse als Datentypen definiert werden. Diese werden in MoDiS *Akteure* genannt.

Eine detaillierte Beschreibung von INSEL findet sich in [23].

6.3.2 Laufzeitumgebung

Die MoDiS-Laufzeitumgebung (libise) legt diejenigen Eigenschaften eines MoDiS-Systems fest, die für alle MoDiS-Systeme gleich sind. Hier werden beispielsweise die systemweiten Kommunikationsmöglichkeiten realisiert, ein gemeinsamer verteilter Speicher (DSM), die Erzeugung und Terminierung neuer Prozesse (in MoDiS Akteure genannt), die Speicherverwaltung usw.

Eine für die Referenzimplementierung von Baum-Uhren relevante Eigenschaft von MoDiS ist, dass kein Nachrichtenaustausch zwischen den Akteuren vorgesehen ist. Die Akteure können jedoch durchaus über einen RPC-Mechanismus Funktionen aufrufen, die von anderen Akteuren bereitgestellt werden. Solche Aufrufe werden K-Order genannt und sind in Abschnitt 6.3.4 näher beschrieben. K-Order werden in der Implementierung von Baum-Uhren zur Simulation von Nachrichtenaustausch verwendet.

Die MoDiS-Laufzeitumgebung ist in C implementiert. Weil für die Realisierung der Baum-Uhren lediglich die Laufzeitumgebung von MoDiS erweitert wurde (zur Begründung siehe Abschnitt 6.5), sind auch die Baum-Uhren vollständig in dieser Programmiersprache implementiert. Daher sind auch alle in diesem Kapitel referenzierten Code-Listings in C.

In allen Funktionen, die als Teil der MoDiS-Laufzeitumgebung realisiert wurden, werden statt den üblichen Funktionen `malloc` und `free` jeweils `ISE_malloc` bzw. `ISE_free` verwendet. Diese können vom Leser gerade wie `malloc` oder `free` betrachtet werden. Die Verwendung der speziellen ISE-Funktionen ist lediglich darin begründet, dass das MoDiS-Laufzeitsystem seine eigene Speicherverwaltung mitbringt. Insbesondere führt es

auch mit jedem Akteur eine sogenannte Manager-Struktur mit, in der eine ganze Reihe von Metadaten verwaltet werden, die den jeweiligen Akteur betreffen, unter anderem auch seine Speicherbereiche. Daher wird bei jeder Speicheranforderung (`ISE_malloc`) und -freigabe (`ISE_free`) auch ein Zeiger auf die zu verwendende Manager-Struktur als (im Vergleich zu `malloc` und `free`) zusätzlicher Parameter übergeben.

6.3.3 Akteure

Die Prozesse werden in MoDiS *Akteure* genannt und in INSEL ähnlich wie Funktionen als Datentypen definiert. Ein tatsächlich im System auftretender Akteur ist dann eine Variable des jeweiligen Akteur-Datentyps.

Alle Akteur-Datentypen gehören einer von zwei Klassen an, nämlich den M- oder den K-Akteuren. Das Besondere an K-Akteuren ist, dass sie sogenannte Procedures definieren können, welche von anderen Akteuren über einen RPC-Mechanismus aufgerufen werden können. M-Akteure dagegen können nur im Ganzen ausgeführt werden.

Im INSEL-Teil eines jeden MoDiS-Systems muss ein sogenannter System-Akteur definiert werden. Ähnlich der main-Funktion in einem C-Programm beginnt mit diesem Akteur die Ausführung des gesamten Systems. Er entspricht also p_1 im Modellsystem (siehe 5.3.1).

6.3.4 K-Order

Wie bereits erwähnt gibt es in MoDiS keinen Nachrichtenaustausch im eigentlichen Sinn, sondern ein operationenorientiertes Rendezvous-Konzept. Akteure können über einen RPC-Mechanismus Funktionen aufrufen, die von anderen Akteuren angeboten werden. Dieser Vorgang wird K-Order genannt und lässt sich durchaus als synchroner Nachrichtenaustausch auffassen. Ein K-Order-Aufruf ist dabei tatsächlich mit zwei Nachrichten verbunden, welche durch das System verschickt werden: Der Funktionsaufruf stellt eine Nachricht des Aufrufers an den K-Akteur dar, die Ergebnisrückgabe eine Nachricht des K-Akteurs an den Aufrufer.

Genau in diesem Sinne werden K-Order von der in diesem Kapitel beschriebenen Implementierung von Baum-Uhren verwendet. Bei jedem in INSEL spezifizierten K-Order-Aufruf werden zwei Nachrichten verschickt, jede mit ihrem eigenen Sende- und Empfangsereignis. Der einzige Unterschied zum in Kapitel 5 verwendeten Modellsystem (siehe 5.3.2) besteht darin, dass Nachrichten in MoDiS nicht einzeln, sondern immer paarweise auftreten. Dies entspricht jedoch lediglich einem Spezialfall und ist für die logischen Uhren völlig ohne Belang.

6.4 Datenstrukturen

Listings A.1 (Seite 143) und A.2 (Seite 144) zeigen die Datentypen, auf denen alle die Baum-Uhren betreffenden Funktionen operieren. Insbesondere repräsentieren diese Datenstrukturen Baumknoten, Bäume und Zeitstempel. Bei letzteren handelt es sich um Bäume, die zu einem Bytestrom serialisiert wurden, um in einer Datei abgespeichert oder an eine Nachricht angehängt werden zu können.

Ein Baum ist eine rekursive Datenstruktur. Aus diesem Grund ist ein Bestandteil der Definition eines Knotens ein Verweis auf einen ebensolchen Knoten. Eine weitere Konsequenz dessen, dass die wichtigste Datenstruktur für Baum-Uhren rekursiv ist, ist naheliegenderweise, dass die Aktualisierungs- und Auswertungsfunktionen, die in den folgenden Abschnitten vorgestellt werden, nahezu allesamt rekursiv implementiert wurden.

Um den Zeitaufwand für die Uhrenaktualisierung konstant zu halten und nicht den jeweiligen Baum nach dem aktuellen Zähler durchsuchen zu müssen, wurde dieser redundant markiert: Zum einen durch ein Flag in der Datenstruktur eines Baumknotens, und zum anderen durch einen Zeiger auf den Zählerknoten in der Repräsentation des Baumes. Dies erhöht die Effizienz dadurch, dass auf der einen Seite beim Durchlaufen des Baumes sofort entschieden werden kann, ob der gerade betrachtete Knoten der aktuelle Zähler ist, auf der anderen Seite der Zähler jedoch auch gezielt angesprungen werden kann, ohne dass der Baum durchsucht werden muss. Dieser Punkt wurde in Abschnitt 5.7.2 bereits einmal angesprochen.

6.5 Uhrenführung

Dieser Abschnitt ist der Realisierung der Uhrenführung gewidmet, also im Wesentlichen der Umsetzung der in Abschnitt 5.6.1 angegebenen Aktualisierungsalgorithmen.

Als erstes war die Frage zu entscheiden, wo und wie die entsprechenden Funktionen implementiert werden sollten. Der Einsprung in eine jede dieser Funktionen kann im Wesentlichen auf zwei verschiedene Weisen erfolgen: Entweder durch die statische Generierung entsprechenden Codes im INSEL-Compiler gic oder durch Aufrufe an den entsprechenden Stellen innerhalb der MoDiS-Laufzeitumgebung (libise). Die Compiler-Modifikation ist hier eindeutig der aufwändigere Weg. Nichtsdestotrotz wäre sie nötig gewesen, wenn auch ein Einsprung in die Laufzeitumgebung erst noch hätte generiert werden müssen. Nun liegen die Dinge jedoch so, dass die Ereignisse, die eine Uhrenaktualisierung erfordern, ohnehin bereits Einsprünge in die Laufzeitumgebung auslösen (Prozesserzeugung, K-Order-Aufrufe, etc.). Daher wurde hier der bequemste Weg gewählt und einfach an denjenigen Stellen in

der Laufzeitumgebung, die bei den relevanten Ereignissen ohnehin angesprungen werden, Aufrufe der jeweiligen Aktualisierungsfunktionen eingefügt.

Listing A.3 auf Seite 145 zeigt die Funktion zum Update des Baumes beim Initialisierungsereignis eines Akteurs. Diese wird von der MoDiS-Laufzeitumgebung im Zuge der Bearbeitung des Ereignisses aufgerufen.

Listing A.4 auf Seite 146 zeigt die Funktionen zum Update des Baumes bei akteurinternen Ereignissen und Erzeugungsereignissen. Die Funktion `tc_tree_update_internal` tut nichts weiter, als den aktuellen Zähler des jeweiligen Baumes zu inkrementieren. Da dies bei allen Ereignistypen mit Ausnahme von Initialisierungsereignissen Bestandteil der Aktualisierungsprozedur ist, wird diese Funktion von allen entsprechenden Aktualisierungsfunktionen verwendet. Insbesondere benötigen auch Sendeereignisse lediglich diese einfache Inkrementierung (vgl. Abschnitt 5.6.1).

Bei Erzeugungsereignissen tritt eine Besonderheit auf. Dem Ereignis, welches dem Erzeugungsereignis in der Prozessordnung folgt, ist ein anderer Baum zuzuordnen als wenn das Vorgängerereignis kein Erzeugungsereignis wäre. Umgekehrt formuliert: Das Erzeugungsereignis verlangt, dass zwei Knoten an den Baum des Akteurs angehängt werden, aber nicht beim Erzeugungsereignis selbst, sondern erst beim darauf folgenden Ereignis innerhalb desselben Akteurs, zu welcher Klasse es auch immer gehören mag. Dies macht es nötig, bei *jedem* Ereignis zu prüfen, ob das Vorgängerereignis (nach der Prozessordnung) ein Erzeugungsereignis war. Da die Funktion `tc_tree_update_internal` wie gesagt bei allen Ereignissen aufgerufen wird (außer Initialisierungsereignissen, die aber bekanntlich keinen Vorgänger in der Prozessordnung haben), macht es Sinn, die Überprüfung in eben dieser Funktion vorzunehmen. Dazu wird einfach ein Flag in der Managerstruktur (siehe Abschnitt 6.4) abgefragt, und gegebenenfalls noch `tc_tree_update_create` aufgerufen. In letzterer Funktion wird das Flag dann wieder zurückgesetzt. Beim nächsten Erzeugungsereignis wird es wieder gesetzt (direkt in der MoDiS-Laufzeitumgebung).

Listing A.5 auf Seite 147 zeigt die Funktion zum Update des Baumes bei Empfangsereignissen. Diese wird von der MoDiS-Laufzeitumgebung im Zuge der Bearbeitung des Ereignisses aufgerufen. Der Nachrichtenversand erfordert, dass der Baum des sendenden Akteurs als Zeitstempel zusammen mit der Nachricht an den Empfänger verschickt wird. Dazu muss der Baum in irgendeiner Form serialisiert werden, d. h. die zweidimensionale Baumstruktur, die im Speicher des Senders vorliegt, muss in einen eindimensionalen Bytestrom überführt werden. Der Empfänger seinerseits muss beim Empfangsereignis diesen Bytestrom wieder decodieren und den ursprünglichen Baum rekonstruieren. Diese beiden Transformationen werden von den Funktionen `tc_timestamp_from_tree` und `tc_tree_from_timestamp` durchgeführt, die in Listings A.6 (Seite 148) und A.7 (Seite 149) angegeben sind. Das Vorgehen dabei ist einfach, dass die einzelnen Knoten einer hin-

ter dem anderen in einem eindimensionalen Feld abgelegt werden, wobei die gegenseitigen Referenzen in relative Verweise innerhalb des Feldes umgewandelt werden.

Bei der Aktualisierung anlässlich eines Empfangsereignisses ist auch die Auswertung des mit der Nachricht eingehenden Zeitstempels involviert. Die dazu verwendete Funktion `tc_node_compare` ist in Listing A.11 auf Seite 153 aufgeführt.

6.6 Ereignisprotokollierung

Für die Baum-Uhren und für den Ereignisverband sind die folgenden Ereignisse relevant und müssen daher protokolliert werden:

- Initialisierungsereignisse
- Terminierungsereignisse
- Erzeugungsereignisse
- Vereinigungsereignisse
- Sendeereignisse
- Empfangsereignisse

Vergleiche dazu die Erklärungen in Abschnitt 5.3. Darüber hinaus werden einige wenige weitere Ereignisse protokolliert, um die Nebenläufigkeit der verschiedenen Ausführungsfäden im resultierenden Ereignisverband etwas deutlicher herauszustellen. Dies sind die im fertig rekonstruierten und visualisierten Ereignisverband (siehe Abbildungen 6.2, 6.3 und 6.1) mit „intern“ gekennzeichneten Ereignisse. Sie betreffen lediglich die internen Aktivitäten eines Akteurs, ohne Auswirkungen auf andere Akteure zu haben, und sind damit lediglich in die lokale Prozessordnung eingebunden.

Im Hinblick auf die Sende- und Empfangsereignisse gibt es in MoDiS einige Besonderheiten zu beachten. Diese wurden bereits in Unterabschnitt 6.3.4 näher erläutert.

Um die Protokollierung vorzunehmen, wurde die MoDiS-Laufzeitumgebung (libise) dahingehend modifiziert, dass sie an den entsprechenden Stellen den Baum des jeweiligen Akteurs in einer Protokolldatei speichert. Dazu muss der Baum allerdings zunächst serialisiert, d. h. von seiner zweidimensionalen rekursiven Struktur in einen eindimensionalen Bytestrom transformiert werden. Dies geschieht mittels der Funktion `tc_timestamp_from_tree`, die in Listing A.6 auf Seite 148 angegeben ist. Das Vorgehen dabei ist einfach, dass die

einzelnen Knoten einer hinter dem anderen in einem eindimensionalen Feld abgelegt werden. Die gegenseitigen Referenzen werden dabei in relative Verweise innerhalb des Feldes umgewandelt.

Die Funktion zum Schreiben der Protokolldatei heißt `tc_log` und ist in Listing A.8 auf Seite 150 angegeben. Eine Kleinigkeit ist hierzu noch anzumerken. Bevor MoDiS die in INSEL spezifizierte Berechnung startet, erzeugt es drei Akteure, die lediglich dem Management dienen. Der vierte erzeugte erzeugte Akteur ist dann der System-Akteur, also der erste Akteur in der eigentlichen Berechnung. Will man also genau den Ereignisverband sehen, der der INSEL-Spezifikation entspricht, müssen die Ereignisse herausgefiltert werden, die zu den ersten drei erzeugten Akteuren gehören. Über den Parameter `startactor` bietet die Funktion `tc_log` die Möglichkeit, eben dies zu steuern.

6.7 Rekonstruktion des Ereignisverbands

Wie in Abschnitt 6.2 bereits begründet, wird die Auswertung der protokollierten Zeitstempel nicht zur Laufzeit der eigentlichen Berechnung durchgeführt. Während der Laufzeit werden die Zeitstempel lediglich protokolliert. Die Auswertung kann daher außerhalb des MoDiS-Systems von einem separaten Programm durchgeführt werden. Dieses Programm wurde komplett neu geschrieben und zum MoDiS-Paket hinzugefügt. Es leistet im Wesentlichen dreierlei:

1. Es liest die protokollierten Zeitstempel aus der Protokolldatei ein, transformiert sie zurück in ihre ursprüngliche Baumstruktur und legt sie in einer übergeordneten Datenstruktur ab, die für die Durchführung von Schritt 2 geeignet ist.

2. Durch Auswertung der Zeitstempel werden die kausalen Abhängigkeiten zwischen den einzelnen Zeitstempeln rekonstruiert und ebenfalls in die Datenstruktur eingetragen. Damit ist der Ereignisverband vollständig.

3. Der rekonstruierte Ereignisverband wird in einer Graphenbeschreibungssprache beschrieben und in einer Datei abgelegt. Diese Datei kann von einem beliebigen Graphenvisualisierungsprogramm eingelesen und graphisch dargestellt werden, das die verwendete Beschreibungssprache versteht.

Zur Transformation der aus der Protokolldatei eingelesenen eindimensionalen Zeitstempel zurück in ihre zweidimensionale Baumstruktur kommt dieselbe Deserialisierungsfunktion zum Einsatz wie auch beim Nachrichtenempfang (vergleiche Abschnitt 6.5). Diese heißt `tc_tree_from_timestamp` und ist in Listing A.7 auf Seite 149 angegeben. Listing A.9 auf Seite 151 zeigt die Funktion `readlog`, die den obigen Schritt 1 durchführt.

Die Bestimmung der kausalen Abhängigkeiten erfolgt in der Funktion `make_dep` durch Auswertung der Baum-Zeitstempel in Kombination mit Lückenerkennung. Es werden jeweils die direkten (d. h. nicht-transitiven) Abhängigkeiten bestimmt und als Referenzen zwischen den einzelnen Bäumen in der dafür vorgesehenen Datenstruktur eingetragen. Siehe Listing A.10 auf Seite 152.

Weitere Funktionen, die direkt oder indirekt zur Auswertung der Baum-Uhren benötigt werden, sind in den Listings A.11 (Seite 153) bis A.14 (Seite 156) angegeben. Sie befassen sich dem Vergleich von einzelnen Knoten, dem Vergleich von ganzen Bäumen, dem Auffinden eines passenden Knotens sowie der Lückenerkennung.

Die in Listing A.15 (Seite 157) angegebene Funktion `make_graph` führt dann Schritt 3 aus. Aus dem rekonstruierten Ereignisverband generiert sie die Beschreibung eines gerichteten Graphen und speichert diese in einer Datei ab.

Zur Beschreibung des Graphen wird die Sprache GML (Graph Modelling Language, [55, 56]) verwendet. Es gibt verschiedene fertige Programme, die in der Lage sind, einen in GML spezifizierten Graphen darzustellen.

6.8 Visualisierung des Ereignisverbands

Wie im letzten Abschnitt gezeigt, wird der fertig rekonstruierte Ereignisverband als gerichteter Graph in der Sprache GML beschrieben und in einer Datei abgelegt. Diese Datei kann als Input für ein beliebiges Visualisierungstool verwendet werden, das GML interpretieren kann. Im Test wurde dazu das Programm yEd [57] verwendet. Dieses ist in der Lage, die graphische Darstellung auf verschiedene Arten automatisch zu formatieren, was sich als sehr hilfreich erwiesen hat. Die Abbildungen 6.2, 6.3 und 6.1 zeigen jeweils einen mittels yEd visualisierten Ereignisverband, der auf die im Laufe dieses Kapitels beschriebene Art und Weise aus einer tatsächlichen MoDiS-Berechnung, d. h. letztendlich aus einer INSEL-Spezifikation, generiert wurde.

6.9 Bewertung

In diesem Kapitel wurde dreierlei gezeigt:

- dass Baum-Uhren praktisch implementiert werden können
- wie Baum-Uhren praktisch implementiert werden können
- dass Baum-Uhren tatsächlich ebenso leistungsfähig sind wie Vektor-Uhren

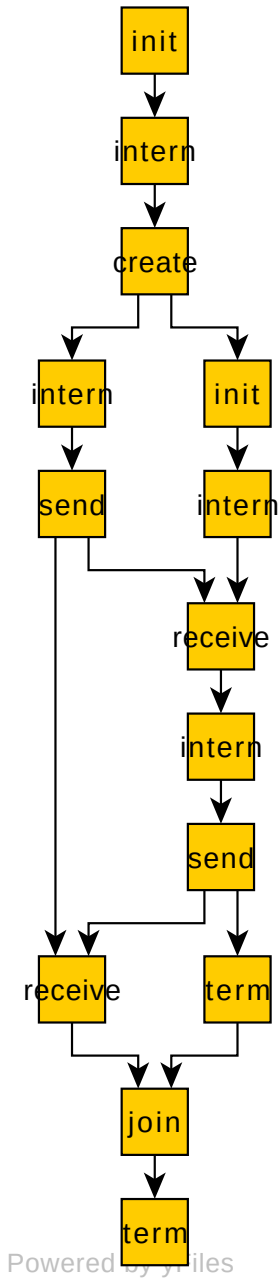

Abbildung 6.1: Visualisierung eines aus Baum-Uhren rekonstruierten Ereignisverbands

Der Ereignisverband, der sich hinter einer Berechnung verbirgt, kann mittels der hier vorgestellten Realisierung von Baum-Uhren tatsächlich korrekt und vollständig rekonstruiert werden. Damit ist die in Abschnitt 6.1 formulierte Zielsetzung erreicht, d. h. der „Proof of Concept“ erbracht worden.

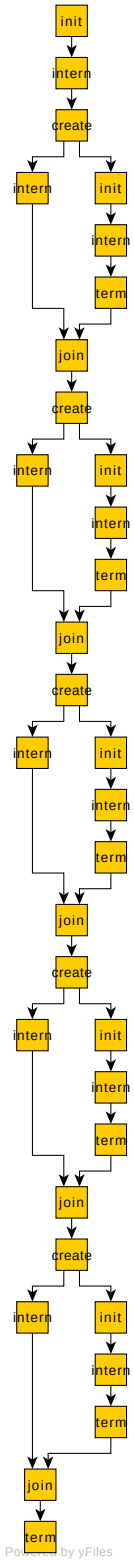

Abbildung 6.2: Visualisierung eines aus Baum-Uhren rekonstruierten Ereignisverbands

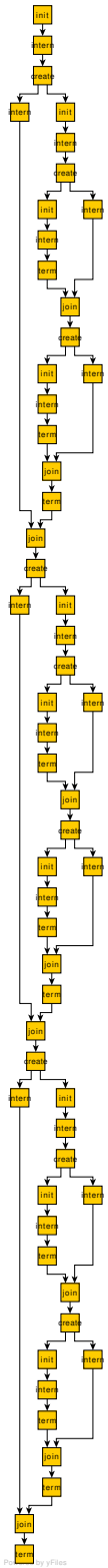

Abbildung 6.3: Visualisierung eines aus Baum-Uhren rekonstruierten Ereignisverbands

Kapitel 7

Logische Zeit und Abstraktion

In diesem Kapitel wird betrachtet, ob und wie man abstrahierte Sichten auf verteilte nebenläufige Berechnungen etablieren kann – und zwar mit möglichst vollständiger Transparenz hinsichtlich der logischen Zeit. Das heißt, die logischen Zeitstempel und ihre Beziehungen untereinander sollen durch eine eventuell vorgenommene Abstraktion möglichst nicht beeinträchtigt werden.

Mit Abstraktion sind hier Glas- und Blackbox-Sichten auf Teile einer Berechnung gemeint. Anschaulich anhand des die Berechnung charakterisierenden Ereignisverbandes erklärt bedeutet dies folgendes: Man möchte in Teile des Ereignisverbandes hinein- oder aus denselben herauszoomen können, ohne die Konsistenz der logischen Zeit zu beeinträchtigen. Konkret auf die Anwendung verteilten Debuggings bezogen mag dies heißen, dass man bestimmte uninteressante Details der Berechnung aus dem Ereignisverband ausblenden kann, und trotzdem in den verbleibenden Teilen weiterhin konsistente Zeitstempelvergleiche oder Lückenerkennung betreiben kann. Idealerweise sollte dabei sichergestellt sein, dass man sich nicht dessen bewusst sein muss, dass man gerade einen abstrahierten Ereignisverband betrachtet.

Ein derartiges Anliegen ist durch den Forschungsbereich der Usability wohlbegründet. Erwähnt sei hier nur das „Visual Information Seeking Mantra“ von Shneiderman [54]:

> Overview first, zoom and filter, then details-on-demand.

Bis zu einem gewissen Punkt ist das oben beschriebene Ziel sowohl mit Vektor-Uhren als auch mit Baum-Uhren erreichbar. Volle Abstraktionstransparenz kann jedoch nur dadurch erzielt werden, dass man sich gezielt die bereits in Abschnitt 5.9 angesprochenen qualitativ neuartigen Eigenschaften von Baum-Uhren zunutze macht.

Die in diesem Sinne erklärten Abstraktionsmöglichkeiten werden hinsichtlich zweier Arten von verteilten Systemkomponenten betrachtet, nämlich aktiven (Ereignisse/Prozesse) und

passiven (gemeinsam genutzte Datenobjekte). Auf diese Weise sollten die in diesem Kapitel vorgestellten Ideen und Ergebnisse wieder auf verteilte Systeme verschiedener Architekturen und Paradigmen anwendbar sein.

7.1 Systemmodell

In diesem Kapitel wird im Wesentlichen das erweiterte Systemmodell mit Nachrichtenaustausch verwendet, genauso wie es in Abschnitt 5.3 eingeführt wurde. Daraus folgt, dass auch die happened-before-Relation aus Definition 5.1 auf Seite 68 unverändert gültig bleibt.

7.2 Abstraktion von Ereignissen

In diesem Abschnitt erfolgt die Betrachtung von Abstraktion im Hinblick auf aktive Systemkomponenten. Das Ziel ist einfach: Nach der Anwendung logischer Uhren zur Gewinnung einer Sicht auf die inhärente Kausalität einer Berechnung sollen bestimmte Details, die für die Interpretation des Ganzen im Sinne des augenblicklichen Ziels gerade nicht wichtig sind, aus dieser Sicht ausgeblendet werden. Anders formuliert heißt das: Es liegt ein Ereignisverband mit Zeitstempeln für alle darin enthaltenen Ereignisse vor, und Teile dieses Ereignisverbandes sollen ausgeblendet werden. Man kann sich dies gut etwa im Zusammenhang mit verteiltem Debugging vorstellen, wo bestimmte Teile der Berechnung von vornherein für die Fehlersuche als uninteressant bekannt sind, andere Teile dafür aber umso genauer unter die Lupe genommen werden sollen.

Die große Frage ist: Was passiert nach dieser Abstraktion des Ereignisverbandes mit den Zeitstempeln und ihrer Konsistenz? Sind sie nach wie vor vergleichbar, d. h. ist die starke Uhrbedingung weiterhin erfüllt? Ist noch eine Lückenerkennung möglich?

Die direkte Antwort auf diese Fragen lautet: Das hängt sehr davon ab, in welcher Weise man die Abstraktion vornimmt. Die konstruktive Antwort, also die Erklärung, wie man die Abstraktion vornehmen muss, um das gewünschte Ergebnis zu erhalten, ist naturgemäß komplexer und eben genau Gegenstand dieses Kapitels.

An der oben beschriebenen Zielsetzung ist leicht zu sehen, worauf die Abstraktion letztendlich hinausläuft: Mehrere Ereignisse müssen zu einem einzigen Ereignis zusammengefasst werden. Das zusammengefasste Ereignis ist dann von einer höheren Abstraktionsebene als seine nicht zusammengefassten Nachbarn. Auf diese Weise wird eine feingranulare und lokalisierte Anpassung der Abstraktion möglich. Der Ereignisverband kann optimal an die Bedürfnisse der Anwendung angepasst werden, für die er ursprünglich durch den Einsatz

logischer Uhren konstruiert worden ist – was auch immer das für eine Anwendung sein mag. Ein Beispiel, wie gesagt, ist das dynamische Hinein- und Hinauszoomen beim Debuggen eines verteilten Programms.

Die oben gestellte Frage, wie die Abstraktion vorzunehmen ist, damit die Konsistenz der logischen Zeitstempel, und damit der Kausalitätswahrnehmung, gewahrt bleibt, reduziert sich damit auf drei konkretere Fragen:

1. Welche Ereignisse sollen zusammengefasst werden?
2. Wie sollen die Ereignisse zusammengefasst werden?
3. Was für ein Zeitstempel ist dem aus der Zusammenfassung resultierenden Ereignis zuzuordnen?

Diese Fragen werden in den Unterabschnitten 7.2.1 bis 7.2.3 nacheinander behandelt. Obwohl die Antworten auf diese Fragen in drei separaten Abschnitten gegeben werden, ist zu beachten, dass sie sich nicht unabhängig voneinander beantworten lassen. Eine spezifische Methode zur Auswahl der Ereignisse kann eine bestimmte Methode erfordern, sie zusammenzufassen oder das resultierende Ereignis mit einem Zeitstempel zu versehen (und umgekehrt). Was in diesem Abschnitt, und insbesondere eben in den drei genannten Unterabschnitten, vorgeschlagen wird, ist lediglich ein Weg, die Abstraktion vorzunehmen. Eine Gesamtlösung, maßgeschneidert für das einzige Ziel, bei der Abstraktion die Konsistenz der logischen Zeit nicht zu verletzen.

7.2.1 Auswahl der Ereignisse

Nahe liegend für eine Reihe von Anwendungen ist das Verfahren, einen oder mehrere ganze Prozesse zu einem Blackbox-Ereignis zusammenzufassen. Um wieder auf das Beispiel verteilten Debuggings zurückzukommen, könnte man daran denken, irgendwelche Hilfsprozesse aus der Sicht zu entfernen, deren Unbedenklichkeit bereits feststeht, oder etwa auch Aufrufe unbedenklicher Bibliotheksfunktionen. Wie sich später in Unterabschnitt 7.2.4 herausstellen wird, ist das im Wesentlich auch genau das, was zur Erhaltung der Lückenerkennungsfähigkeiten von Baum-Uhren notwendig ist. Nichtsdestotrotz gibt es offensichtliche Restriktionen zu beachten im Hinblick darauf, genau welche Ereignisse zusammengefasst werden dürfen und welche nicht.

Frage 1 kann definitiv nicht allgemein beantwortet werden. Ganz offensichtlich hängt die Antwort stark von den Anforderungen des Zieles ab, dem die Abstraktion dienen soll. Abstraktion impliziert immer einen (beabsichtigten) Informationsverlust. Man kann nur dann entscheiden, welche Informationen verzichtbar und welche essentiell sind, wenn man

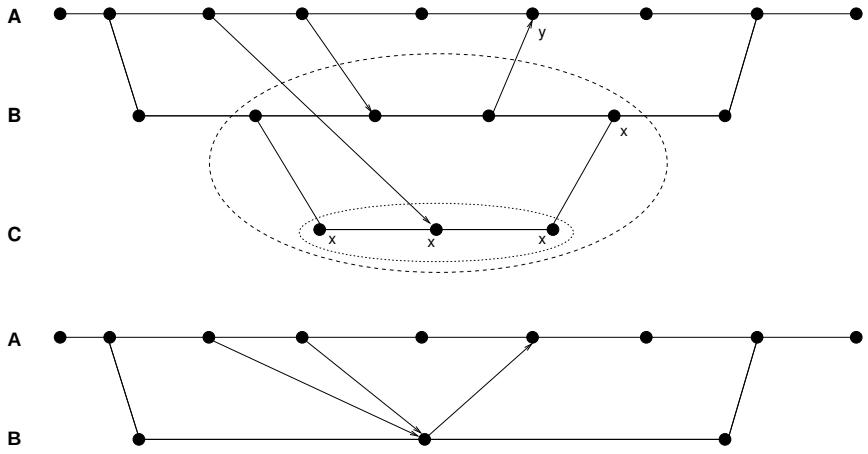

Abbildung 7.1: Ungeeignete Abstraktion

sich die Bedürfnisse der Anwendung der Abstraktion vor Augen führt. Um diesbezüglich keine willkürlichen Annahmen treffen zu müssen, beschränken wir uns hier ausschließlich auf die bereits formulierte Zielsetzung, die logische Zeit konsistent zu halten.

Ein Beispiel ist in Abbildung 7.1 gegeben. Mehrere Ereignisse aus dem ursprünglichen Ereignisverband (oberer Teil) werden zu einem einzigen abstrahierten Ereignis (unterer Teil) zusammengefasst. In diesem Fall wurden die Ereignisse derart ausgewählt, dass ein schwerer Verlust von Nebenläufigkeitsinformation entsteht: Auf der Basis des abstrahierten Ereignisverbandes ist die Tatsache nicht mehr erkennbar, dass Teile des zusammengefassten Ereignisses, nämlich die mit `x` markierten Ereignisse, eigentlich nebenläufig zu dem mit `y` markierten Ereignis sind. Wenn diese Information für die Anwendung des abstrahierten Verbandes von Bedeutung ist, müssen die Ereignisse anders gewählt werden.

Die besagten spezifischen Bedürfnisse der jeweiligen Anwendung können und sollen hier jedoch nicht näher betrachtet werden. In diesem Kapitel wird an die Abstraktion nur eine einzige Anforderung gestellt: Die Konsistenz der logischen Zeitstempel soll gewahrt bleiben. Wie bereits erwähnt erfordert dies eine Abstimmung der Methoden aufeinander, die als Antworten auf die drei eingangs formulierten Fragen angegeben werden. In diesem Sinne ist der folgende Vorschlag zur Auswahl von Ereignissen so konzipiert, dass er speziell mit den Antworten auf die Fragen 2 und 3 zusammenpasst, die in den beiden folgenden Unterabschnitten gegeben werden.

Wenn eine Anwendung im abstrahierten Ereignisverband die volle Kausalitätsinformation voraussetzt, müssen die Ereignisse strikt nach den Regeln zur Konstruktion eines *Nebenläufigkeitsverbands* zusammengefasst werden. Diese wurden bereits in [15] (der Diplomarbeit des Autors) und [16] ausgearbeitet.

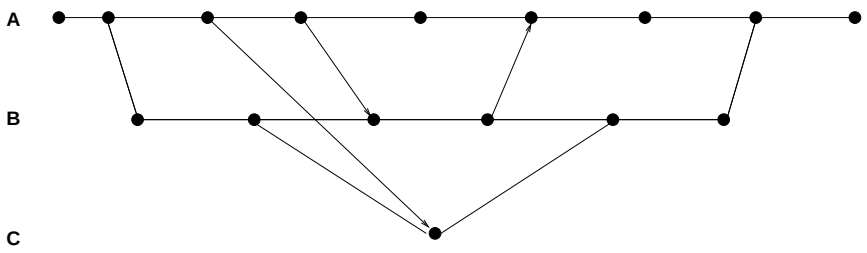

Abbildung 7.2: Gültige Abstraktion

Da wir an dieser Stelle jedoch einzig und allein an der Konsistenz der logischen Zeitstempel interessiert sind, können wir die Restriktionen etwas lockern.* Daher selektieren wir die Ereignisse, die zusammengefasst werden können, nach der folgenden Regel:

$$\forall \text{ Ereignisse } e_i^x,\ \forall \text{ Ereignisse } \epsilon,\ \text{die zu Ereignis } E \text{ zusammengefasst werden sollen,}$$
$$\text{muss gelten: wenn } E \Rightarrow e_i^x,\ \text{dann } \epsilon \Rightarrow e_i^x.$$

Wie entschieden wird, ob $E \Rightarrow e_i^x$, wird im Rahmen der Antwort auf Frage 2 in Abschnitt 7.2.2 geklärt.

Während die in Abbildung 7.1 gezeigte Abstraktion auch im Hinblick auf diese Regel nicht gültig ist, zeigt Abbildung 7.2 eine gültige Möglichkeit, von demselben Ereignisverband zu abstrahieren. Es wird nochmals darauf hingewiesen, dass auch hier noch ein Verlust von Kausalitätsinformation auftritt: Vor der Abstraktion war das erste Ereignis von Prozess C nebenläufig zum dritten Ereignis von Prozess A, nach der Abstraktion ist dies nicht mehr erkennbar. Dieser Verlust ist genau darauf zurückzuführen, dass hier eben nicht die strengen Regeln zur Konstruktion eines Nebenläufigkeitsverbandes angewendet wurden. Die oben postulierte Regel ist nur dafür gedacht, in Kombination mit den nachfolgend gelieferten Methoden zum Zusammenfassen der Ereignisse und zum Zuordnen eines Zeitstempels, die logische Zeit im Verband konsistent zu halten.

7.2.2 Zusammenfassung der Ereignisse

Im letzten Unterabschnitt wurde besprochen, wie die Ereignisse auszuwählen sind, die im Zuge der Abstraktion aus dem Ereignisverband entfernt werden sollen. Damit ist die

* Da die erwähnten Regeln zur Konstruktion eines Nebenläufigkeitsverbandes deutlich strenger sind, wären sie selbstverständlich auch für das hier gesetzte Ziel hinreichend. Sie würden uns jedoch erstens unnötig einschränken und zweitens nicht weiterbringen. Schließlich geht es hier darum, auszuloten, welche Maßnahmen zur Erhaltung der logischen Zeit *notwendig* sind.

Abstraktion jedoch erst zur Hälfte charakterisiert. Wenn Ereignisse aus dem Verband entfernt werden, dann müssen zwangsläufig auch Kanten entfernt werden, egal ob sie aus der Prozessordnung, dem Nachrichtenaustausch oder der Prozesserzeugung/-terminierung entstanden sind.

Damit die Zeitstempel im abstrahierten Ereignisverband konsistent bleiben, dürfen naheliegenderweise nur solche Kanten entfernt werden, deren benachbarte Ereignisse *beide* zu einem abstrahierten Ereignis zusammengefasst werden. Alle Kanten, die von außen in das abstrahierte Ereignis hinein- oder aus diesem herausführen, müssen im Zuge der Abstraktion unberührt bleiben. Dies kann durchaus dazu führen, dass das zusammengefasste Ereignis viele Ein-/Ausgangskanten hat. Wenn dies für die spezifische Anwendung der Abstraktion nicht akzeptabel ist, müssen wieder die zusammenzufassenden Ereignisse im Zuge von Frage 1 anders gewählt werden. Eine dieser Kanten zu entfernen, würde jedoch unweigerlich zu einer Verletzung der starken Uhrbedingung durch die verbleibenden Zeitstempel führen.

Nichtsdestoweniger gibt es von dieser Regel zwei Ausnahmen:

- Wenn durch die Zusammenfassung zwei Kanten zwischen dem abstrahierten Ereignis und einem einzigen seiner Nachbarn entstehen, dann sollte eine davon entfernt werden. Zwei Kanten zwischen denselben beiden Knoten sind überflüssig, weil die Kanten lediglich eine kausale Abhängigkeit bzw. einen potentiellen Informationsfluss repräsentieren, und ein(e) solche(r) kann nur entweder gegeben sein oder nicht.
- Aufgrund redundanten Informationsflusses darf eine eingehende *Nachrichtenkante* genau dann entfernt werden, wenn sie von einem Ereignis e_i^x ausgeht, für das gilt: $e_i^x \Rightarrow e_j^y$ und von e_j^y geht ebenfalls eine *Nachrichtenkante* zum abstrahierten Ereignis.

Als Illustration der hier vorgeschlagenen Methode, Ereignisse zusammenzufassen, können die Abbildungen 7.1 und 7.2 herangezogen werden.

7.2.3 Zuordnung eines neuen Zeitstempels

Von Ereignissen zu abstrahieren heißt, wie bereits erklärt, mehrere Ereignisse zu einem einzigen zusammenzufassen. Dies wiederum bedeutet, dass anschließend nur noch ein Ereignis ist, wo theoretisch mehrere sein sollten (siehe Abbildung 7.1). Da nun aber jedem Ereignis im ursprünglichen Verband sein eigener (einzigartiger) Zeitstempel zugeordnet war, impliziert dies, dass auch nur noch ein Zeitstempel vorhanden ist, wo theoretisch mehrere sein sollten. In diesem Unterabschnitt wird erklärt, wie man die Zeitstempel trotzdem konsistent zu halten vermag.

Wenn der abstrahierte Ereignisverband auf die im letzten Unterabschnitt vorgeschlagene Weise konstruiert wird, ist sichergestellt, dass die relative Ordnung der Zeitstempel gewahrt bleibt. Dies gilt sowohl für Vektor- als auch für Baum-Uhren. Welcher Zeitstempel ist nun aber dem abstrahierten Ereignis zuzuordnen?

Die Antwort auf diese Frage ist stark davon abhängig, welche Ereignisse wie zusammengefasst wurden, d. h. von den Antworten auf die Fragen 1 und 2. Wir werden also nun davon ausgehen, dass die Methoden aus den beiden vorangegangenen Abschnitten angewendet wurden. Diese sollten mit der folgenden Methode zur Zuordnung eines Zeitstempels kombiniert werden:

Dem abstrahierten Ereignis wird ein Zeitstempel zugeordnet, der im Wesentlichen die Vereinigung der Zeitstempel derjenigen Ereignisse darstellt, die zwar selbst Teil des zusammengefassten Ereignisses sind, aber *ausgehende* Kanten zu Knoten besitzen, die ihrerseits nicht Teil des zusammengefassten Ereignisses sind. Diese Knoten sollten identifiziert werden, *bevor* irgendwelche Kanten nach den im letzten Unterabschnitt erklärten Regeln entfernt werden.

Die besagte Vereinigung der Zeitstempel wird folgendermaßen gebildet:*

- Bei Vektor-Uhren ist der neue Vektor das komponentenweise Maximum der Zeitstempel, die vereinigt werden.
- Bei Baum-Uhren wird analog zunächst der Wert jedes Knotens auf das Maximum aller *passenden* Knoten aus den zu vereinigenden Bäumen gesetzt. Darüber hinaus muss sichergestellt werden, dass jeder Knoten aller ursprünglichen Bäume im neu generierten Baum vertreten ist.

Der Algorithmus für Baum-Uhren wird in Abbildung 7.3 veranschaulicht.

Der in diesem Abschnitt vorgeschlagene Satz von Algorithmen funktioniert für beide in Kapitel 5 eingeführten Arten von Baum-Uhren, nämlich sowohl in Systemen mit, als auch in solchen ohne Nachrichtenaustausch.

---

* Dies darf invertiert werden, solange der gesamte Satz der hier vorgeschlagenen Methoden gleichermaßen invertiert wird. Wir können die Ereignisse nach der Regel zusammenfassen: $\forall$ Ereignisse e_i^x, $\forall$ Ereignisse ϵ, die zu Ereignis E zusammengefasst werden sollen, muss gelten: falls $E \Rightarrow e_i^x$, dann $\epsilon \Rightarrow e_i^x$. Dann muss dem abstrahierten Ereignis die Vereinigung der Zeitstempel all derjenigen Ereignisse zugeordnet werden, die zwar selbst Teil des zusammengefassten Ereignisses sind, aber *eingehende* Kanten von Knoten besitzen, die ihrerseits nicht Teil des zusammengefassten Ereignisses sind. Die Vereinigung muss in diesem Fall über das komponentenweise *Minimum* gebildet werden. Eine *ausgehende* Nachrichtenkante darf genau dann entfernt werden, wenn sie zu einem Ereignis e_j^y führt, für das gilt: $e_i^x \Rightarrow e_j^y$ und e_j^y hat ebenfalls eine *eingehende* Nachrichtenkante vom abstrahierten Ereignis.

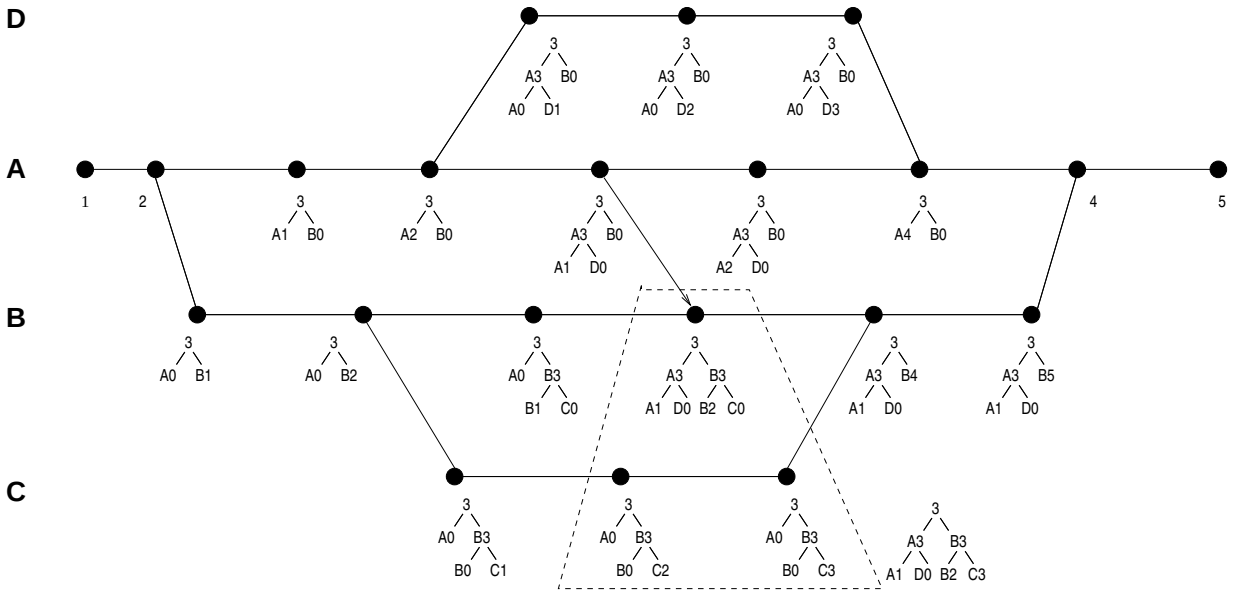

Abbildung 7.3: Zeitstempel für ein abstrahiertes Ereignis

Lückenerkennung

Der Satz von Methoden, soweit er bis jetzt angegeben wurde, erlaubt es, Ereignisse (Knoten), Kanten und Zeitstempel aus einem Ereignisverband zu entfernen, ohne die logische Zeit zu beeinträchtigen. Die starke Uhrbedingung bleibt erfüllt. Ein Problem tritt dabei jedoch auf, das nicht ohne weiteres vermieden werden kann: Die verwendeten logischen Uhren, egal ob Vektor- oder Baum-Uhren, verlieren ihre Lückenerkennungsfähigkeiten.

Wenn von Teilen des Verbandes abstrahiert wird, werden Ereignisse samt ihren Zeitstempeln ausgeblendet. Logischerweise resultiert dies in Lücken zwischen den verbleibenden Zeitstempeln. Dies negiert inhärent alle Lückenerkennungsfähigkeiten, die das verwendete Uhrensystem ohne die Abstraktion an den Tag gelegt haben würde – einfach weil wir nicht mehr auseinander halten können, welche Lücken aus der Abstraktion resultieren und welche nicht. Und wenn wir beginnen wollten, dieses Problem zu lösen, müssten wir uns dessen bewusst sein, dass und wie eine Abstraktion vorgenommen wurde. Auf der anderen Seite beinhaltet das Ziel einer Abstraktion für gewöhnlich Transparenz, und diese ginge dabei verloren.

Die Methoden, die bis hierhin vorgeschlagen wurden, sind von ihren Annahmen her sehr allgemein gehalten worden, so dass sie auf nahezu jedes System anwendbar sind. Außerdem funktionieren sie sowohl mit Vektor- als auch mit Baum-Uhren. Sie sind jedoch nicht geeignet für Anwendungen, die Lückenerkennung *und* Abstraktionstransparenz gleichzeitig voraussetzen.

Der nächste Unterabschnitt zeigt, wie auch dieses Problem, für eine bestimmte Abstraktionsmethode, durch Nutzung von Baum-Uhren gelöst werden kann.

7.2.4 Nutzung von Baum-Uhren

Wie bereits in Abschnitt 5.9 erklärt wurde, besitzen Baum-Uhren gewisse qualitative Vorteile gegenüber Vektor-Uhren. Dort wurde, gewissermaßen als Ausblick, angemerkt, dass sich für die sinnvolle Nutzung dieser Vorteile noch viele Anwendungen finden lassen sollten. Eine davon ist die Lösung des im letzten Unterabschnitt beschriebenen Problems mit der Lückenerkennung nach der Abstraktion eines Ereignisverbandes. Die dazu ausgenutzte Eigenschaft von Baum-Uhren ist deren, wie gesagt in Abschnitt 5.9 bereits angesprochenes, „Gedächtnis“.

Sei $\Sigma = \{\epsilon_1 \dots \epsilon_n\}$ die Menge der Ereignisse, die zu dem abstrahierten Ereignis E zusammengefasst werden. Der Kniff ist nun, die Ereignisse ϵ_i derart zu wählen, dass die folgenden sechs Bedingungen alle erfüllt sind:

- e_i^x ist ein Erzeugungsereignis und e_i^y ein Vereinigungsereignis derart, dass sich $c(i)$ in beiden Bäumen $t(e_i^x)$ und $t(e_i^y)$ auf derselben Ebene befindet.
- $\forall \epsilon_i \in \Sigma : \; e_i^x \Rightarrow \epsilon_i \Rightarrow e_i^y$.
- $\nexists \, e_j^z$, so dass $e_i^x \Rightarrow e_j^z \Rightarrow e_i^y$ und $e_j^z \notin \Sigma$.
- $\forall \epsilon \in \Sigma$: Wenn ϵ ein Terminierungsereignis ist, dann ist das korrespondierende Vereinigungsereignis entweder e_i^y oder $\in \Sigma$.
- $\forall \epsilon \in \Sigma$: Wenn ϵ ein Initialisierungsereignis ist, dann ist das korrespondierende Erzeugungsereignis entweder e_i^x oder $\in \Sigma$.
- $\forall \epsilon \in \Sigma$: Wenn ϵ weder ein Initialisierungsereignis noch ein Terminierungsereignis ist, dann sind sowohl das nächste als auch das vorige Ereignis nach der Prozessordnung $\rightarrow$ entweder $\in \Sigma$ oder e_i^x oder e_i^y (entsprechend).

Dieser Regelsatz besagt im Wesentlichen, dass von einer ganzen Prozesserzeugung, oder einem Prozess-Komplex, abstrahiert werden soll, wie in Abbildung 7.4 illustriert. Dort werden zwei verschiedene Ereignisverbände gezeigt. In jedem von diesen sind die zwei Abstraktionsmöglichkeiten eingetragen, die nach den obigen Regeln gültig sind.* Alle vier

* Mehr gültige Abstraktionen im Sinne dieser Regeln gibt es nicht.

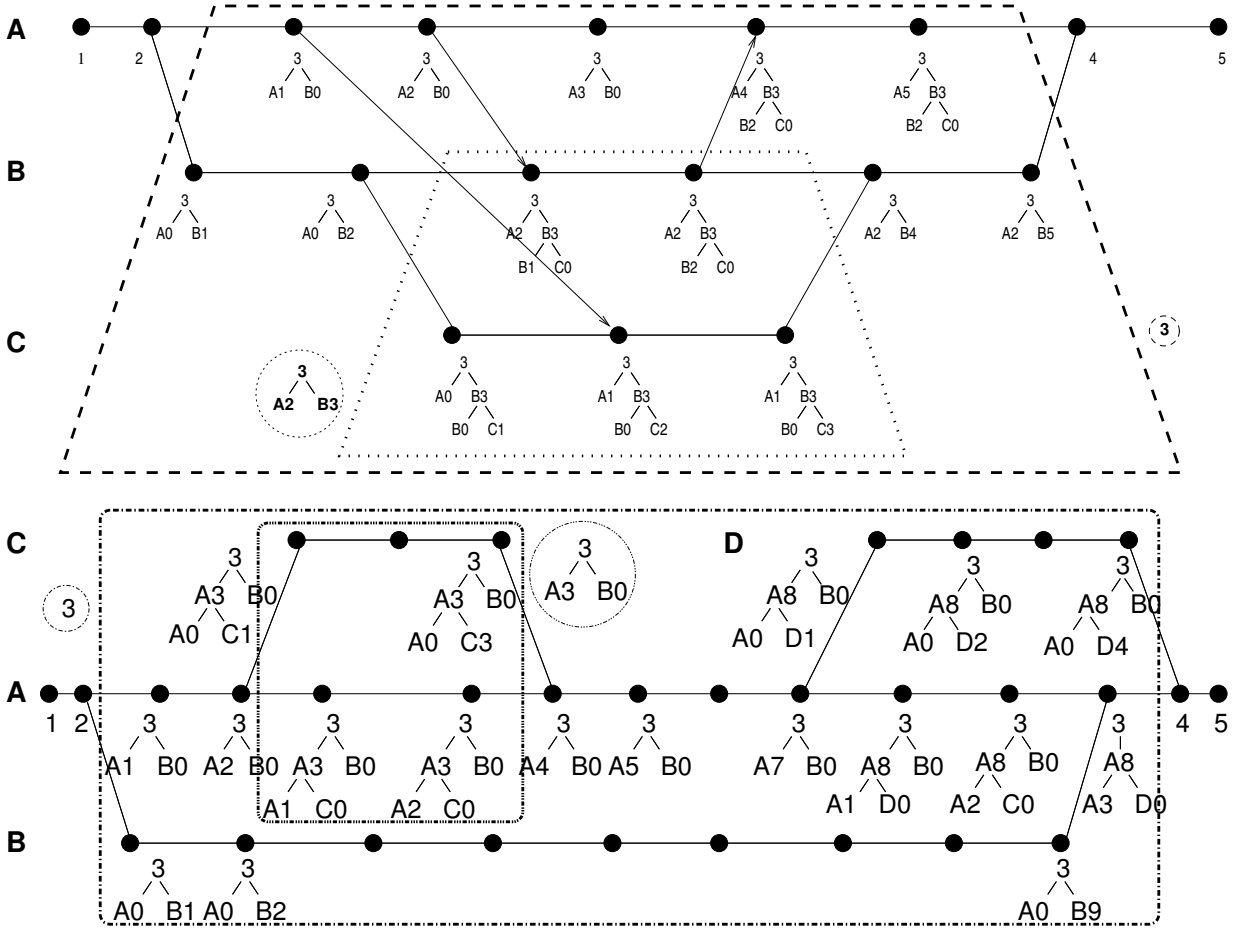

Abbildung 7.4: Abstraktion mit unbeeinträchtigter Lückenerkennung

Abstraktionen sind durch verschiedene Linienstile markiert; die jeweils dazugehörigen abstrahierten Zeitstempel sind entsprechend eingekreist.

Der springende Punkt ist der, dass wir einfach alles ausblenden, was zwischen einem Erzeugungsereignis und dem korrespondierenden Vereinigungsereignis liegt.* Auf diese Weise wird quasi so getan, als ob es keinerlei Grund gegeben hätte, überhaupt eine neue Baumebene einzuführen. Das abstrahierte Ereignis erhält dann denselben Zeitstempel, wie e_i^y mit $v(c(i))$ um 1 dekrementiert. Dieser passt so perfekt zwischen die Zeitstempel der nicht abstrahierten Ereignisse, dass keine Lücke bemerkbar ist (nach den Auswertungsalgorithmen für Baum-Uhren, die in Abschnitt 5.6.2 gegeben sind).

Selbstverständlich tritt auch hier immer noch ein Verlust von Kausalitätsinformation auf, aber das Verwerfen von Informationen ist, wie schon öfter angemerkt, überhaupt der Sinn von Abstraktion. Es muss sich lediglich mit den Anforderungen vertragen, die von der Anwendung der Abstraktion gestellt werden.

Da, wie gesagt, alle drei eingangs gestellten Fragen, die letztendlich die Abstraktionsmethode ausmachen, eng miteinander zusammenhängen, hängen die Antworten voneinander ab, und wir müssen der Vollständigkeit halber die Antwort auf Frage 2 an dieser Stelle noch geringfügig anpassen. Damit die Lückenerkennungsmerkmale der Baum-Uhren vollständig erhalten bleiben, *müssen* wir alle eingehenden Nachrichtenkanten entfernen, die laut Abschnitt 7.2.2 entfernt werden *dürfen*. Als Beispiel kann die gepunktete Abstraktion im oberen Teil von Abbildung 7.4 dienen. Die Nachrichtenkante von A nach C muss entfernt werden.

7.3 Abstraktion von Objekten

Nachdem im letzten Abschnitt die Abstraktion aktiver Systemkomponenten behandelt wurde, soll nun noch ein kurzer Blick auf passive Komponenten geworfen werden. Dazu wird das verwendete Systemmodell für diesen Abschnitt wie folgt erweitert:

Die Prozesse können auf gemeinsam genutzte Datenobjekte zugreifen, die in irgendeiner Art von gemeinsamem verteiltem Speicher (distributed shared memory, DSM) liegen. Wo und wie dieser Speicher implementiert ist, ist für die hier angestellten Betrachtungen ohne Belang. Es wird einfach angenommen, dass eine Menge $\Omega = \{o_1, o_2, \ldots\}$ von Datenobjekten existiert, von denen jedes eine Menge M_i von Methoden anbietet. Der Aufruf dieser

* Genau hier kommen die besonderen Merkmale der Baum-Uhren zum Tragen. Mit Vektor-Uhren wäre schon die gegenseitige Zuordnung dieser Ereignisse ohne zusätzliche Informationen schlichtweg nicht möglich.

Methoden sei der einzige Weg für Prozesse, die im Objekt gespeicherten Daten zu manipulieren. Eine Methode, die die Daten (zumindest potentiell) nicht nur ausliest, sondern auch verändert, wird *Schreibmethode* genannt. Anderenfalls handelt es sich um eine *Lesemethode*. Die Aufrufe der Methoden führen in den Prozessen zu Ereignissen zweier neuer Klassen, nämlich zu *Schreibereignissen* und *Leseereignissen*.

Wenn gemeinsam genutzte Objekte vorliegen, dann wird durch das Lesen und Schreiben auf diesen Objekten ein Informationsfluss zwischen den Prozessen etabliert, der auf irgendeine Weise durch das logische Zeitsystem erfasst werden muss. An dieser Stelle werden wir die einfache Lösung annehmen, dass ein Prozess beim Aufruf einer Schreibmethode seinen lokalen Zeitstempel in dem entsprechenden Objekt hinterlegt. Wann immer ein Prozess aus einem Objekt liest,* vereinigt er den dort hinterlegten Zeitstempel mit seinem eigenen, gerade so wie auch beim Empfang einer Nachricht. Für detailliertere Betrachtungen der Erfassung dieser Art von Kausalität siehe Kapitel 8 sowie [17].

Das Grundkonzept von Objekten beinhaltet Abstraktion schon inhärent, da Objekte (zwar nicht notwendigerweise, praktisch aber eigentlich immer) in irgendeiner Form hierarchisch organisiert sind. Um sich dies vor Augen zu führen, mag man etwa an ein Objekt denken, das einen Datenbankeintrag repräsentiert. Dieser besteht aus einer Reihe von Feldern, die wiederum aus Zeichen bestehen, welche ihrerseits letztendlich aus einzelnen Bits aufgebaut sind. Jede Schicht in in dieser Hierarchie kann als Abstraktion von den darunterliegenden Schichten aufgefasst werden. Ein anderes Beispiel, näher an den typischen Objektorientierungs-Konzepten, ist die Komposition eines Objekts aus verschiedenen anderen Objekten. Die Schreibmethode des Aggregatsobjekts würde in diesem Fall Schreibmethoden entsprechender Unterobjekte aufrufen.

Vor dem Hintergrund einer derartigen Situation (und ohne sie weiter zu konkretisieren) soll nun erklärt werden, wie Objekten und Unterobjekten Zeitstempel so zugeordnet werden können, dass die Konsistenz der logischen Zeit wieder erhalten bleibt:

Die jeweils kleinsten Unterobjekte, die von einem Prozess verändert werden, erhalten den aktuellen Zeitstempel des Prozesses selbst. Ein jedes Aggregatsobjekt erhält einen Zeitstempel, der eine Kombination der Zeitstempel aller direkten Unterobjekte darstellt, aus denen das Aggregatsobjekt besteht. Die Kombination der Zeitstempel erfolgt dabei mittels derselben Vereinigungsoperation (im Wesentlichen komponentenweises Maximum), die in Abschnitt 7.2.3 erklärt und verwendet wurde.

Auf diese Weise trägt nun jedes Objekt, auf jeder Abstraktionsebene, die vollständige Information über die Historie seines aktuellen Zustands, und damit den zur Interpretation des Zustands erforderlichen Kontext. Ein Prozess, der aus einem Objekt liest, erhält alle

* Dies kann auch im Rahmen einer Schreibmethode der Fall sein!

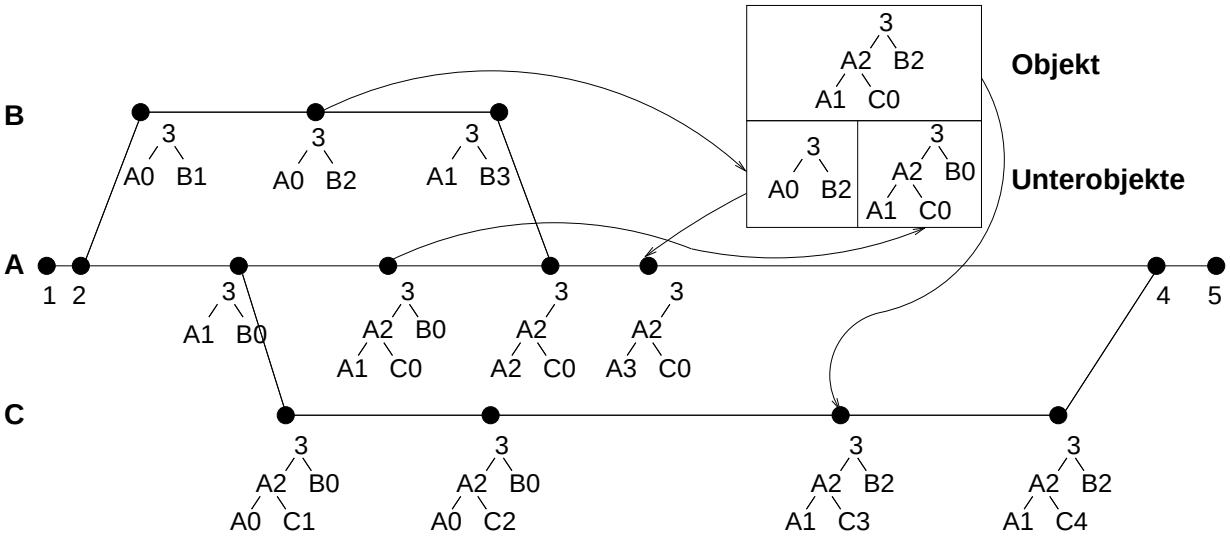

Abbildung 7.5: Logische Zeit und Objekt-Abstraktion

Informationen, die für seine Sicht auf das Objekt relevant sind, je nach verwendeter Abstraktionsebene. Die interne Uhr eines lesenden Prozesses wird genauso aktualisiert wie beim Empfang einer Nachricht, wobei der entsprechende Algorithmus für den verwendeten Uhrentyp zur Anwendung kommt (es können wieder sowohl Vektor- als auch Baum-Uhren eingesetzt werden).

Ein einfaches Beispiel, unter Verwendung von Baum-Uhren, ist in Abbildung 7.5 dargestellt. In diesem Beispiel tritt auch ein Fall von redundantem Informationsfluss auf. Der Zeitstempel des Leseereignisses in Prozess A wird durch den Lesevorgang nicht beeinflusst. Dies liegt daran, dass das Auslesen des Unterobjekts nur Kausalitätsinformation liefert, die ohnehin bereits Teil der Ereignis-Abhängigkeiten des Prozesses ist: Es gibt in diesem Graphen zwei Wege vom entsprechenden Schreib- zum besagten Leseereignis.

Auf diese Weise sind nicht nur die Zeitstempel konsistent mit dem Informationsfluss, sondern jeder interne oder externe Beobachter ist auch in der Lage, zusammengesetzte Objekte auf jeder gewünschten Abstraktionsebene zu betrachten und trotzdem die vollständige Information über die Historie des Objekts zu erhalten.

7.4 Zusammenfassung

In diesem Kapitel wurde detailliert ausgeführt, wie eine mittels logischer Uhren konstruierte Sicht auf die Kausalität einer verteilten Berechnung abstrahiert werden kann, ohne die logische Zeit zu kompromittieren. Mit Abstraktion ist hier gemeint, dass Details der

Berechnung je nach Bedarf ein- oder ausgeblendet werden können. Dies mag für viele Anwendungen interessant sein, insbesondere jedoch für Monitoring und verteiltes Debugging.

Das Ziel war, die Abstraktion in einer Weise vorzunehmen, die die Konsistenz der logischen Uhren erhält. Idealerweise sollte die Abstraktion darüber hinaus transparent sein. Es wurde gezeigt, wie diese Ziele durch Ausnutzung nützlicher Charakteristika von Baum-Uhren und durch den Einsatz eines maßgeschneiderten Satzes von Methoden erreicht werden können. Darüber hinaus wurde diese Art von Abstraktion auch im Zusammenhang mit gemeinsam genutzten passiven Objekten betrachtet.

Kapitel 8

Konsistente Ereignisordnung in DSM-Systemen

Bis hierher wurde die Erfassung der kausalen Zusammenhänge innerhalb einer verteilten Berechnung (fast) nur in Umgebungen betrachtet, die ausschließlich auf der Basis des reinen Berechnungsablaufs modelliert waren. Zugunsten eines Systemmodells von allgemeinstmöglicher Anwendbarkeit beschränkten sich die Grundelemente auf Ereignisse und ihre kausalen Abhängigkeiten, beides Bestandteile des Berechnungsablaufs und der Programmlogik. Selbst die Nachrichten, die man durchaus auch als rudimentäre Repräsentation von Daten verstehen könnte, wurden lediglich im Hinblick auf ihre die Kausalität der Programmlogik betreffenden Eigenschaften betrachtet. Die Wurzeln des bislang verwendeten Systemmodells gehen zurück bis zu den ersten Betrachtungen logischer Zeit durch Lamport [1].

In diesem Kapitel soll der Schwerpunkt nun auf eine Erweiterung dieses üblichen Modells gelegt werden, indem den nebenläufigen Prozessen ein abstrakter, gemeinsam nutzbarer Speicher zur Verfügung gestellt wird, was den Informationsfluss und damit die Abhängigkeiten signifikant vermehrt.

Verwandte Arbeiten wurden von Babaoğlu/Marzullo in [5] und von Li/Girard in [41] veröffentlicht, jeweils im Kontext allgemeiner Konsistenzmodelle bzw. deren Verifikation. In der vorliegenden Arbeit geht es darum, erstens, die relevanten Teile der verwandten Arbeiten zu konsolidieren, und, zweitens, einen in sich stimmigen Satz von nützlichen Definitionen, Zusammenhängen und Grundsatzüberlegungen zu erarbeiten, der als Grundlage zu weiterer Forschung dienen kann. Darüber hinaus wird in Abschnitt 8.3 eine exemplarische Realisierung der entwickelten Konzepte vorgestellt.

Der gesamte Inhalt dieses Kapitels ist in Zusammenarbeit des Autors mit Jörg Preißinger entstanden.

8.1 Systeme ohne DSM

So gut wie alle Abhandlungen über konsistente Sichten in verteilten Systemen betrachten ein Systemmodell, in dem die verteilten Prozesse ausschließlich mittels Nachrichtenaustausch miteinander kommunizieren. Beginnend mit Chandy und Lamport [28] blieben die Systemmodelle, die den Betrachtungen verteilter Konsistenz zu Grunde lagen, im Allgemeinen stets dieselben und variierten lediglich in wenigen zweitrangigen Annahmen, wie z. B. FIFO-Kanälen (siehe Abschnitt 3.1). Dabei spielte es auch keine Rolle, ob die jeweilige Arbeit im Kontext von Monitoring, Breakpointing, Debugging oder globalen Prädikaten entstand.

Auch die in dieser Arbeit bislang verwendeten Systemmodelle waren zugunsten von Allgemeinheit und/oder Vergleichbarkeit immer so nahe wie möglich an dem ursprünglichen Modell gehalten. Da dieses Kapitel in dieser Hinsicht eine Erweiterung darstellt, sowohl im Hinblick auf das zu Grunde liegende Systemmodell als auch, infolgedessen, auf die Untersuchungen als solche, sollen in diesem Abschnitt die wichtigsten Konzepte bezüglich konsistenter Sichten in reinen Nachrichtenaustausch-Systemen noch einmal kurz zusammengefasst werden.

8.1.1 Systemmodell

Wie in der Einleitung zu Abschnitt 8.1 bereits angekündigt, gilt in diesem Abschnitt das Systemmodell, das bereits in 3.1 beschrieben wurde. Daraus folgt, dass auch die happened-before-Relation aus Definition 3.1 auf Seite 29 ihre Gültigkeit behält.

8.1.2 Konsistente Sichten

Eine konsistente Sicht auf eine spezifische verteilte Berechnung muss den *globalen Zustand* der Berechnung wiedergeben. Dieser ist auf natürliche Weise als die Menge der *lokalen Zustände* aller beteiligten Prozesse zu definieren. Weitere Komponenten hat der globale Zustand nicht, da laut Modell keine weiteren zustandsbehafteten Komponenten, wie etwa ein gemeinsamer Speicher, zugelassen sind.

Das grundsätzliche Problem besteht darin, die lokalen Prozesszustände derart zu erfassen und zusammenzufügen, dass daraus ein sinnvoller globaler Zustand entsteht. Dies ist deswegen nicht trivial, weil dem verteilten System eine hinreichend genau synchronisierte globale Uhr fehlt (zumindest ist eine solche eben nicht vorausgesetzt). Ein sinnvoller oder

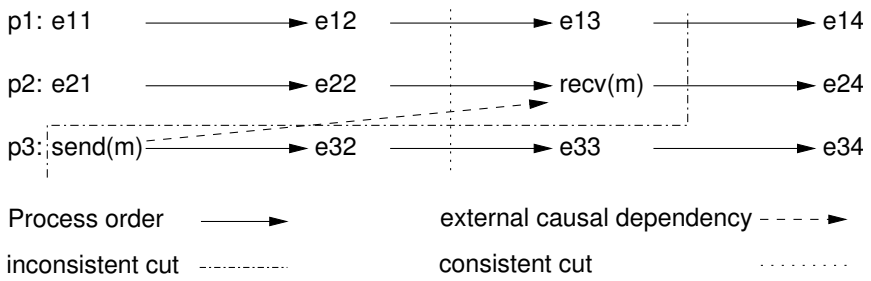

Abbildung 8.1: Abhängigkeiten und Schnitte

konsistenter globaler Zustand oder *konsistenter Schnitt** ist ein solcher, der während eines konkreten *Ablaufs* der Berechnung tatsächlich hätte auftreten können – auch wenn man nicht einwandfrei feststellen kann, ob er tatsächlich aufgetreten ist.

Im Wesentlichen heißt das, dass für jeden lokalen Prozesszustand im globalen Zustand gelten muss: Alle Ereignisse, von denen er kausal abhängt, müssen von lokalen Prozesszuständen reflektiert werden, die Teil des globalen Zustands sind. Beispielsweise darf der globale Zustand keinen Prozesszustand enthalten, der angibt, dass dieser Prozess eine Nachricht erhalten hat, wenn der lokale Zustand des Senderprozesses noch nicht das entsprechende Sendeereignis beinhaltet.

Letztendlich geht es darum, den globalen Zustand dadurch sinnvoll und interpretierbar zu machen, dass er in seinen logischen Kontext eingeordnet wird, und dieser wird wesentlich durch die relevante Historie bestimmt. Mangels einer globalen Zeit wird erst dadurch eine (partielle) Orientierung in dem von Zeit und Systemzuständen aufgespannten zweidimensionalen Raum möglich, dass durch die Kausalität der Ereignisse ein anderer Ordnungsbegriff etabliert wird. Dies ist genau der Sinn und Zweck von logischer Zeit (vergleiche hierzu auch Abschnitt 2.1.2 und Kapitel 3).

Abbildung 8.1 zeigt ein Beispiel mit einem konsistenten und einem inkonsistenten Schnitt durch eine Berechnung mit drei Prozessen. Ein Schnitt teilt eine Berechnung in zwei Teile, von denen einer als „Vergangenheit“ und der andere als „Zukunft“ betrachtet werden kann. Der globale Zustand, der durch den Schnitt beschrieben wird,† wäre dann die „Gegenwart“. Ein globaler Zustand ist genau dann konsistent, wenn keine kausale Abhängigkeit (hier

* Da in reinen Nachrichtenaustausch-Systemen sowohl Schnitte als auch globale Zustände jeweils durch eine Menge prozesslokaler Zustände (einer pro Prozess) ausgedrückt werden können, werden die beiden Begriffe an dieser Stelle synonym verwendet. Für Genaueres hierzu siehe Abschnitt 8.2.6.

† Die Pfeile zwischen den Ereignissen eines Prozesses können auch als seine Zustände interpretiert werden. Ein Ereignis, wie bereits in der Beschreibung des Systemmodells definiert, ändert schließlich den Zustand des jeweiligen Prozesses.

durch Pfeile dargestellt) von der Zukunft (hier rechts des Schnittes) in die Vergangenheit (hier links des Schnittes) weist.

Algorithmen, die einem Monitorprozess die Konstruktion eines globalen Zustands erlauben oder das allgemeine Problem zur Generierung verteilter Breakpoints adaptieren, werden in [28, 29, 33, 34, 35, 36, 37, 38, 39, 40, 41] gegeben.

8.2 Systeme mit DSM

In diesem Abschnitt wird das Systemmodell um einen gemeinsam nutzbaren verteilten Speicher (distributed shared memory, DSM) erweitert, ein mächtiges Mittel zur Interprozesskommunikation. Die Natur dieses Speichers wird hier nicht betrachtet. Es kann sich etwa um einen physikalisch verteilten Speicher handeln, bei dem die gemeinsame Nutzbarkeit durch eine Abstraktionsschicht in der Software realisiert wird, oder auch um einen nicht-verteilten physikalisch gemeinsam nutzbaren Speicher.

Wie zuvor ist das Ziel, eine konsistente Sicht auf eine gegebene verteilte Berechnung zu etablieren, was zu dem grundlegenden Problem führt, eine konsistente Ereignisordnung konstruieren zu müssen. Systeme mit einem gemeinsam nutzbaren verteilten Speicher werden in diesem Kapitel abkürzend als *DSM-Systeme* bezeichnet, reine Nachrichtenaustausch-Systeme als *MP-Systeme* (für message passing).

8.2.1 Systemmodell

In Abschnitt 8.2 gilt das Systemmodell, das bereits in 3.1 beschrieben wurde. Es wird jedoch um einen abstrakten, nicht genauer spezifizierten Speicher erweitert, auf den alle beteiligten Prozesse Zugriff besitzen. O. B. d. A. bestehe dieser Speicher aus einer Menge logischer *Speicherplätze* oder *Speicherobjekte*. Sowohl die Speicherobjekte als auch die elementaren Zugriffsoperationen *Lesen* und *Schreiben*, seien atomar. Der Wert, den ein Speicherplatz vor dem ersten darauf stattfindenden Schreibereignis besitzt, ist undefiniert.

8.2.2 Terminologie und Notationen

Die Prozessordnung, wie ein Abschnitt 3.1 eingeführt, erlaubt für verteilte Systeme zwei verschiedene Interpretationen. In der Forschung zur Konsistenzmodellverifikation wird sie als die totale Ordnung aller Ereignisse auf einem gegebenen Prozessor interpretiert, ohne Unterscheidung der einzelnen Prozesse, von denen ja durchaus mehrere in irgendeiner Form

verzahnt auf demselben Prozessor ausgeführt werden können [43, 19, 25]. Dieses Verständnis wird hier im Folgenden als *Prozessorordnung* bezeichnet werden, um den Unterschied zur *Prozessordnung* $\rightarrow$ hervorzuheben. Diese bezeichnet hier nach wie vor die totale Ordnung aller Ereignisse, die zu ein und demselben Prozess gehören, egal auf welchem Prozessor dieser ausgeführt wird.

Es ist offensichtlich, dass nur globale Ereignisordnungen konsistent sein können, die die Prozessordnung *respektieren*, d. h. intakt lassen. Für die Verifikation von Konsistenzmodellen ist die Unterscheidung zwischen Prozessor- und Prozessordnung unerheblich, weil man annehmen kann, dass auf jedem Prozessor nur ein Prozess ausgeführt wird, womit die beiden Ordnungen wieder identisch sind. In der Beobachtung verteilter Berechnungen ist die Unterscheidung hingegen sehr wichtig, weil das Verhalten aller Prozesse von Interesse ist, einschließlich mehrerer Prozesse auf demselben Prozessor. Die Betrachtung der Prozessorordnung anstatt der Prozessordnung würde hier einen Informationsverlust hinsichtlich der potentiellen Nebenläufigkeit kausal unabhängiger Ereignisse auf einem gegebenen Prozessor bedeuten. Dies liegt daran, dass durch die Prozessorordnung mehr Ereignisse total geordnet werden, als durch die Programmlogik vorgegeben (die Platzierung der Prozesse sowie das Scheduling auf den einzelnen Prozessoren sind in der Prozessorordnung bereits gegeben und eingeflossen). Die verlorene Information könnte für diverse Anwendungen der konsistenten Sicht durchaus signifikant sein; beispielsweise für eine dynamische Lastverteilung, die die Parallelität der Berechnungsausführung durch Migration von Prozessen steigern soll. Da im Folgenden das Ziel ist, mehr Informationen eine verteilte Berechnung zu sammeln, muss dabei also die Prozessordnung als Grundlage dienen, bei der keinerlei Einschränkungen durch Platzierung oder Scheduling einfließen.

In Gegenwart eines gemeinsam nutzbaren verteilten Speichers existieren analog zu den Sende- und Empfangsereignissen beim Nachrichtenaustausch nun auch *Schreib-* und *Leseereignisse*. Für Schreib-/Leseereignisse, die einen Wert a nach/vom Speicherplatz x schreiben/lesen, schreiben wir abkürzend $W(x)a/R(x)a$.

Die happened-before-Relation wird im Laufe dieses Kapitels mehrmals unterschiedlichen Bedürfnissen angepasst werden. Daher wird sie nicht an dieser zentralen Stelle an das neue Systemmodell angepasst, wie das in den anderen Kapiteln der Fall war.

8.2.3 Konsistenzmodelle

Das Konsistenzmodell eines gemeinsam nutzbaren verteilten Speichers spezifiziert, welche Anordnungen von potentiell nebenläufigen Speicherzugriffsereignissen effektiv auftreten dürfen. Adve und Gharachorloo haben in [44] rekapituliert, dass "effectively, the consistency model places restrictions on the values that can be returned by a read in a shared-memory program execution". Demnach erfordert eine konsistente Beobachtung die Berück-

sichtigung des zu Grunde liegenden Konsistenzmodells. Das Konsistenzmodell bietet letztendlich eine Abstraktion von der tatsächlichen Implementierung.

Im Folgenden sollen einige wichtige Konsistenzmodelle kurz beschrieben werden. Für weitere Informationen zu Konsistenzmodellen siehe beispielsweise [44] oder [51].

Strikte Konsistenz

Strikte Konsistenz ist das strengste Konsistenzmodell und besagt, dass jeder Lesevorgang auf einem Datum genau denjenigen Wert liefern muss, der von dem letzten Schreibvorgang auf demselben Datum dort hinterlegt wurde. Dabei ist der „letzte“ Schreibvorgang eindeutig und zeitlich definiert.

Strikte Konsistenz ist nicht nur das strengste, sondern auch das natürlichste und intuitivste Verständnis von Konsistenz. Dieses ist das Systemverhalten, das der Programmierer eines rein sequentiellen Systems voraussetzt und gewöhnlich auch vorfindet. Entsprechend wäre strikte Konsistenz auch in nebenläufigen Systemen wünschenswert, scheitert jedoch im Allgemeinen daran, dass keine systemweit eindeutige Uhr bzw. Zeit existiert, mittels derer man den „letzten“ Schreibvorgang eindeutig spezifizieren könnte. Daher behilft man sich in nebenläufigen verteilten Systemen gewöhnlich mit schwächeren Konsistenzmodellen. Einige davon werden in den folgenden Unterabschnitten kurz vorgestellt.

Sequentielle Konsistenz

Sequentielle Konsistenz wurde von Lamport [20] definiert. Die informelle Formulierung von Raynal [46] lautet, übersetzt, folgendermaßen:

> Sequentielle Konsistenz postuliert, dass ein nebenläufiges Programm korrekt ausgeführt wurde, wenn die Ausführung desselben Programms auf einem Ein-Prozessor-System dasselbe Ergebnis geliefert hätte. Das bedeutet, dass eine Ausführung korrekt ist, wenn ihre Ereignisse derart geordnet werden können, dass erstens die Prozessordnung erhalten bleibt, und zweitens jede Leseoperation den Wert der letzten Schreiboperation zurückliefert, wobei sich die „letzte Schreiboperation“ auf die totale Ordnung bezieht.

Insbesondere bedeutet dies, dass, im Gegensatz zur strikten Konsistenz, ein Lesevorgang nicht unbedingt den tatsächlich zuletzt geschriebenen Wert zurückliefern muss. Es muss lediglich sichergestellt sein, dass alle Prozesse *dieselbe* Abfolge von Speicherzugriffen sehen.

Sequentielle Konsistenz ist deshalb ein besonders wichtiges Modell, weil es das strengste und damit intuitivste der praktisch realisierbaren Konsistenzmodelle ist (eine globale

Zeit wird nicht vorausgesetzt). Aus diesem Grund wird es auch für die Konsistenzmodell-abhängigen Teile dieses Kapitels verwendet.

Kausale Konsistenz

Kausale Konsistenz, eingeführt von Hutto und Ahamad [45], schwächt das sequentielle Konsistenzmodell (s. o.) folgendermaßen ab:

Nur Ereignisse, zwischen denen potentiell eine kausale Abhängigkeit besteht, müssen von allen Prozessen in derselben Reihenfolge gesehen werden. Nebenläufige (d. h. kausal unabhängige) Ereignisse dürfen unterschiedlich beobachtet werden.

Durch diese Abschwächung kann ein höherer Grad an Nebenläufigkeit und damit eine höhere Performanz erzielt werden als mit sequentieller Konsistenz.

Es ist offensichtlich, dass zur Implementierung kausaler Konsistenz die kausalen Abhängigkeiten der Ereignisse bekannt sein müssen. Diese können mittels logischer Uhren ermittelt werden, wie in den vorangegangenen Kapiteln ausführlich erläutert.

PRAM- oder FIFO-Konsistenz

PRAM- oder *FIFO-Konsistenz* ist wiederum eine Abschwächung gegenüber der kausalen Konsistenz (s. o.) und wurde von Lipton und Sandberg in [47] beschrieben. Sie besagt folgendes:

Nur Schreibzugriffe (auf den DSM) eines jeden einzelnen Prozesses müssen von allen anderen Prozessen in der Reihenfolge gesehen werden, in der sie ausgeführt wurden. Schreibzugriffe verschiedener Prozesse dürfen in unterschiedlichen Ordnungen wahrgenommen werden, d. h. sie gelten als nebenläufig.

Schreiboperationen eines einzelnen Prozesses können in Form von Pipelining ausgeführt werden, daher der Name PRAM-Konsistenz; PRAM steht für „Pipelined RAM“. PRAM-Konsistenz ermöglicht daher eine noch weiter erhöhte Performanz und ist außerdem leicht zu implementieren.

Schwache Konsistenz

Die Semantik eines *schwachen Konsistenzmodells* [30] basiert auf Synchronisationsvariablen. Diese werden verwendet, um alle Schreiboperationen zwischen den Maschinen zu propagieren, und um lokale Daten an Änderungen anzupassen, die an anderen Stellen im System aufgetreten sind. Für schwache Konsistenzmodelle gelten dabei die folgenden drei Anforderungen:

- Die Zugriffe auf Synchronisationsvariablen sind sequentiell konsistent.
- Auf Synchronisationsvariable kann nur zugegriffen werden, wenn zuvor alle Schreiboperationen im System abgeschlossen wurden.
- Weder Lese- noch Schreiboperationen dürfen ausgeführt werden, solange nicht alle Zugriffe auf Synchronisationsvariablen abgeschlossen worden sind.

Im Gegensatz zu den bisher vorgestellten Konsistenzmodellen betrachtet schwache Konsistenz nicht einzelne Speicherzugriffe, sondern Gruppen davon. Daher ist dieses Modell nützlich, wenn die Speicherzugriffe hauptsächlich schubweise erfolgen. Darüber hinaus ist schwache Konsistenz das einzige der hier vorgestellten Modelle, in dem die Konsistenz in ihrer zeitlichen Gültigkeit eingeschränkt wird, und nicht in ihrer Semantik (diese entspricht sequentieller Konsistenz).

8.2.4 Unterschiede zu Systemen ohne DSM

Die Ereignisse einer verteilten Berechnung konsistent zu ordnen, ist in DSM-Systemen signifikant schwieriger als in MP-Systemen. Dafür gibt es mehrere Gründe. In beiden Fällen ist der entscheidende Punkt der Informationsfluss zwischen den Prozessen. Jeder mögliche Informationsfluss macht den Empfänger kausal abhängig vom Sender, und die Erfassung dieser kausalen Abhängigkeiten ist für die Konstruktion einer konsistenten Ereignisordnung essentiell. Es ist offensichtlich, dass der Informationsfluss in DSM-Systemen wesentlich schwerer zu verfolgen ist als in MP-Systemen. Während eine Nachricht einzigartig ist und Information nur von einem Sender zu genau einem spezifizierten Empfänger vermittelt, sind Speicherobjekte dazu geeignet, Information anonym an eine unspezifizierte Gruppe möglicher Leser zu verteilen. Darüber hinaus können Speicherobjekte repliziert vorliegen und, je nach dem Konsistenzmodell des DSM, in jeder Kopie unterschiedliche Werte besitzen.

Die Anzahl der möglichen konsistenten Ordnungen über einer Menge von Ereignissen wird durch das Konsistenzmodell bestimmt, dem der gemeinsam nutzbare verteilte Speicher genügt. Eine gegebene Ereignisordnung, die in einem System mit kausaler Konsistenz als konsistent gilt, hätte vielleicht in einem System mit sequentieller Konsistenz überhaupt nicht auftreten können. Das Konsistenzmodell bestimmt das Verhalten eines DSM-Systems hinsichtlich nebenläufiger Speicherzugriffe und ist damit ausschlaggebend für die Entscheidung, ob eine Ereignisordnung als konsistent gilt oder nicht. Für einen Überblick über Konsistenzmodelle siehe Abschnitt 8.2.3 und [44]. In Abschnitt 8.2.5 werden diejenigen Anforderungen an konsistente Ereignisordnungen besprochen, die vom verwendeten Konsistenzmodell unabhängig sind. In Abschnitt 8.2.6 wird dann der Einfluss des Konsistenz-

modells auf die Suche nach einer konsistenten Ordnung erklärt, basierend auf der weit verbreiteten sequentiellen Konsistenz.

8.2.5 Konsistenzmodellunabhängige Ordnung

Analog zu den kausalen Abhängigkeiten zwischen Sende- und Empfangsereignissen existieren Abhängigkeiten zwischen den Schreib- und Lesevorgängen für Speicherobjekte. Ein Wert kann nur dann von einem Speicherplatz gelesen werden, wenn er zuvor hineingeschrieben wurde. Daher muss in jeder konsistenten Ereignisordnung ein Leseereignis $R(x)a$ nach dem ersten Schreibereignis $W(x)a$ eingeordnet werden. Dies ist ganz ähnlich wie die Sende-/Empfangs-Abhängigkeiten beim Nachrichtenaustausch, jedoch mit dem Unterschied, dass mehrere identische Schreibereignisse auftreten können. Daher gehen wir davon aus, dass vor einem Leseereignis *mindestens* ein dazu passendes Schreibereignis aufgetreten sein muss.

Von dieser Regel gibt es jedoch eine Ausnahme zu beachten. Wie in der Beschreibung des Systemmodells bereits postuliert, ist der Inhalt eines Speicherobjekts vor dem ersten Schreibzugriff undefiniert. Das bedeutet, dass ein Leseereignis potentiell jeden beliebigen Wert aus einem Speicherplatz lesen könnte, wenn es nur vor dem ersten Schreibereignis auf diesen Platz auftreten könnte, ohne dass andere kausale Abhängigkeiten verletzt würden. Letztendlich wäre ein solcher Fall nichts anderes als ein nicht hinreichend synchronisiertes Systemverhalten, also eigentlich ein Fehler in der System- oder Anwendungsspezifikation. Dennoch ist dieser Fall nicht durch das Systemmodell ausgeschlossen und muss daher als möglich und konsistent eingestuft werden. Alle weiteren Leseereignisse, die andere Werte von derselben Stelle lesen, müssen dann natürlich nach einem solchen fragwürdigen Leseereignis eingeordnet werden. Dieser Fall wird hier nur der Vollständigkeit halber betrachtet, auch wenn er in (sauber programmierten) realen Systemen eigentlich nicht auftreten sollte.

Angesichts dieser zusätzlichen Abhängigkeiten wird die happened-before-Relation an dieser Stelle zu der nachfolgend definierten *vorläufigen DSM-Kausalitäts-Relation* $\overset{p}{\Rightarrow}$ erweitert, wobei $E_{W(x)a}$ und $E_{R(x)a}$ Teilmengen der Menge aller Ereignisse E bezeichnen, und zwar jeweils die Menge aller Ereignisse $W(x)a$ bzw. $R(x)a$.

Definition 8.1. $E_{R(x)a}$ sei die Menge aller Leseereignisse, die den Wert a aus einer Speicherstelle x lesen, und $E_{W(x)}$ sei die Menge aller Schreibereignisse auf x. Die *vorläufige DSM-Kausalitäts-Relation* oder *preliminary DSM causality relation* $\overset{p}{\Rightarrow}$ ist die kleinste Relation (im Sinne der Mächtigkeit), welche die folgenden drei Bedingungen erfüllt:

- $\forall e_i, e_j \in E$: Wenn $e_i \Rightarrow e_j$, dann $e_i \overset{p}{\Rightarrow} e_j$.

- $\forall e \in E_{R(x)a}$: eine der beiden folgenden Bedingungen muss erfüllt sein:

  1. $\exists\, e_w \in E_{W(x)a} :\ e_w \overset{p}{\Rightarrow} e.$

  2. $\forall e_x \in E_{W(x)} :\ (e \overset{p}{\Rightarrow} e_x)$ und
     $\forall e_r \in E_{R(x)b}$: Fall 1. muss gelten für $a \neq b.$

- $\forall e_i, e_j, e_k \in E$: Wenn $e_i \overset{p}{\Rightarrow} e_j$ und $e_j \overset{p}{\Rightarrow} e_k$, dann $e_i \overset{p}{\Rightarrow} e_k$.

□

8.2.6 Konsistenzmodellabhängige Ordnung

Wie in Abschnitt 8.2.3 bereits erklärt, spezifiziert das Konsistenzmodell eines gemeinsam nutzbaren verteilten Speichers, welche Anordnungen von potentiell nebenläufigen Speicherzugriffsereignissen effektiv auftreten dürfen, so dass, nach Adve und Gharachorloo [44], "effectively, the consistency model places restrictions on the values that can be returned by a read in a shared-memory program execution". Demnach erfordert eine konsistente Beobachtung die Konstruktion der beobachteten Ereignisordnung die Berücksichtigung des zu Grunde liegenden Konsistenzmodells.

In diesem Abschnitt geht es um die Aspekte der konsistenten Ereignisordnung, die vom zu Grunde liegenden Konsistenzmodell abhängig sind. Anstatt diese Aspekte nun völlig abstrakt und allgemein zu behandeln, wird im Folgenden beispielhaft die sequentielle Konsistenz zu Grunde gelegt, ein häufig verwendetes und semantisch recht einfaches Konsistenzmodell (siehe Abschnitt 8.2.3). Dieses Vorgehen sollte auch verdeutlichen, wie ähnliche Ansätze auf andere Konsistenzmodelle angewendet werden können.

Ereignisse hinsichtlich ihrer Prozessordnung zu sortieren stellt kein Problem dar, weil die Prozessordnung bereits in der happened-before-Relation nach Definition 3.1 auf Seite 29 erfasst ist.* Das Sammeln der benötigten Informationen kann, gerade wie in MP-Systemen, durch den Einsatz logischer Uhren erfolgen.

Darüber hinaus müssen jedoch auch die Schreib- und Leseereignisse geordnet werden. Zugriffe auf unterschiedliche Speicherstellen sind dabei, soweit nicht anderweitig geordnet (d. h. durch die Prozessordnung oder Nachrichten), kausal unabhängig voneinander. Für diese kann daher jede Ordnung, die die Prozessordnung und die durch Nachrichtenaustausch entstehenden Abhängigkeiten berücksichtigt, als konsistent gelten. Daher wird der

* Die besagte Definition wurde zwar für MP-Systeme formuliert, wird in diesem Abschnitt jedoch als Basis für die DSM-bezogenen Relationen verwendet.

Schwerpunkt im Folgenden auf Ereignissen liegen, die auf jeweils denselben Speicherplatz zugreifen. Die Annahme, dass der gemeinsam nutzbare verteilte Speicher sequentielle Konsistenz implementiert, befähigt uns dabei, von tatsächlichen Speicheradressen, Replikation und Ähnlichem zu abstrahieren.

Der zweite Teil von Raynals Definition sequentieller Konsistenz (siehe Abschnitt 8.2.3) besagt, dass jedes Leseereignis, das einen spezifischen Wert liefert, einerseits nach einem entsprechenden Schreibereignis eingeordnet werden muss, andererseits aber vor dem nächsten Schreibereignis auf diese Speicherstelle. Dabei bezieht sich „das nächste“ hier auf die totale Ordnung, und „passend“ auf ein beliebiges Schreibereignis, dass den gelesenen Wert in den Speicherplatz geschrieben hat, ohne Unterscheidung identischer Schreibereignisse.

Die notwendige Vorbedingung, um einen Wert a aus Speicherstelle x lesen zu können, ist nichts anderes als die Existenz des Wertes a in Speicherstelle x. Aus diesem Grund betrachten wir das Leseereignis nicht als kausal abhängig von demjenigen Schreibereignis, mit dem der Wert tatsächlich zuvor geschrieben wurde. Das Leseereignis ist vielmehr lediglich abhängig vom Zustand der Speicherzelle. Dieser wiederum hängt vom jeweils letzten Schreibzugriff ab. Es wird also postuliert, dass ein Leseereignis $R(x)a$ erst nach irgendeinem Schreibereignis auf die Stelle x auftreten kann. Zusätzlich muss berücksichtigt werden, dass der undefinierte Anfangszustand von x so ausgelesen werden könnte wie in Abschnitt 8.2.5 beschrieben. Für diese Forderungen ist die vorläufige DSM-Kausalitäts-Relation nicht hinreichend, was zu folgendem Satz führt:

Satz 8.1. Sei $E_{R(x)a}$ die Menge aller Leseereignisse, die den Wert a von Speicherstelle x lesen, und $E_{W(x)a}$ die Menge aller Schreibereignisse, die den Wert a in den Speicherplatz x schreiben. Die Erfüllung der folgenden drei Bedingungen ist notwendig und hinreichend dafür, dass eine totale Ereignisordnung $\succ$ konsistent ist:*

1. $\forall e_i, e_j \in E$: Wenn $e_i \rightarrow e_j$, dann $e_i \succ e_j$.
2. $\forall e_i^x, e_j^y \in E$: Wenn e_i^x ein Sendeereignis und e_j^y das Empfangsereignis derselben Nachricht ist, dann $e_i^x \succ e_j^y$.
3. $\forall e \in E_{R(x)a}$: Es muss einer der folgenden beiden Fälle erfüllt sein:
   (a) $\big(\exists\, e_w \in E_{W(x)a} :\ e_w \succ^* e\big)$ und $\big(\forall e_x \in E_{W(x)} \setminus \{e_w\} :\ e_x \succ^* e_w$ oder $e \succ^* e_x\big)$.
   (b) $\big(\forall e_x \in E_{W(x)} :\ e \succ^* e_x\big)$ und $\big(\forall e_r \in E_{R(x)b} :$ Fall (a) muss gelten für $a \neq b\big)$.

□

* $e_i \succ^* e_j$ ist die übliche Notation für einen transitiven Pfad in $\succ$ von e_i nach e_j.

Beweis: Zuerst wird gezeigt, dass alle im Satz geforderten Bedingungen notwendig für eine konsistente* Ereignisordnung sind. Die ersten beiden Bedingungen sind direkt aus Lamports happened-before-Relation abgeleitet, und ihre Notwendigkeit ist aus MP-Systemen bereits bekannt [28]. Nehmen wir also an, die totale Ordnung $\dot{\succ}$ sei konsistent, verletze aber Bedingung 3. Während einer Ausführung des Systems kann ein Leseereignis entweder einen Wert lesen, der anfänglich in der betreffenden Speicherstelle stand, nämlich wenn noch kein Schreibereignis auf dieser Speicherstelle stattgefunden hat, oder einen Wert, der von einem Schreibereignis explizit hineingeschrieben wurde. Es gibt zwei Möglichkeiten für die Ordnung $\dot{\succ}$, Bedingung 3 zu verletzen. Im ersten Fall würde $\dot{\succ}$ zwei Leseereignisse $e_i \in E_{R(x)a}$ und $e_j \in E_{R(x)b}$ vor dem ersten Schreibereignis für Speicherplatz x einordnen. Dies könnte in einer tatsächlichen Ausführung jedoch niemals auftreten und führt daher zu einem Widerspruch mit der Annahme, $\dot{\succ}$ sei konsistent. Im zweiten Fall würde $\dot{\succ}$ ein Schreibereignis $e_x \in E_{W(x)b}$ zwischen einem anderen Schreibereignis $e_w \in E_{W(x)a}$ und darauf folgenden Leseereignissen $e \in E_{R(x)a}$ einordnen. Dies würde bedeuten, dass ein Leseereignis einen bereits wieder überschriebenen Wert zurückliefern würde, was in einem sequentiell konsistenten DSM-System (hier immer noch angenommen) nicht auftreten kann und somit wieder zu einem Widerspruch führt.

Nun wird gezeigt, dass die in Satz 8.1 geforderten Bedingungen hinreichend sind, um eine konsistente Ereignisordnung zu gewährleisten. Angenommen, es läge eine totale Ordnung $\dot{\succ}$ vor, die alle Bedingungen erfüllt, aber trotzdem inkonsistent ist. Aus dieser Annahme folgt, dass $\dot{\succ}$ ein Ereignis e_i, das tatsächlich nur vor einem anderen Ereignis e_j auftreten kann, nach e_j einordnet, ohne die Bedingungen zu verletzen. Es gibt drei Möglichkeiten für eine Ordnung $\dot{\succ}$, inkonsistent zu werden. Im ersten Fall muss e_i nach der Prozessordnung vor e_j auftreten (d. h. beide Ereignisse gehören zum selben Prozess), wird aber so eingeordnet, dass $e_j \dot{\succ}^* e_i$. In diesem Fall führt die Annahme, $\dot{\succ}$ erfülle Bedingung 1, zum Widerspruch. Der zweite Fall liegt ähnlich, wobei nun e_j den Empfang einer Nachricht repräsentiert, und e_i deren Versand. Dies führt, analog zum ersten Fall, zu einem Widerspruch mit Bedingung 2. Im dritten Fall müsste die – per Annahme inkonsistente – Ordnung $\dot{\succ}$ ein Leseereignis $e_j \in E_{R(x)a}$ nicht so nach einem Schreibereignis $e_i \in E_{W(x)a}$ einordnen, dass kein weiteres Schreibereignis dazwischenliegt. Eine andere Möglichkeit wäre nur noch, mindestens zwei verschiedene Leseereignisse $e_i \in E_{R(x)a}$ und $e_j \in E_{R(x)b}$ vor dem allerersten Schreibereignis auf Speicherstelle x einzuordnen. Beide Möglichkeiten führen jedoch gleichermaßen zum Widerspruch mit Bedingung 3. □

* Zur Erinnerung: Eine Ereignisordnung ist genau dann konsistent, wenn sie während einer Ausführung der Berechnung tatsächlich auftreten könnte. Was während einer Ausführung tatsächlich auftreten kann, ist durch das Konsistenzmodell (siehe Abschnitt 8.2.3) definiert. Hier wird sequentielle Konsistenz angenommen.

Satz 8.1 führt direkt zur finalen Definition der DSM-Kausalitäts-Relation, diesmal als Bezeichnung für eine Klasse von Relationen. $E_{W(x)}$ bezeichne die Menge aller Schreibereignisse auf Speicherplatz x.

Definition 8.2. Eine *DSM-Kausalitäts-Relation* $\overset{c}{\Rightarrow}$ ist eine transitive Ordnungsrelation, die alle durch Satz 8.1 geforderten Bedingungen erfüllt. □

Korollar 8.1. Die folgenden beiden Aussagen ergeben sich direkt aus Satz 8.1:

- Alle DSM-Kausalitäts-Relationen sind konsistent.
- Jede konsistente Ereignisordnung ist eine DSM-Kausalitäts-Relation.

□

Ein anderes Problem ist die konstruktive Erzeugung einer totalen DSM-Kausalitäts-Relation. Wie Gibbons und Korach in ihrer Arbeit über die Verifikation von Konsistenzmodellen [25] gezeigt haben, ist dieses Problem für sequentiell konsistente DSM-Systeme (und ohne weitere Informationen) np-vollständig. Hinsichtlich der Suche nach einer solchen totalen Ordnung stellen sich insbesondere zwei Probleme:

- die Ordnung mehrerer Schreibereignisse $e_w \in E_{W(x)}$
- die Zuordnung eines Leseereignisses $e_r \in E_{R(x)a}$ zu einem Schreibereignis $e_w \in E_{W(x)a}$, dem es ohne Verletzung anderer Abhängigkeiten nachgeordnet werden kann

In diesem Zusammenhang haben Gibbons und Korach die *Schreibordnung (write-order)* und die *Leseabbildung (read-mapping)* eingeführt. Sie haben gezeigt, dass eine sequentiell konsistente totale Ordnung in $O(n\ log(n))$ konstruiert werden kann. Das Opfer, das dafür gebracht werden muss, ist die Inkaufnahme von noch mehr Einschränkungen in der Ordnung als nötig, was dazu führt, dass nicht alle konsistenten Ordnungen akzeptiert werden. Aber, und dies ist für viele Anwendungen ausreichend, es kann immer eine konsistente Ordnung konstruiert werden (vorausgesetzt das DSM-System implementiert sequentielle Konsistenz), weil die zusätzlichen Informationen direkt aus dem tatsächlichen Auftreten der Ereignisse in einer beobachteten Ausführung gewonnen werden. Im Folgenden werden kurz die Ergebnisse von Gibbons und Korach erläutert und danach wird die Definition der DSM-Kausalitäts-Relationen zu einer ordnungs-einschränkenden Version verändert werden, die als Basis für effiziente Implementierungen dienen kann.

Definition 8.3. Die Schreibordnung $\succ_{wo}$ ist die totale Ordnung aller Schreibereignisse auf derselben Speicherstelle, in der sie während einer beobachteten Systemausführung aufgetreten sind. □

Diese Information muss während der Ausführung gesammelt und dann dem Ordnungs-Algorithmus zur Verfügung gestellt werden. Die totale Ordnung wird dann unter Berücksichtigung der Schreibordnung konstruiert.

In realen DSM-Systemen können von verschiedenen Prozessen die gleichen Werte an dieselbe Speicherstelle geschrieben werden. Ein Leseereignis, das einen solchen Wert ausliest, kann im Prinzip jedem beliebigen dieser Schreibereignisse nachgeordnet werden. Diese Möglichkeiten sind durch andere Abhängigkeiten jedoch oft eingeschränkt. Eines der zur Auswahl stehenden Schreibereignisse ist jedoch trotz aller sonstigen Einschränkungen immer als Vorgänger für das Leseereignis verwendbar, nämlich das Schreibereignis, das den gelesenen Wert in einer beobachteten Ausführung tatsächlich zuletzt geschrieben hat.

Definition 8.4. Die Leseabbildung ist eine Funktion $f_{rm} : E_{R(x)a} \mapsto E_{W(x)a} \cup \{\bot\}$, die jedem Leseereignis das eindeutige Schreibereignis zuordnet, das den Wert a tatsächlich zuletzt in die Speicherstelle x geschrieben hat. □

Auch die Leseabbildung muss während eines beobachteten Systemablaufs aufgezeichnet werden, damit diese Informationen bei der Konstruktion der totalen Ereignisordnung zur Verfügung stehen.

Die Leseereignisse selber sind nur durch die Prozessordnung geordnet (und möglicherweise indirekt durch Nachrichtenaustausch), weil Leseereignisse verschiedener Prozesse immer (potentiell) nebenläufig sind und parallel ausgeführt werden dürfen.

Wenn ein Leseereignis den undefinierten Anfangswert eines Speicherobjekts liest, wird es von der Leseabbildung auf $\bot$ abgebildet. In diesem Fall muss es dem ersten Schreibereignis auf der entsprechenden Speicherstelle vorgeordnet werden.

Auf der Basis von Leseabbildung, Schreibordnung und Definition 8.2 können wir nun die folgende Relation definieren:

Definition 8.5. Es seien eine Schreibordnung $\succ_{wo}$ und eine Leseabbildung f_{rm} gegeben. Die *eingeschränkte DSM-Kausalitäts-Relation* oder *restricted DSM causality relation* $\overset{r}{\Rightarrow}$ ist dann die kleinste Ordnungsrelation, die die folgenden vier Bedingungen erfüllt:

- $\forall e_i, e_j \in E$: Wenn $e_i \Rightarrow e_j$, dann $e_i \overset{r}{\Rightarrow} e_j$.
- $\forall e_i, e_j \in E_{W(x)}$: Wenn $e_i \succ_{wo} e_j$, dann $e_i \overset{r}{\Rightarrow} e_j$.
- $\forall e \in E_{R(x)a}$: Es muss einer der folgenden beiden Fälle erfüllt sein:
  - $\exists e_w \in E_{W(x)a} :\ f_{rm}(e) = e_w$ und $e_w \overset{r}{\Rightarrow} e$ und $\big(\forall e_x \in E_{W(x)} \setminus \{e_w\} :$ wenn $(e_w \succ_{wo} e_x)$, dann $(e \overset{r}{\Rightarrow} e_x)\big)$.
  - $f_{rm}(e) = \bot$ und $\forall e_x \in E_{W(x)} :\ e \overset{r}{\Rightarrow} e_x$.

- $\forall e_i, e_j, e_k \in E$: Wenn $e_i \overset{r}{\Rightarrow} e_j$ und $e_j \overset{r}{\Rightarrow} e_k$, dann $e_i \overset{r}{\Rightarrow} e_k$.

□

Es ist hervorzuheben, dass die Menge der totalen Ereignisordnungen, die die eingeschränkte DSM-Kausalitäts-Relation (eine partielle Ordnung!) erfüllen, lediglich eine Teilmenge aller möglichen konsistenten Ordnungen umfasst. Dies stellt natürlich einen Nachteil gegenüber der uneingeschränkten DSM-Kausalitäts-Relation aus Definition 8.2 dar. Der Vorteil der eingeschränkten DSM-Kausalitäts-Relation liegt darin, dass sie aufgrund der Nutzung von Zusatzinformationen die *effiziente Konstruktion* einer totalen Ordnung erlaubt. Aus diesem Grund wird der Rest dieses Abschnitts auf Definition 8.5 basieren.

Nun, da wir eine totale Ereignisordnung konstruieren können, sind wir in der Lage, konsistente Schnitte und konsistente globale Zustände für DSM-Systeme zu definieren. Der Schnitt oder globale Zustand darf lediglich die eingeschränkte DSM-Kausalitäts-Relation nicht verletzen.

Definition 8.6. $P = \{p_1, p_2, \ldots, p_n\}$ sei die endliche Menge von Prozessen und E die Menge aller Ereignisse. C sei ein Schnitt, der aus der Menge $S = \{s_1, s_2, \ldots, s_n\}$ lokaler Prozesszustände besteht, und $E_c = \{e_1, e_2, \ldots, e_n\}$ sei die Menge der direkten Vorgängerereignisse zu den Zuständen in S (d. h. die Menge der Ereignisse, die diese Zustände hervorrufen). Sei ferner $E_p = E_c \cup \{e_i \mid e_i \overset{r}{\Rightarrow} e_j,\ e_j \in E_c\}$ die Menge aller Ereignisse in der Vergangenheit des Schnittes. Dann ist C ein *konsistenter Schnitt*, wenn gilt:

$$\forall e_i, e_j \in E: \text{ Wenn } e_i \overset{r}{\Rightarrow} e_j \text{ und } e_j \in E_p, \text{ dann } e_i \in E_p.$$

□

In reinen MP-Systemen ist jeder Schnitt im Sinne der obigen Definition gleichzeitig auch ein globaler Zustand, da der Systemzustand, gerade wie der Schnitt, durch einen vollständigen Satz lokaler Prozesszustände hinreichend charakterisiert ist.*

In DSM-Systemen ist die Situation inhärent anders, da ein globaler Zustand mehr als nur die lokalen Prozesszustände erfassen muss, nämlich auch den Zustand des gemeinsam genutzten Speichers – ein globaler Zustand soll eine umfassende Beschreibung des Systemzustands liefern, mit allen Informationen die nötig sind, um beispielsweise einen Rollback zu diesem Zustand machen zu können.

* Vorausgesetzt ist dabei, dass jede Nachricht zusammen mit dem entsprechenden Zustand gespeichert wird, so dass der Zustand vollständig erfasst ist und Nachrichten, die sich zum (logischen) Zeitpunkt des Schnittes gerade in der Übertragung befinden, gegebenenfalls neu gesendet werden können. Diese Voraussetzung kann o. B. d. A. postuliert werden, was für den Rest des Abschnitts hiermit getan sei.

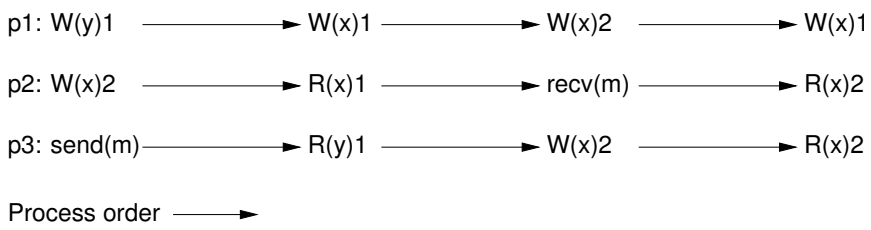

Abbildung 8.2: Ereignisse, geordnet nur durch die Prozessordnung

Da die Erfassung eines Speicherabbilds, das konsistent mit einem gegebenen Schnitt ist, eine komplexe und schwierige Aufgabe ist, werden wir diese Erfassung hier implizit vornehmen, indem wir den globalen Zustand auf der Basis der total geordneten Historie definieren, die zu den im Schnitt reflektierten Prozesszuständen geführt hat.* Die Historie wird durch die eingeschränkte DSM-Kausalitäts-Relation gegeben und ermöglicht implizit die Rekonstruktion des Speicherzustands, da sie die Antwort auf die Frage enthält, welches das jeweils letzte Schreibereignis für einen jeden Speicherplatz war. In dem Fall, dass für eine Speicherstelle noch kein Schreibereignis aufgetreten ist, wird natürlich der entsprechende (undefinierte und daher als zufällig angenommene) Anfangswert benötigt.

Definition 8.7. $P = \{p_1, p_2, \ldots, p_n\}$ sei die endliche Menge von Prozessen und E die Menge aller Ereignisse. C sei ein Schnitt, der aus der Menge $S = \{s_1, s_2, \ldots, s_n\}$ lokaler Prozesszustände besteht, und $E_c = \{e_1, e_2, \ldots, e_n\}$ sei die Menge der direkten Vorgängerereignisse zu den Zuständen in S (d. h. die Menge der Ereignisse, die diese Zustände hervorrufen). Sei ferner $E_p = E_c \cup \{e_i \mid e_i \overset{\tau}{\Rightarrow} e_j,\ e_j \in E_c\}$ die Menge aller Ereignisse in der Vergangenheit des Schnittes.

Ein *globaler Zustand* ist dann ein Tupel $T = (E_p, \succ)$, wobei E_p durch $\succ$ total geordnet ist. Wenn C konsistent ist und $\succ$ alle Bedingungen erfüllt, die in Satz 8.1 gefordert werden (was $\overset{c}{\Rightarrow}$ und $\overset{\tau}{\Rightarrow}$ tun), dann ist T ein *konsistenter globaler Zustand*. □

8.2.7 Illustration

Die zuvor diskutierten Ergebnisse sollen nun noch anhand von zwei Abbildungen illustriert werden. Abbildung 8.2 zeigt drei Prozesse $p1$, $p2$ und $p3$, einige Lese- und Schreibereignisse, sowie den Versand einer Nachricht. Die Ereignisse sind ausschließlich durch die Prozessordnung partiell geordnet, andere Abhängigkeiten sind nicht eingezeichnet. Daher wäre die Aufgabe, die Ereignisse total zu ordnen, um einen konsistenten Schnitt oder globalen Zustand zu finden, np-vollständig [25]. Es werden zusätzliche Informationen benötigt.

* Tatsächlich ist dies mehr Information als für den globalen Zustand eigentlich benötigt wird.

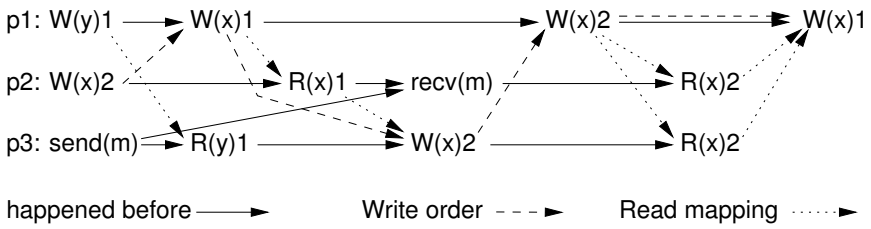

Abbildung 8.3: Ereignisse, geordnet durch die eingeschränkte DSM-Kausalitäts-Relation

Abbildung 8.3 zeigt genau dieselben Ereignisse. Diesmal jedoch ist die eingeschränkte DSM-Kausalitäts-Relation eingezeichnet, und zwar unterschieden nach den Teilen, die jeweils aus der happened-before-Relation, der Schreibordnung und der Leseabbildung resultieren. Die Ereignisse sind entsprechend dieser Ordnung bereits an einer (nicht eingezeichneten) Zeitachse ausgerichtet, die von links nach rechts verläuft. Je weiter links ein Ereignis angeordnet ist, desto früher tritt es in der kausalen Kette auf. Die normalen Pfeile geben die happened-before-Relation wieder, wie sie bereits aus MP-Systemen bekannt ist. Dadurch werden das Sende- und das Empfangsereignis kausal geordnet.

Die gestrichelten Pfeile geben die Schreibordnung wieder, die ebenfalls ein Teil der eingeschränkten DSM-Kausalitäts-Relation ist: Alle Schreibereignisse auf dieselbe Speicherstelle werden total geordnet, so wie sie während des Systemablaufs beobachtet wurden.

Die Leseabbildung wird durch die gepunkteten Pfeile dargestellt. Sie stellt sicher, dass jedes Leseereignis dem (laut Systembeobachtung) korrespondierenden Schreibereignis nach-, jedem anderen Schreibereignis auf die betreffende Speicherstelle jedoch vorgeordnet ist. Zum Beispiel sind die beiden Leseereignisse $R(x)2$ in $p2$ und $p3$ nach dem zweiten Schreibereignis $W(x)2$ eingeordnet, aber bevor der Speicherplatz x durch $W(x)1$ in $p1$ wieder überschrieben wird.

Ereignisse, die vertikal untereinander ausgerichtet sind, sind kausal unabhängig voneinander und daher nebenläufig; es gibt hier keinen Pfad von einem zum anderen.

Jede totale Ordnung, die die eingeschränkte DSM-Kausalitäts-Relation respektiert, ist konsistent. Graphisch würde die Herstellung einer solchen Ordnung bedeuten, dass die Ereignisse nicht in drei sondern in einer einzigen Reihe angeordnet werden müssten, und zwar derart, dass alle Pfeile zwischen den Ereignissen von links nach rechts weisen. Jede Gruppe von nebenläufigen Ereignissen kann dabei in eine beliebige Reihenfolge gebracht werden. Durch Permutation der nebenläufigen Ereignisse können alle konsistenten totalen Ordnungen konstruiert werden, die die eingeschränkte DSM-Kausalitäts-Relation respektieren. Ein Beispiel: eine konsistente totale Ordnung kann hier mit einem von genau drei Ereignissen beginnen, nämlich $W(y)1$ in p1, $W(x)2$ in p2 oder $send(m)$ in p3.

Eine Linie, die vertikal durch Abbildung 8.3 gezogen wird, repräsentiert genau dann einen konsistenten Schnitt, wenn es keinen Pfeil gibt, der von der rechten Seite der Linie in die linke verläuft. Wenn zusätzlich alle Ereignisse auf der linken Seite der Linie (d. h. in der „Vergangenheit“) im Hinblick auf die eingeschränkte DSM-Kausalitäts-Relation total geordnet sind, dann definiert der Schnitt einen konsistenten globalen Zustand, wobei durch die total geordneten Ereignisse der Vergangenheit der Zustand des gemeinsamen Speichers festgelegt wird.

8.3 Realisierung einer konsistenten Ereignisordnung

Wie man eine konsistente Sicht auf reine MP-Systeme konstruieren kann, haben bereits Lamport und viele andere untersucht (vgl. auch Abschnitt 8.1). In Abschnitt 8.2 wurden diese Betrachtungen und Konzepte auf Systeme mit gemeinsam nutzbarem verteiltem Speicher erweitert. Damit ist eine Basis für die Implementierung dieser Konzepte geschaffen worden. In diesem Abschnitt sollen nun Vorgehensweise, Probleme, Lösungen und Erfahrungen hinsichtlich einer praktischen Realisierung besprochen werden.

Ziel der Implementierung war die Konstruktion des der ausgeführten Berechnung zu Grunde liegenden Ereignisverbands. Dies soll Schritt für Schritt zur Laufzeit geschehen, so dass der Fortschritt der Berechnung anhand des Ereignisverbands gewissermaßen visualisiert werden kann.

Die praktische Realisierung erfolgte hauptsächlich im Rahmen der Diplomarbeit von Sebastian Haas [21].

8.3.1 MoDiS

Auch diese Implementierung entstand im Rahmen der Arbeit am Experimentalsystem MoDiS. Dieses wurde bereits in Abschnitt 6.3 vorgestellt, so dass an dieser Stelle lediglich auf Seite 87 verwiesen wird.

8.3.2 Beobachtung von Speicherzugriffen

Wie bereits in Abschnitt 8.2 deutlich gemacht wurde, ist die Aufzeichnung von Ereignissen und ihren gegenseitigen kausalen Abhängigkeiten in Systemen mit gemeinsam nutzbarem verteilten Speicher besonders kompliziert.

Der übliche Nachrichtenaustausch ist in MoDiS (in Form von K-Ordern, siehe Abschnitt 6.3.4) bereits in geeigneten Bibliotheken implementiert, die leicht um entsprechende Auf-

zeichnungsmechanismen ergänzt werden können. Derselbe Ansatz wird in manchen Systemen, insbesondere in Systemen mit geringer DSM-Nutzung, für den gemeinsamen Speicher verfolgt. In MoDiS basiert die DSM-Nutzung jedoch teilweise auf direkten Maschinenbefehlen. Der gemeinsam nutzbare verteilte Speicher wird unter Verwendung des Seitenfehler-Mechanismus direkt in den virtuellen Speicher eines Prozesses gemappt. Dort verbleibt er, solange er gültig ist. Nach dem ersten Zugriff kann daher jeder weitere Zugriff auf den gemeinsamen Speicher nicht mehr von einem lokalen Speicherzugriff unterschieden werden. Darüber hinaus müssen lesende und schreibende Zugriffe unterschieden werden können, weil diese verschiedene kausale Abhängigkeiten implizieren.

Es wurden verschiedene Ansätze zur Aufzeichnung von DSM-Zugriffen untersucht. Einer davon basiert auf Unterstützung durch die Hardware. Standard-x86-Computer besitzen Hardware-Debug-Register, die eine kontrollierte Fehlerbehandlung auslösen können, wann immer auf eine Speicheradresse zugegriffen wird. Diese Fähigkeit ist jedoch als Lösung nicht hinreichend, weil lediglich vier derartige Register zur Verfügung stehen, von denen jedes einen Adressbereich von 4 Byte überwachen kann. Es ist klar, dass der DSM viel zu sehr eingeschränkt werden müsste, wenn diese Register genutzt werden sollten.

Ein anderer Ansatz kann von Debugging-Techniken abgeleitet werden, wie sie z. B. der gdb (GNU Debugger) verwendet. Sogenannte „software watchpoints“ können verwendet werden, um Speicherzugriffe zu überwachen. Dies würde jedoch die Ausführung der Prozesse im Einzelschritt-Modus erfordern und die Systemleistung um Faktor 100 senken, was offensichtlich unakzeptabel wäre.

Die einzige befriedigende Lösung besteht in der Modifikation des Compilers. Der Nachteil hinsichtlich Portierbarkeit ist der, dass alle Applikationen mit dem modifizierten Compiler neu kompiliert werden müssen. Für das hier verwendete Experimentalsystem MoDiS ist dies jedoch gar kein Problem, einfach aufgrund des sprachbasierten Gesamtsystemansatzes wie in Abschnitt 6.3 beschrieben. Im Zuge der Analyse der verwendeten Hochsprache (hier INSEL) durch den Compiler können lesende und schreibende Speicherzugriffe zusätzlich untersucht werden. Der Compiler kann dann entsprechenden Zusatzcode für die Ereigniserfassung generieren. Auf diese Weise erfolgt die Analyse statisch, so dass zur Laufzeit nur noch die notwendigen Aufzeichnungsoperationen ausgeführt werden müssen und der Performanzverlust minimiert wird.

8.3.3 Konstruktion des Ereignisverbands

Das Ziel der Implementierung ist, wie gesagt, die Konstruktion des der ausgeführten Berechnung zu Grunde liegenden Ereignisverbands. Diese soll Schritt für Schritt zur Laufzeit erfolgen, so dass der Ereignisverband zu einem gegebenen Zeitpunkt aus Ereignissen be-

steht, die bereits stattgefunden haben, und aus gerichteten Kanten dazwischen. Eine Kante von Ereignis e zu Ereignis e' bedeutet $e \stackrel{r}{\Rightarrow} e'$.

Um die Konsistenz eines solchen teilweise fertig gestellten Graphen G zu erhalten, muss gewährleistet werden, dass nur Ereignisse e' in den Graphen aufgenommen werden, deren Abhängigkeiten bereits erfüllt sind in dem Sinne, dass alle (transitiven) Vorgängerereignisse bereits in den Graphen aufgenommen wurden: $e' \in G$ nur dann, wenn $\forall e | e \stackrel{r}{\Rightarrow} e' : e \in G$. Die entsprechenden Kanten müssen dann natürlich ebenfalls eingefügt werden.

Logische Uhren oder direkte Referenzierung?

Die zentrale Problemstellung bei der Konstruktion einer konsistenten Sicht auf eine nebenläufige verteilte Berechnung ist die Erfassung der kausalen Abhängigkeiten zwischen den Ereignissen. Diese sind durch die in Abschnitt 8.2 erklärten Relationen spezifiziert. Um die kausalen Abhängigkeiten zu erfassen, werden hier zwei verschiedene Ansätze betrachtet: Die Abhängigkeiten können erstens implizit durch den Einsatz logischer Uhren erfasst werden, und zweitens durch explizite Referenzierung kausaler Vorgänger-Ereignisse.

Die Funktionsweise logischer Uhren wurde bereits in Kapitel 3 ausführlich erläutert. Den Ereignissen werden während des Berechnungsablaufs logische Zeitstempel zugeordnet, und der Vergleich dieser Zeitstempel lässt auf die kausalen Abhängigkeiten schließen. Daraus lässt sich, wie bereits in Kapitel 6 gezeigt wurde, ohne weiteres ein Ereignisverband generieren.

Die Alternative zum Einsatz logischer Uhren ist die, während des Berechnungsablaufs schrittweise direkte, d. h. nicht-transitive, Abhängigkeiten in Form von Referenzen auf geeignete Repräsentationen der entsprechenden Ereignisse zu speichern. Wenn wir annehmen, dass jedem Ereignis ein eindeutiger Identifikator zugeordnet ist, dann können wir mit jedem Ereignis auch eine Liste seiner direkten kausalen Vorgänger aufzeichnen. Da wir die Granularität der Ereignisse so wählen, dass ein Ereignis nicht mehr als eine Nachricht versenden oder empfangen darf, kann die Länge einer jeden solchen Liste nie mehr als 2 betragen.

Die eindeutigen Ereignisidentifikatoren können als Kombination aus einem Rechenknoten-Identifikator, einem Prozessidentifikator und einem prozesslokalen Ereigniszähler realisiert werden. Die Größe eines solchen systemweiten Ereignisidentifikators in Bytes kann mit der Größe des Systems variieren, bleibt in einem jeden speziellen System jedoch konstant.

Die Konstruktion des Ereignisverbandes aus derart markierten Ereignissen kann dann ablaufen wie folgt: Für jedes aufgezeichnete Ereignis e wird geprüft, ob alle in seiner Abhängigkeitsliste referenzierten Ereignisse (entweder 1 oder 2) bereits im Graphen vorhanden sind. Wenn ja, kann e in den Graphen aufgenommen werden, ohne die Konsistenz

desselben zu verletzen. Anderenfalls wird rekursiv dieselbe Prüfung für die Listeneinträge durchgeführt.

Wenn logische Uhren und direkte Referenzierung als Möglichkeiten der Kausalitätserfassung verglichen werden sollen, sind dabei zwei Dinge zu beachten: erstens der Platz, der zur Aufzeichnung der Ereignisse und Zusatzinformationen benötigt wird, und zweitens die Eignung für die Konstruktion des Ereignisverbands, wie sie hier angestrebt wird.

Der große Vorteil logischer Uhren ist der Informationsgehalt der Zeitstempel. Diese beinhalten nämlich nicht nur Informationen über die direkten, sondern auch über alle transitiven Abhängigkeiten. Wenn zwei beliebige Ereignisse gegeben sind, kann anhand ihrer Zeitstempel sofort eine umfassende Aussage über ihre Abhängigkeit gemacht werden. Dies ist mit der Methode der direkten Referenzierung nicht möglich. Hier müsste für dieselbe Entscheidung eine Pfadsuche durchgeführt werden, und zwar in beiden Richtungen, um feststellen zu können, ob das eine Ereignis vom anderen aus erreichbar (d. h. abhängig) ist. Im Falle des hier angestrebten Zieles ist dieser (sonst entscheidende) Unterschied jedoch vernachlässigbar, weil eben nicht das Abhängigkeitsverhältnis beliebiger Ereignisse bestimmt werden soll, sondern lediglich der sukzessive Aufbau des Ereignisverbandes im Vordergrund steht. Dafür reicht der nicht-transitive Informationsgehalt direkter Referenzen völlig aus.

Der Speicherplatz, der für die direkten Referenzen benötigt wird, ist darüber hinaus konstant, was für logische Uhren nicht gilt, wenn dynamisch erzeugte Prozesse im Spiel sind. Vergleiche dazu die Abschnitte 4.6 und 5.7. Daher sind, wenn auch nur für den hier vorliegenden Zweck, die direkten Referenzen eindeutig zu bevorzugen. Durch den Verzicht auf die informationelle Überlegenheit (Transitivität) der logischen Uhren, können wir eine höhere Effizienz erzielen, wenn wir nur die nicht-transitiven Abhängigkeiten speichern.

8.3.4 Sammlung der benötigten Informationen

Grundlage der oben beschriebenen Methode zur dynamischen Konstruktion eines Ereignisverbandes sind die während des Systemablaufs auftretenden Ereignisse, markiert mit den Identifikatoren ihrer jeweils direkten kausalen Vorgänger. In den folgenden Unterabschnitten wird beschrieben, wie wie diese Information gesammelt wird, im Hinblick auf die durch die eingeschränkte DSM-Kausalitäts-Relation gegebenen Abhängigkeiten.

Es werden alle Ereignisse aufgezeichnet, die für die Synchronisation der Prozesse untereinander von Bedeutung sind. In MoDiS sind dies die Erzeugung und die Terminierung von Kindprozessen, Nachrichtenaustausch und Zugriffe auf den gemeinsam genutzten verteilten Speicher. Je nachdem, wofür der Ereignisverband nach seiner Konstruktion genutzt werden soll, könnte man die Granularität der Ereignisse natürlich durchaus feiner wählen. Dies

wäre jedoch mit einem (rein quantitativ) höheren Aufwand verbunden, weil entsprechend mehr Ereignisse aufgezeichnet werden müssten. Für die in diesem Kapitel verfolgten Zwecke wäre das nicht gerechtfertigt.

Prozessordnung

Die Prozessordnung wird implizit für alle aufgezeichneten Ereignisse erfasst, weil der systemweit eindeutige Ereignisidentifikator unter anderem einen skalaren Ereigniszähler für den jeweiligen Prozess enthält. Siehe dazu Abschnitt 8.3.3.

Nachrichtenaustausch

Die Erfassung der Abhängigkeiten zwischen Sende- und Empfangsereignissen wird so implementiert, dass der Identifikator des Sendeereignisses zusammen mit der Nachricht an den Empfänger übertragen wird. Der Empfänger registriert dann das damit eindeutig identifizierte Sendeereignis als einen der beiden direkten Vorgänger des entsprechenden Empfangsereignisses. In MoDiS können beide Maßnahmen vom Kommunikator-Modul ergriffen werden.

Es sei an dieser Stelle noch einmal auf die Besonderheiten beim „Nachrichtenaustausch“ in MoDiS hingewiesen. Diese wurden in Abschnitt 6.3.4 erklärt.

DSM

In Abschnitt 8.3.2 wurde erläutert, warum die Generierung zusätzlichen Programmcodes durch den Compiler, zumindest im Falle der hier vorgestellten Implementierung, der einzig sinnvolle Weg zur Erfassung der Speicherzugriffe und ihrer Abhängigkeiten ist. Der INSEL-Compiler gic analysiert jeden Speicherzugriff und erzeugt Code, wenn er einen Zugriff auf den DSM feststellt. In INSEL ist dies beispielsweise bei Variablenzuweisungen, zusammengesetzten Ausdrücken und Funktionsaufrufen der Fall.

Die Erfassung der Lese-/Schreib-Abhängigkeiten, wie sie in Abschnitt 8.2.6 ausführlich behandelt wurden, wird mittels eindeutiger Objektidentifikatoren realisiert. Jedem Objekt im DSM wird ein solcher Identifikator zugewiesen. In der Praxis besteht dieser aus der virtuellen Adresse im gemeinsam genutzten Adressraum und einem skalaren Zähler.

Lese- und Schreibereignisse werden durch die MoDiS-Laufzeitumgebung aufgezeichnet. Das Zugriffsereignis wird mit der Objekt-ID und dem inkrementierten skalaren „Zeitstempel“ des Zugriffs markiert. Durch wechselseitigen Ausschluss wird die Atomarität des Speicherzugriffs und seiner Aufzeichnung sichergestellt.

Die Größe der Speicherobjekte hängt von der Speicherimplementierung und den Sprachkonzepten ab. In MoDiS gibt es keine integrierten Synchronisationsmechanismen für gemeinsam genutzte Speicherobjekte, d. h. Zugriffe auf Speicherobjekte sind unterbrechbar. Aus diesem Grund müssen wir einzelne Speicheradressen als Objekte betrachten. In Systemen mit Zugriffssynchronisation dagegen könnte eine beliebige Anzahl von Speicherzugriffen als ein einziges Zugriffsereignis modelliert werden, solange sie nur atomar ausführbar sind.

Die aufgezeichneten Ereignisse werden dann zu einem Ereignisverband zusammengefügt wie in Abschnitt 8.3.3 erklärt.

8.3.5 Performanz

Das Monitoring eines laufenden Systems führt zwangsläufig zu einem gewissen Performanzverlust, da es schließlich auch auf die Systemressourcen angewiesen ist und somit ein Teil derselben nicht mehr für die eigentliche Nutzleistung zur Verfügung steht. Gerechtfertigt wird eine solche Investition im Allgemeinen durch (mindestens) zwei Argumente.

Zum Ersten kann man entscheiden, das Monitoring nur in einem dedizierten Debugging-Modus zu verwenden, in dem ein Performanzverlust nicht relevant ist. Dabei sollte man sich allerdings bewusst sein, dass sich, wie Schrödinger in der Quantenphysik durch sein berühmtes Gedankenexperiment mit der Katze demonstriert hat [49], ein beobachtetes System durchaus anders verhalten kann als ein unbeobachtetes. Es könnte also sein, dass nicht alle Fehler, die im produktiven System auftreten, sich auch im Debugging-Modus beobachten lassen (oder umgekehrt).

Zum Zweiten gibt es viele Anwendungen, bei denen die Performanz eine wesentlich geringere Rolle spielt als andere Anforderungen wie etwa Verfügbarkeit, Sicherheit, oder Zuverlässigkeit. Ein Systemfehler kann fatale Folgen haben, und zwar nicht nur in Systemen, die gefährliche Anlagen steuern, sondern auch z. B. im ökonomischen Bereich. Die Beobachtung derartiger Systeme kann eine sehr sinnvolle Maßnahme zur Erfüllung von Qualitätsanforderungen sein, für die man einen gewissen Performanzverlust gerne in Kauf nimmt.

Die Auswertung der aufgezeichneten Ereignisse im Sinne der Konstruktion des Ereignisverbandes sowie dessen Nutzung zur Analyse und Visualisierung der Systemausführung kann durch einen externen Prozess erfolgen, und auf einer Maschine, die nicht Teil des produktiven Systems ist. In diesem Fall entsteht dem produktiven Teil des Systems kein Performanzverlust, abgesehen von einer Erhöhung der Netzwerklast. Was als entscheidender Faktor übrig bleibt, ist die für den Performanzverlust durchaus signifikante Sammlung der Informationen wie in Abschnitt 8.3.4 beschrieben.

In den Fällen von Prozesserzeugung und Nachrichtenaustausch kann die Aufzeichnung der Ereignisse und ihrer Identifikatoren vernachlässigt werden, angesichts dessen, dass der geringe Overhead lediglich einen Bruchteil der Zeit darstellt, die für die Erzeugung eines Prozesses oder den Versand einer Nachricht über ein Standard-Netzwerk benötigt wird.

Es bleibt also als einziger wirklich signifikanter Beitrag zum Overhead die Speicherbeobachtung. Jeder DSM-Zugriff benötigt zwischen 4 und 6 zusätzliche Speicherzugriffe sowie eine Inkrementierungs-Instruktion. Eine (sehr großzügige) obere Schranke für den hierdurch erzeugten Performanzverlust kann daher mit Faktor sechs im Vergleich zur normalen Systemausführung beziffert werden. Wie sich in der Praxis herausgestellt hat, ist das Verhalten tatsächlich jedoch wesentlich besser, weil, erstens, nur ein Teil aller Programminstruktionen Speicherzugriffe sind, und zweitens nur ein Teil der Speicherzugriffe wiederum dem DSM gelten. Um den Performanzverlust präziser beschreiben zu können, werden daher Messungen und Vergleiche von repräsentativen Anwendungen jeweils mit und ohne Observierung durchgeführt werden müssen.

8.3.6 Beispiel

Abbildung 8.4 zeigt den Ereignisverband zu einer einfachen MoDiS-Berechnung. Der Graph wurde mittels der in diesem Kapitel beschriebenen Methoden konstruiert. Ereignisse, die Speicherzugriffe repräsentieren, sind weiß dargestellt, Erzeugungs- und Terminierungsereignisse grau und K-Order-Ereignisse schwarz. K-Order sind der in MoDiS eingesetzte Spezialfall von Nachrichtenaustausch. Siehe dazu Abschnitt 6.3.4. Die entsprechenden Ereignisse sind in dem Graphen mit `CORDER_*` bezeichnet.

Die etwas dickeren Pfeile kennzeichnen die Prozessordnung, also die nebenläufigen sequentiellen Ausführungsfäden. Die dünneren Pfeile markieren die Abhängigkeiten durch K-Order, also Nachrichten. Die Abhängigkeiten durch die Schreibordnung sind gepunktet, die Prozesserzeugungs-/-terminierungsabhängigkeiten gestrichelt dargestellt.

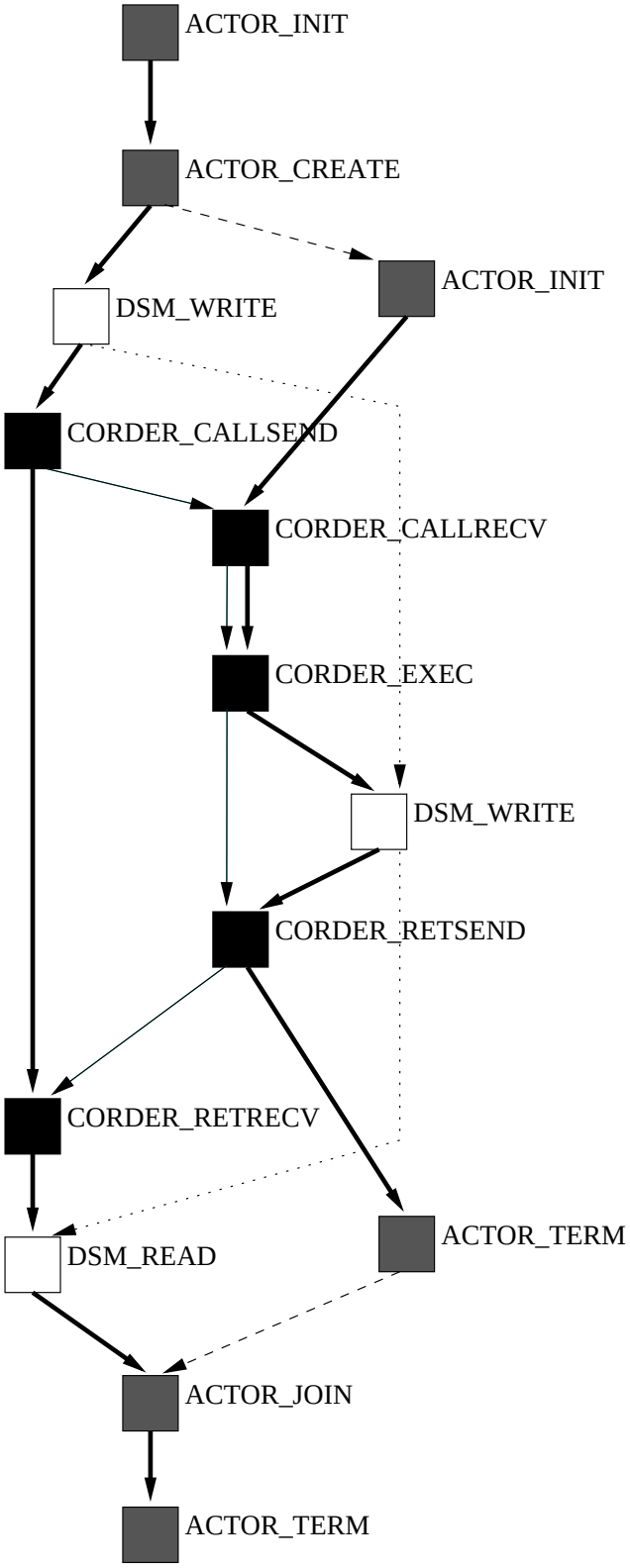

Abbildung 8.4: Der fertig konstruierte Ereignisverband am Ende der Berechnung

Kapitel 9

Zusammenfassung und Ausblick

In diesem Kapitel sollen zunächst die wesentlichen Inhalte der vorliegenden Arbeit noch einmal zusammengefasst und bewertet werden. Im Anschluss daran wird noch auf einige offene Fragestellungen hingewiesen, die sich aus der dieser Arbeit ergeben und eventuell Ansätze zu weiterführender Forschung bilden können.

9.1 Zusammenfassung

Verteilte Systeme bestehen aus mehreren rechenfähigen Komponenten, die miteinander kooperieren. Im Allgemeinen besitzt jede dieser Komponenten ihre eigene lokale Uhr. Diese lokalen Uhren können praktisch nicht perfekt synchronisiert werden. Die Folge davon ist, dass kein systemweit einheitliches Zeitverständnis vorausgesetzt werden kann. Ohne ein solches ist es jedoch nicht möglich, eine aussagekräftige Sicht auf den Gesamtzustand des verteilten Systems zu gewinnen, wie sie etwa für verteiltes Debugging oder Breakpoint-/Rollback-Verfahren benötigt wird. Aus diesem Grund ist es erforderlich, ein neues Zeitkonzept einzuführen, das einem systemweit eindeutigen Verständnis unterliegt. Da sich die eigentlich interessanten Fragestellungen, denen auf der Basis von Systemzuständen nachgegangen werden soll, ohnehin mehr auf kausale Zusammenhänge als auf rein zeitliche Verhältnisse beziehen, liegt es nahe, die Kausalität der Berechnungsschritte als Grundlage zu nehmen. Diese unterliegt einer partiellen Ordnung, mittels derer die Systemzustände wieder in einen aussagekräftigen Kontext gebracht werden können. Dieses Zeitkonzept wird *logische Zeit* genannt.

Die ersten logischen Uhren wurden von Leslie Lamport eingeführt und bestanden in rein skalaren Zählern für die Berechnungsschritte. Für die Erfassung der Kausalität haben sie sich jedoch als unzureichend erwiesen, da sie nur einen Teil der kausalen Abhängigkeiten

zu erfassen in der Lage sind. Weitere Forschung brachte die Vektor-Uhren hervor, bei denen der skalare Zähler durch einen Vektor ersetzt wird, und die häufig eingesetzt werden. Diese Uhren sind in der Lage, die Kausalität vollständig und zufriedenstellend zu erfassen, haben jedoch den gravierenden Nachteil, dass sie eine konstante und von vornherein bekannte Anzahl von Prozessen im System voraussetzen. Diese Anforderung ist für die Mehrzahl moderner Applikationen inakzeptabel, da die dynamische Erzeugung von Prozessen allenthalben Standard ist.

Ausgehend von dieser Ausgangslage wurden in der vorliegenden Arbeit neue logische Uhren entwickelt, um die Leistungsfähigkeit von Vektor-Uhren mit der expliziten Unterstützung dynamischer Prozesserzeugung zu verbinden. Ein erster Schritt führte zu den *dynamischen Vektor-Uhren*, einer um die gewünschte Dynamik ergänzten Erweiterung von Standard-Vektor-Uhren. Die Leistungsfähigkeit von Vektor-Uhren hinsichtlich vollständiger Kausalitätserfassung bleibt bei dynamischen Vektor-Uhren erhalten, und dynamische Prozesserzeugung wird unterstützt. Zur Reduktion der Zeitstempelgröße nach der Terminierung von Prozessen sind jedoch von Zeit zu Zeit explizite Maßnahmen von nicht unerheblichem Aufwand nötig, einschließlich des vorübergehenden Anhaltens einzelner oder gar aller Prozesse.

Um das Ziel einer effizienten logischen Uhr mit impliziter, vollständig „automatischer“ Dynamik zu erreichen, wurden in einem zweiten Schritt die *Baum-Uhren* oder *tree clocks* entwickelt. Diese spiegeln die Struktur des Prozessbaumes wider, und liefern auf diese Weise eine adäquatere, weil natürlichere, Beschreibung der kausalen Zusammenhänge. Die Größe der Zeitstempel wächst und schrumpft implizit und dynamisch mit dem tatsächlichen Prozessaufkommen im System. Die Kausalität wird wie bei Vektor-Uhren vollständig erfasst (in einer bestimmten Klasse von Systemen ermöglichen Baum-Uhren im Gegensatz zu Vektor-Uhren sogar eine uneingeschränkte Lückenerkennung). Bei alledem erweisen sich Baum-Uhren im Vergleich zu Vektor-Uhren insgesamt sogar als effizienter.

Baum-Uhren haben darüberhinaus im Vergleich zu allen anderen erwähnten Typen logischer Uhren eine qualitativ neue Eigenschaft: Da sie sich bei der Erzeugung eines neuen Prozesses den alten Zählerstand merken, während sie auf einer neuen Ebene weiterzählen, besitzen sie ein gewisses „Gedächtnis“. Diese Eigenschaft konnte in der vorliegenden Arbeit dafür genutzt werden, eine Abstraktion von Teilen einer Sicht auf das System vollständig transparent für die logische Zeit zu halten. Dies kann man sich so vorstellen, dass man mittels logischer Uhren eine Sicht auf die möglicherweise zahlreichen Prozesse und Ereignisse (Rechenschritte) konstruiert hat, und nun Teile dieser Sicht je nach Bedarf zoomartig ein- und ausblenden möchte, etwa wie Teile eines Verzeichnisbaumes im Dateimanager. Dabei sollen die Zeitstempel der nicht ausgeblendeten Teile möglichst nicht in ihrer Konsistenz und Aussagekraft beeinträchtigt werden. Als Anwendung könnte man sich etwa die gesteigerte Usability eines verteilten Debuggers für umfangreiche Applikationen vorstellen. Für eine derartige Abstraktion wurde in dieser Arbeit ein Verfahren entwickelt. Es zeigt

sich, dass eine vollständige Konsistenzerhaltung der Zeitstempel bei Abstraktion, auch im Hinblick auf die Lückenerkennung, nur mit Baum-Uhren möglich ist (eben aufgrund ihrer „Gedächtnis“-Eigenschaft).

Angesichts all dieser Vorteile von Baum-Uhren, bei lediglich einer leicht erfüllbaren Annahme als Nachteil, erscheint die Vermutung gerechtfertigt, dass Baum-Uhren (oder möglicherweise auch verbesserte Varianten davon) Vektor-Uhren künftig in den meisten verteilten Systemen ablösen werden. Dass die zunächst theoretisch entwickelten Konzepte der Baum-Uhren auch praktisch umsetzbar sind und die erwarteten Ergebnisse liefern, wurde im Rahmen der vorliegenden Arbeit durch eine reale Implementierung unter Beweis gestellt.

Zusätzlich zu den bereits genannten Inhalten wurde in dieser Arbeit auch die Konstruktion konsistenter Sichten in einer Klasse von Systemen betrachtet, die den Prozessen einen gemeinsam nutzbaren verteilten Speicher (DSM) zur Verfügung stellen. Die diesbezügliche Problematik wurde analysiert, und auf dieser Basis wurden nützliche Definitionen erstellt sowie Konzepte für eine effiziente Erfassung der Kausalität in DSM-Systemen erarbeitet. Die Funktionsfähigkeit dieser Konzepte wurde ebenfalls durch eine praktische Implementierung nachgewiesen.

9.2 Ausblick

Wenn man davon ausgeht, dass sich die aktuellen Trends in Zukunft fortsetzen (und alles spricht dafür, dass diese Annahme sinnvoll ist), dann ist weiterhin mit wachsender Popularität kooperativer verteilter Systeme und Rechenverfahren zu rechnen. Auch Umfang und Komplexität der Systeme und Applikationen wird weiterhin wachsen. Vor diesem Hintergrund ist die Thematik der vorliegenden Arbeit besonders bedeutsam. Um größeren verteilten Anwendungen mit immer mehr Prozessen und komplexerem Kooperationsverhalten Herr werden zu können, werden effiziente Verfahren zur Gewinnung und Auswertung aussagekräftiger Sichten auf derartige Systeme benötigt. Solche Konzepte wurden in der vorliegenden Arbeit entwickelt. Wie jeder Schritt nach vorn wirft jedoch auch dieser unweigerlich neue Fragen auf. Auf einige davon soll an dieser Stelle noch explizit hingewiesen werden.

Gerade angesichts der zunehmenden Bedeutung verteilter Systeme stellt sich die Frage nach ihrer Zuverlässigkeit. Im Zusammenhang mit den in dieser Arbeit entwickelten Baum-Uhren sei auf ein mögliches Robustheitsproblem hingewiesen, das unter Umständen beim praktischen Einsatz von Baum-Uhren in fehlerbehafteten Systemen auftreten kann. Die Funktionsweise der Baum-Uhren basiert auf einem Informationsfluss, der bei der Terminierung eines Prozesses vom terminierten Prozess zu dessen Vaterprozess stattfindet. Dieser

setzt jedoch im Allgemeinen eine planmäßige Terminierung voraus. Wenn der Kindprozess abstürzen sollte, wäre der benötigte Informationsfluss nicht mehr gegeben und der Vaterprozess könnte seine logische Uhr nicht entsprechend aktualisieren, was die logische Zeit kompromittieren würde. Mit dem Ziel allgemeiner Fehlertoleranz vor Augen sollte man sich daher noch die Frage stellen, was getan werden kann, um einen solchen Fall möglichst glimpflich abzufangen. Eventuell wäre es möglich, hier eine Management-Komponente einspringen zu lassen, die über die Ereignisse und damit auch die Zeitstempel des abgestürzten Prozesses informiert ist und den letzten bekannten Zeitstempel dann dem Vaterprozess zur Verfügung stellen kann. Machbarkeits- und Effizienzüberlegungen hierzu stehen noch aus, zumal dieses Problem vermutlich ohnehin nur vor dem Hintergrund einer gegebenen spezifischen Systemumgebung betrachtet und gelöst werden kann.

Ein weiterer Punkt wurde bereits in Abschnitt 5.9 angesprochen und betrifft die qualitativ neuen Eigenschaften von Baum-Uhren gegenüber skalaren, Vektor- und dynamischen Vektor-Uhren: ihr „Gedächtnis“. Die Verwendbarkeit dieser Eigenschaft konnte im Rahmen der vorliegenden Arbeit leider nicht mehr erschöpfend untersucht werden. Die Frage, die sich daher noch stellt, ist: Welche Vorteile bringt diese neue Qualität mit sich, bzw. wie lässt sie sich sinnvoll ausnutzen? Eine mögliche Anwendung wurde mit der Abstraktion in Kapitel 7 bereits behandelt, aber es scheint nahe liegend, dass sich das „Gedächtnis“ der Baum-Uhren auch für viele weitere Anwendungsbereiche mit unterschiedlichsten Zielsetzungen und Anforderungen als vorteilhaft erweisen kann.

Was ebenfalls noch aussteht, ist eine Implementierung des in Kapitel 7 entwickelten Verfahrens zur Abstraktion einmal konstruierter konsistenter Sichten. Nahe liegend wäre ein Monitorprogramm, das mittels logischer Uhren eine Sicht auf den Berechnungsablauf erstellt (so wie mit der in Kapitel 6 besprochenen Implementierung von Baum-Uhren bereits realisiert), visualisiert und dann ein strukturiertes Ein- und Ausblenden von Teilen des visualisierten Graphen erlaubt, ohne die in den (im Graphen ebenfalls eingetragenen) Zeitstempeln enthaltene Kausalitätsinformation zu verfälschen.

Eine andere hochinteressante Frage, die sich im Zuge dieser Arbeit mehrmals anbot, aber leider nicht mehr verfolgt werden konnte, betrifft eine mögliche Arithmetik über logischen Zeitstempeln. Lassen sich, analog zu Realzeitstempeln, Differenzen zwischen logischen Zeitstempeln bilden? Könnte man auf diese Weise eine Art kausalen Abstand definieren? Wofür ließe sich ein solcher verwenden? Vielleicht für Platzierungsentscheidungen oder sonstige Heuristiken im automatisierten Management?

Anhang A

Code-Listings

Um den Lesefluss in Kapitel 6 nicht zu beeinträchtigen, wurden die doch recht umfangreichen Code-Listings in diesen Anhang ausgelagert. Sie geben die entscheidenden Funktionen aus der Referenzimplementierung von Baum-Uhren an. Die verwendete Programmiersprache ist durchweg C.

```
/* global process ID */
struct gpid_t {
    ip_addr_t hid;
    size_t    pid;
    int       status;
};


/* clock tree node */
typedef
struct tc_node {
    int             value;     /* time value of this node */
    struct gpid_t   label;     /* label (process ID) of this node */
    int             iscounter; /* flag indicating whether this node is the current counter */
    struct tc_node* lchild;    /* pointer to left child node */
    struct tc_node* rchild;    /* pointer to right child node */
    struct tc_node* father;    /* pointer to father node */
} tc_node_t;


/* clock tree */
typedef
struct tc {
    tc_node_t* root;    /* pointer to root of clock tree */
    tc_node_t* counter; /* pointer to currently counting node */
} tc_t;
```

Listing A.1: Datenstrukturen für Baum-Uhren

```
/* timestamp (serialized clock tree) */
typedef
struct tc_timestamp {
    tc_node_t* ts;   /* pointer to an array of nodes with relative cross-references */
    int        size; /* size of the array in bytes */
} tc_timestamp_t;


/* comparison result constants */
typedef
enum {
    TC_NULL,
    TC_NOTMATCHING,
    TC_EQUAL,
    TC_GREATER,
    TC_SMALLER
} tc_comp_t;


/* event type constants */
typedef
enum {
    TC_INTERNAL,
    TC_CREATE,
    TC_INIT,
    TC_TERM,
    TC_JOIN,
    TC_SEND,
    TC_RECEIVE
} tc_event_t;


/* struct holding data recorded for an event */
typedef
struct listelem {
    int              id;    /* id of associated event (needed for GML) */
    tc_event_t       event; /* type of associated event */
    tc_t             tree;  /* timestamp tree */
    tc_t             pre1;  /* first predecessor tree on case of join event */
    tc_t             pre2;  /* second predecessor tree on case of join event */
    struct listelem* suc1;  /* pointer to one of no more than two recorded events which are
                               direct causal successors */
    struct listelem* suc2;  /* pointer to one of no more than two recorded events which are
                               direct causal successors */
    struct listelem* next;  /* pointer to next recorded timestamp */
} tc_eventdata_t;


typedef struct manager
{
    /* loads of meta data concerning the actor and used for management left out here */

    /* Tree clock entries. */
    tc_t          treeclock;   /* The clock tree of the actor. */
    struct gpid_t lastcreated; /* The gpid of the child actor that was last created.
                                  This info is needed when appending nodes to the clock tree on
                                  the next event. The status member is used as a flag here,
                                  which is set to 1 on the create event and back to 0 on the
                                  next event when the info is no longer needed. */
} manager_t;
```

Listing A.2: Datenstrukturen für Baum-Uhren (Fortsetzung)

```
void tc_tree_update_init(struct manager* m)
/* updates a clock tree on an init event */
{
    tc_node_t* newnode;

    /* copy clock tree of daddy process */
    tc_tree_copy(m->daddy->treeclock, &(m->treeclock), m);

    /* add new node as left child of current counter */
    newnode = (tc_node_t*)ISE_malloc(m, sizeof(tc_node_t));
    newnode->lchild = newnode->rchild = NULL;
    newnode->father = m->treeclock.counter;
    newnode->label = m->daddy->gpid;
    newnode->value = 0;
    m->treeclock.counter->lchild = newnode;

    /* add new node as right child of current counter */
    newnode = (tc_node_t*)ISE_malloc(m, sizeof(tc_node_t));
    newnode->lchild = newnode->rchild = NULL;
    newnode->father = m->treeclock.counter;
    newnode->label = m->gpid;
    newnode->value = 1;
    m->treeclock.counter->rchild = newnode;

    /* update counter */
    m->treeclock.counter->value++;
    m->treeclock.counter->iscounter = 0;
    m->treeclock.counter = newnode;
    newnode->iscounter = 1;
}
```

Listing A.3: Baum-Uhren-Aktualisierung für Initialisierungsereignisse

```
void tc_tree_update_create(struct manager* m)
/* adds the new nodes to a clock tree on the next after a create event */
{
    tc_node_t* newnode;

    /* add new node as right child of current counter */
    newnode = (tc_node_t*)ISE_malloc(m, sizeof(tc_node_t));
    newnode->lchild = newnode->rchild = NULL;
    newnode->father = m->treeclock.counter;
    newnode->label = m->lastcreated;
    newnode->value = 0;
    m->treeclock.counter->rchild = newnode;

    /* add new node as left child of current counter */
    newnode = (tc_node_t*)ISE_malloc(m, sizeof(tc_node_t));
    newnode->lchild = newnode->rchild = NULL;
    newnode->father = m->treeclock.counter;
    newnode->label = m->gpid;
    newnode->value = 1;
    m->treeclock.counter->lchild = newnode;

    /* set counter to new left child*/
    m->treeclock.counter->iscounter = 0;
    m->treeclock.counter = newnode;
    newnode->iscounter = 1;

    /* now the create event has been taken care of */
    m->lastcreated.status = 0;
}


void tc_tree_update_internal(struct manager* m)
/* does the standard part in updating the given manager's clock tree on any but init events */
{
    /* increment counter */
    m->treeclock.counter->value++;

    /* if last event was a create, new nodes have to be added */
    if (m->lastcreated.status) tc_tree_update_create(m);
}
```

Listing A.4: Baum-Uhren-Aktualisierung für interne und Erzeugungsereignisse

```
tc_node_t* tc_receive_recursive(const tc_node_t* sourcenode, tc_node_t* targetnode,
                                tc_node_t* fathernode, const manager_t* m)
/* recursive function to do all the dirty work for tc_tree_update_receive */
{
    if (!sourcenode) return targetnode; /* make sure source node is present */

    switch (tc_node_compare(sourcenode, targetnode)) { /* compare source and target node */

        case TC_NULL: /* target node is missing, so copy it */
            if (tc_gpid_equal(fathernode->label, m->gpid)) /* with this exception */
                return NULL;
            targetnode = (tc_node_t*)ISE_malloc(m, sizeof(tc_node_t));
            targetnode->value = sourcenode->value;
            targetnode->label = sourcenode->label;
            targetnode->iscounter = 0;
            targetnode->lchild = targetnode->rchild = NULL;
            targetnode->father = fathernode;
            break; /* continue with children */

        case TC_NOTMATCHING: /* the node is already present with a different label */
            return targetnode; /* no place to copy it to, so skip this whole subtree */

        case TC_GREATER: /* source value is greater, so assign it to target */
            targetnode->value = sourcenode->value;
            break; /* continue with children */

        case TC_SMALLER:
        case TC_EQUAL: /* nothing to do for this node, but continue with its children */
    }

    /* recursively process the children */
    targetnode->lchild =
        tc_receive_recursive(sourcenode->lchild, targetnode->lchild, targetnode, m);
    targetnode->rchild =
        tc_receive_recursive(sourcenode->rchild, targetnode->rchild, targetnode, m);

    return targetnode;
}


void tc_tree_update_receive(struct manager* m, const tc_timestamp_t incoming)
/* updates the clock tree of the given manager on a receive event */
{
    tc_t incomingtree;

    /* do standard incrementing for internal events */
    tc_tree_update_internal(m);

    /* reconstruct tree from timestamp */
    incomingtree = tc_tree_from_timestamp(m, incoming);

    /* max values and/or copy nodes from incoming tree */
    tc_receive_recursive(incomingtree.root, m->treeclock.root, NULL, m);

    /* free reconstructed tree */
    tc_free(m, incomingtree.root);
}
```

Listing A.5: Baum-Uhren-Aktualisierung für Empfangsereignisse

```
tc_node_t* tc_serialize_node(const tc_node_t* node, tc_node_t* pos)
/* recursive function to do all the dirty work for tc_timestamp_from_tree */
{
    tc_node_t *cpos, *nextfreepos = pos+1;

    *pos = *node; /* copy node struct to pos */

    if (node->lchild) { /* do lchild */
        cpos = nextfreepos; /* remember child position for reference conversion */
        nextfreepos = tc_serialize_node(node->lchild, cpos); /* save into next free position */
        (*pos).lchild = (tc_node_t*)(cpos - pos); /* convert to relative pointer (offset) */
    }

    if (node->rchild) { /* do rchild */
        cpos = nextfreepos; /* remember child position for reference conversion */
        nextfreepos = tc_serialize_node(node->rchild, cpos); /* save into next free position */
        (*pos).rchild = (tc_node_t*)(cpos - pos); /* convert to relative pointer (offset) */
    }

    return nextfreepos; /* return next free position */
}


tc_timestamp_t tc_timestamp_from_tree(const manager_t* m, const tc_t tree)
/* Serializes a clock (sub)tree into a sendable/saveable timestamp. */
{
    tc_timestamp_t stamp;

    stamp.size = tc_node_count(tree.root) * sizeof(tc_node_t); /* calculate size of timestamp */
    stamp.ts = (tc_node_t*)ISE_malloc(m, stamp.size); /* allocate timestamp */

    /* serialize tree into timestamp */
    if ((int)tc_serialize_node(tree.root, stamp.ts) != (int)(stamp.ts) + stamp.size) {
        puts("TREE CLOCKS: Serialization does not match allocated timestamp size!");
    }

    return stamp; /* return timestamp */
}
```

Listing A.6: Serialisierung von Bäumen

```
tc_node_t* tc_deserialize_node
(const tc_node_t* pos, tc_node_t* father, const manager_t* m, tc_node_t** counter)
/* recursive function to do all the dirty work for tc_tree_from_timestamp */
{
    tc_node_t* node = (tc_node_t*)ISE_malloc(m, sizeof(tc_node_t)); /* allocate node */

    *node = *pos; /* copy node struct from timestamp */
    node->father = father; /* update father reference */
    if (node->iscounter) *counter = node; /* set pointer to counter */

    /* recursively deserialize children, based on the relative cross-references */
    if (pos->lchild)
        node->lchild = tc_deserialize_node(pos + (int)(pos->lchild), node, m, counter);
    if (pos->rchild)
        node->rchild = tc_deserialize_node(pos + (int)(pos->rchild), node, m, counter);

    return node;
}


tc_t tc_tree_from_timestamp(const manager_t* m, const tc_timestamp_t stamp)
/* reconstructs a clock tree from a timestamp serialization and returns a pointer to its root */
{
    tc_t tree;

    tree.root = tc_deserialize_node(stamp.ts, NULL, m, &(tree.counter)); /* build up the tree */

    if (stamp.size != (tc_node_count(tree.root) * sizeof(tc_node_t)))
        puts("TREE CLOCKS: Reconstructed tree does not match timestamp size!");

    return tree;
}
```

Listing A.7: Deserialisierung von Bäumen

```
void tc_log(const tc_event_t eventtype, const tc_t tree, const tc_t pre1, const tc_t pre2,
            const struct manager* m, const int startactor)
/* Logs an event with its time stamp to TC_LOGFILE.
 * On a join event both predecessor trees must be provided
 * (on any other events pre1 and pre2 are ignored). */
{
    int              fd;
    tc_timestamp_t   ts;

    if (tree.counter->label.pid < startactor) return; /* actors 1-3 are for management,
                                                         4th is SYSTEM */

    fd = open(TC_LOGFILE, O_WRONLY | O_CREAT | O_APPEND, S_IWRITE | S_IREAD);
    if (fd == -1) {
        printf("TREE CLOCKS: Could not open log file %s!", TC_LOGFILE);
        return;
    }

    ts = tc_timestamp_from_tree(m, tree); /* generate timestamp */
    write(fd, &eventtype, sizeof(tc_event_t)); /* write integer representing the event type */
    write(fd, &(ts.size), sizeof(int)); /* write size of timestamp data */
    write(fd, ts.ts, ts.size); /* write timestamp data */
    ISE_free(m, ts.ts); /* free timestamp data */

    if (eventtype == TC_JOIN) { /* pre1 and pre2 have also to be saved */

        ts = tc_timestamp_from_tree(m, pre1); /* generate timestamp */
        write(fd, &(ts.size), sizeof(int)); /* write size of timestamp data */
        write(fd, ts.ts, ts.size); /* write timestamp data */
        ISE_free(m, ts.ts); /* free timestamp data */

        ts = tc_timestamp_from_tree(m, pre2); /* generate timestamp */
        write(fd, &(ts.size), sizeof(int)); /* write size of timestamp data */
        write(fd, ts.ts, ts.size); /* write timestamp data */
        ISE_free(m, ts.ts); /* free timestamp data */
    }

    close(fd);
}
```

Listing A.8: Schreiben der Protokolldatei

```
tc_eventdata_t* readlog(char* logfile)
/* reads all logged events from logfile,
 * stores them in a linked list, and returns a pointer to the first element */
{
    int            count = 0, fd = open(logfile, O_RDONLY);
    tc_event_t     ev;
    tc_eventdata_t *first = NULL, *p;
    tc_timestamp_t ts;

    if (fd == -1) {
        puts("Could not open log file!");
        return NULL;
    }
    puts("Log file successfully opened");

    while (read(fd, &ev, sizeof(tc_event_t))) {

        p = (tc_eventdata_t*)malloc(sizeof(tc_eventdata_t));
        p->id = ++count;
        p->next = first;
        first = p;
        p->event = ev;
        read(fd, &(ts.size), sizeof(int));
        ts.ts = (tc_node_t*)malloc(ts.size);
        read(fd, ts.ts, ts.size);
        p->tree = tc_tree_from_timestamp(ts);

        free(ts.ts);
        p->suc1 = p->suc2 = NULL; /* needed for event lattice generation */

        if (ev == TC_JOIN) { /* the two predecessor trees have also to be read */

            read(fd, &(ts.size), sizeof(int));
            ts.ts = (tc_node_t*)malloc(ts.size);
            read(fd, ts.ts, ts.size);
            p->pre1 = tc_tree_from_timestamp(ts);
            free(ts.ts);

            read(fd, &(ts.size), sizeof(int));
            ts.ts = (tc_node_t*)malloc(ts.size);
            read(fd, ts.ts, ts.size);
            p->pre2 = tc_tree_from_timestamp(ts);
            free(ts.ts);
        }
        else p->pre1.root = p->pre2.root = p->pre1.counter = p->pre2.counter = NULL;
    }
    close(fd);
    printf("%d events read, log file closed\n", count);
    return first; /* last event logged is now the first in the list */
}
```

Listing A.9: Einlesen der Protokolldatei

```
void make_dep(tc_eventdata_t* first)
/* determines and inserts dependencies in list of recorded events */
{
    tc_eventdata_t* this = first;
    tc_eventdata_t* p    = first->next;
    int gap;

    while (this) {
        while (p) {
            if (tc_tree_compare(this->tree, p->tree) == TC_SMALLER) {
                /* p is a transitive successor event of this */

                /* now check for gap as only direct dependencies will be marked */
                if (p->event == TC_JOIN)
                    /* use gap detection for join events */
                    gap = tc_gap_join(p->pre1, p->pre2, this->tree);

                else {
                    if ((p->event == TC_INIT) || (p->event == TC_RECEIVE) ||
                        (tc_gpid_equal(this->tree.counter->label, p->tree.counter->label)))
                        /* greater event is receive/init or both events are of the same process,
                         * so gap detection is possible */
                        /* use gap detection for non-join events */
                        gap = tc_gap(this->tree, p->tree, (p->event == TC_INIT));

                    else
                        /* gap detection impossible because "i!=k" */
                        gap = 1; /* but there has to be a gap because
                                    the receive/init case has already been handled above */
                }

                if (!gap) { /* p is direct successor to this */
                    /* so mark direct dependency */
                    if (this->suc1){ /* this is already the second direct successor */
                        this->suc2 = p;
                        break; /* there can be no more direct successor, so break inner loop */
                    }
                    else this->suc1 = p;
                }
            }
            p = p->next;
            if (p == this) p = p->next;
        }
        this = this->next;
        p = first;
    }
}
```

Listing A.10: Rekonstruktion des Ereignisverbandes aus Baum-Uhren

```
int tc_gpid_equal(const struct gpid_t gpid1, const struct gpid_t gpid2)
/* determines whether two given gpids are equal */
{
    return (
        (gpid1.hid == gpid2.hid) &&
        (gpid1.pid == gpid2.pid)
    );
}


tc_comp_t tc_node_compare(const tc_node_t* node1, const tc_node_t* node2)
/* compares two tree nodes */
{
    if (node1 == NULL) return TC_NULL;
    if (node2 == NULL) return TC_NULL;
    if (!tc_gpid_equal(node1->label, node2->label)) return TC_NOTMATCHING;
    if (node1->value > node2->value) return TC_GREATER;
    if (node1->value < node2->value) return TC_SMALLER;
    return TC_EQUAL;
}
```

Listing A.11: Vergleich von Knoten

```
int tc_tree_identical(tc_node_t* node1, tc_node_t* node2)
/* determines whether two given (sub)trees are identical */
{
    if (node1 == node2) return 1;

    if (node1 == NULL) return 0;
    if (node2 == NULL) return 0;

    if (node1->value != node2->value) return 0;
    if (!tc_gpid_equal(node1->label, node2->label)) return 0;

    return (
        tc_tree_identical(node1->lchild, node2->lchild) &&
        tc_tree_identical(node1->rchild, node2->rchild)
    );
}


tc_comp_t tc_tree_compare_recursive(const tc_node_t* node1, const tc_node_t* node2)
/* recursive function to do all the dirty work for tc_tree_compare */
{
    tc_comp_t ret = tc_node_compare(node1, node2);

    switch (ret) {

        case TC_NULL:
        case TC_NOTMATCHING:
            return TC_GREATER;

        case TC_SMALLER:
            return TC_SMALLER;

        case TC_EQUAL:
            ret = TC_GREATER;
        case TC_GREATER:
    }

    if (node1->lchild) {
        ret = tc_tree_compare_recursive(node1->lchild, node2->lchild);
        if (ret == TC_SMALLER) return ret;
    }

    if (node1->rchild)
        ret = tc_tree_compare_recursive(node1->rchild, node2->rchild);

    return ret;
}


tc_comp_t tc_tree_compare(const tc_t tree1, const tc_t tree2)
/* compares two clock trees (assuming they are not entirely identical) */
{
    if (tc_tree_compare_recursive(tree1.root, tree2.root) == TC_GREATER) return TC_GREATER;
    if (tc_tree_compare_recursive(tree2.root, tree1.root) == TC_GREATER) return TC_SMALLER;
    return TC_EQUAL;
}
```

Listing A.12: Vergleich von Bäumen

```
tc_node_t* tc_find_matching_recursive(const tc_node_t* node1, tc_node_t* node2, tc_node_t* node)
/* recursive function to do all the dirty work for tc_find_matching */
{
    tc_node_t* ret;

    if (!node) return NULL;

    if (node1 == node) {
        if (!node2) return NULL;
        if (tc_gpid_equal(node1->label, node2->label)) return node2;
        return NULL;
    }

    if (!node1) return node;
    if (!node2) return node;

    ret = tc_find_matching_recursive(node1->lchild, node2->lchild, node);
    if (ret != node) return ret;

    return tc_find_matching_recursive(node1->rchild, node2->rchild, node);
}


tc_node_t* tc_find_matching(const tc_t tree1, const tc_t tree2, tc_node_t* node)
/* Finds the node in tree2 that matches the given node which must be in tree1.
 * Returns NULL if there is no matching node. */
{
    tc_node_t* ret;

    ret = tc_find_matching_recursive(tree1.root, tree2.root, node);
    if (ret == node) return NULL;
    return ret;
}
```

Listing A.13: Bestimmung passender Knoten

```
int tc_gap(const tc_t treei, const tc_t treej, const int init)
/* Determines whether there is a gap between two given clock trees.
 * treej must be greater than treei. */
{
    tc_node_t *node, *match;
    int       val;

    if ((tc_gpid_equal(treei.counter->label, treej.counter->label)) || (init)) {
        /* timestamps belong to the same process (i==j) */
        node = treej.counter;
        while (node) {
            match = tc_find_matching(treej, treei, node);
            if (match) val = match->value;
            else val = 0;
            if (node->value - val > 1) return 1; /* gap */
            node = node->father;
        }
        return 0; /* no gap */
    }

    /* timestamps do not belong to the same process (i!=j) */
    node = treei.counter;
    while (node) {
        match = tc_find_matching(treei, treej, node);
        if (!match) {
            puts("TREE CLOCKS: Error! No matching node in gap detection!");
            return 2; /* error */
        }
        if (node->value != match->value) return 1; /* gap */
        node = node->father;
    }
    return 0; /* no gap */
}


int tc_gap_join(const tc_t tree_join_1, const tc_t tree_join_2, const tc_t tree)
/* Determines whether there is a gap between a join event and another event the tree of which
 * is smaller. The two trees of the join event's predecessors must be given along with
 * the tree in question. */
{
    if (tc_tree_identical(tree.root, tree_join_1.root)) return 0;
    if (tc_tree_identical(tree.root, tree_join_2.root)) return 0;
    return 1;
}
```

Listing A.14: Lückenerkennung

```
void make_graph(tc_eventdata_t* first, char* gmlfile)
/* generates GML graph file */
{
    char*           label;
    tc_eventdata_t* this = first;
    FILE*           fp = fopen(gmlfile, "w");

    if (!fp) {
        puts("Could not open GML file!");
        return;
    }
    puts("GML file successfully opened");

    fprintf(fp, "graph [\n\nid 1\n");
    fprintf(fp, "label \"Event lattice generated from tree clock time stamps\"\n\n");

    while (this) {
        switch (this->event) {
            case TC_INTERNAL:
                label = "intern";
                break;
            case TC_CREATE:
                label = "create";
                break;
            case TC_INIT:
                label = "init";
                break;
            case TC_TERM:
                label = "term";
                break;
            case TC_JOIN:
                label = "join";
                break;
            case TC_SEND:
                label = "send";
                break;
            case TC_RECEIVE:
                label = "receive";
                break;
            default:
                label = "ERROR!";
        }
        fprintf(fp, "node [\nid %d\nlabel \"%s\"\n]\n\n", this->id, label);
        if (this->suc1)
            fprintf(fp, "edge [\nsource %d\ntarget %d\n]\n\n", this->id, this->suc1->id);
        if (this->suc2)
            fprintf(fp, "edge [\nsource %d\ntarget %d\n]\n\n", this->id, this->suc2->id);
        this = this->next;
    }

    fprintf(fp, "]");

    fclose(fp);
    puts("GML file written and closed");
}
```

Listing A.15: Spezifikation des Graphen in GML

Abbildungsverzeichnis

Verzeichnis der Code-Listings

Literaturverzeichnis

[1] Leslie Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
In: *Communications of the ACM*, 21(7), S. 558–565, 1978.

[2] Reinhard Schwarz, Friedemann Mattern.
Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail.
In: *Distributed Computing*, 7(3), S. 149–174, 1994.

[3] Tobias Landes.
Tree Clocks: An Efficient and Entirely Dynamic Logical Time System.
In: H. Burkhart (Hrsg.), *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks*, S. 375–380, Innsbruck, Österreich, ACTA Press, Februar 2007.

[4] Tobias Landes.
Dynamic Vector Clocks for Consistent Ordering of Events in Dynamic Distributed Applications.
In: Hamid R. Arabnia (Hrsg.), *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Bd. I, S. 31–37, Las Vegas, Nevada, USA, Juni 2006.

[5] Özalp Babaoğlu, Keith Marzullo.
Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms.
In: Sape Mullender (Hrsg.), *Distributed Systems*, 2nd edition, S. 97–145, Addison Wesley, 1993.

[6] Tobias Landes.
Preserving Logical Time while Abstracting from Distributed Computations.
In: Hamid R. Arabnia (Hrsg.), *Proceedings of the International Conference on Parallel*

and Distributed Processing Techniques and Applications, Bd. II, S. 543–549, Las Vegas, Nevada, USA, Juni 2007.

[7] Jörg Preissinger, Tobias Landes.
Fundamentals for Consistent Event Ordering in Distributed Shared Memory Systems.
In: Hamid R. Arabnia (Hrsg.), *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Bd. III, S. 890–896, Las Vegas, Nevada, USA, Juni 2005.

[8] Tobias Landes, Jörg Preissinger.
Realizing Consistent Event Ordering in Distributed Shared Memory Systems.
In: Hamid R. Arabnia (Hrsg.), *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, Bd. I, S. 10–16, Las Vegas, Nevada, USA, Juni 2006.

[9] Friedemann Mattern.
Virtual Time and Global States of Distributed Systems.
In: M. Cosnard et al. (Hrsg.), *Proceedings of the Workshop on Parallel and Distributed Algorithms*, Elsevier Science Publishers B.V., North-Holland, S. 215–226, 1989.

[10] Colin Fidge.
Timestamps in Message-Passing Systems that Preserve the Partial Ordering.
In: *Proceedings of the 11th Australian Computer Science Conference*, S. 55–66, 1988.

[11] Colin Fidge.
Logical Time in Distributed Computer Systems.
In: *Computer*, 24(8), S. 28–33, 1991.

[12] P.P. Spies, C. Eckert, M. Lange, D. Marek, R. Radermacher, F. Weimer, H.-M. Windisch.
Sprachkonzepte zur Konstruktion verteilter Systeme.
In: *SFB-Bericht 342/09/96 A TUM-I9618*, Technischer Bericht, Technische Universität München, Deutschland, 1996.

[13] C. Eckert, M. Pizka.
Improving Resource Management in Distributed Systems Using Language-Level Structuring Concepts.
In: *Journal of Supercomputing*, 13(1), S. 33–55, 1999.

[14] Peter P. Spies.
Ereignisverbände – Ein flexibles Beschreibungsinstrumentarium für die Entwicklung verteilter Systeme.
In: *FBT'98-Fachgespräch*, Cottbus, Deutschland, 1998.

[15] Tobias Landes.
Anwendungsangepasste Platzierungsentscheidung in verteilten Systemen.
Diplomarbeit, Technische Universität München, Deutschland, Juli 2002.

[16] Christian Rehn.
Überwindung der Stellengrenzen in kooperativen verteilten Systemen.
Dissertation, Technische Universität München, Deutschland, März 2004.

[17] L. Gunaseelan, R.J. LeBlanc, Jr.
Event Ordering in a Shared Memory Distributed System.
In: *Proceedings of the 13th International Conference on Distributed Computing Systems*, S. 256–263, 1993.

[18] Igor A. Zhuklinets, D. A. Khotimsky.
Logical Time in Distributed Software Systems.
In: *Programming and Computing Software*, 28(3), Plenum Press, S. 174–184, 2002.

[19] Shaz Qadeer.
Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking.
In: *IEEE Trans. Parallel Distrib. Syst.*, Bd. 14, Nr. 8, S. 730–741, 2003.

[20] Leslie Lamport.
How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs.
In: *IEEE Transactions on Computers*, Bd. 28, Nr. 9, S. 690–691, 1979.

[21] Sebastian Haas.
Erfassung konsistenter Sichten von verteilten, nebenläufigen Systemen.
Diplomarbeit, Technische Universität München, Deutschland, 2005.

[22] Markus Pizka.
Design and Implementation of the GNU INSEL Compiler gic.
In: *SFB-Bericht 342/09/97 A TUM-I9713*, Technischer Bericht, Technische Universität München, Deutschland, 1997.

[23] Ralph Radermacher, Frank Weimer.
INSEL Syntax-Bericht.
In: *SFB-Bericht 342/08/96 A TUM-I9617*, Technischer Bericht, Technische Universität München, Deutschland, März 1996.

[24] Anurag Agarwal, Vijay K. Garg.
Efficient Dependency Tracking for Relevant Events in Shared-Memory Systems.

In: *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing*, S. 19–28, Las Vegas, Nevada, USA, 2005.

[25] Phillip B. Gibbons, Ephraim Korach.
Testing Shared Memories.
In: *SIAM J. Comput.*, Bd. 26, Nr. 4, S. 1208–1244, 1997.

[26] Friedemann Mattern.
Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation.
In: *Journal of Parallel and Distributed Computing*, 18(4), S. 423–434, August 1993.

[27] Neeraj Mittal, Vijay K. Garg.
On Detecting Global Predicates in Distributed Computations.
In: *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, S. 3–10, April 2001.

[28] K. Mani Chandy, Leslie Lamport.
Distributed Snapshots: Determining Global States of Distributed Systems.
In: *ACM Transactions on Computer Systems*, Bd. 3, Nr. 1, S. 63–75, Februar 1985.

[29] Jerry Fowler, Willy Zwaenepoel.
Causal Distributed Breakpoints.
In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, S. 134–141, 1990.

[30] M. Dubois, C. Scheurich, F. Briggs.
Synchronization, Coherence, and Event Ordering in Multiprocessors.
In: *IEEE Computer*, Bd. 21, Nr. 2, S. 9–21, Februar 1988.

[31] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, J. Hennessy.
Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.
In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ACM, S. 15–26, Februar 1990.

[32] A. J. Bernstein.
Analysis of Programs for Parallel Processing.
In: *ACM Transactions on Electronic Computers*, EC-15(5), S. 757–763, 1966.

[33] Madalene Spezialetti, Phil Kearns.
Efficient Distributed Snapshots.

In: *Proceedings of 6th International Conference on Distributed Computing Systems*, S. 382–388, 1986.

[34] S. Venkatesan.
Message-optimal Incremental Snapshots.
In: *Proceedings of the 9th International Conference on Distributed Computing Systems*, S. 53–60, Newport Beach, California, USA, Juni 1989.

[35] Hon F. Li, T. Radhakrishnan, K. Venkatesh.
Global State Detection in Non-FIFO Networks.
In: *Proceedings of the 7th International Conference on Distributed Computing Systems*, S. 364–370, September 1987.

[36] Ten H. Lai, Tao H. Yang.
On Distributed Snapshots.
In: *Information Processing Letters*, 25(5), S. 153–158, Mai 1987.

[37] Carroll Morgan.
Global and Logical Time in Distributed Algorithms.
In: *Information Processing Letters*, 20(5), S. 189–194, Mai 1985.

[38] Barton P. Miller, Jong-Deok Choi.
Breakpoints and Halting in Distributed Programs.
In: *Proceedings of the 8th International Conference on Distributed Computing Systems*, S. 141–150, Juni 1988.

[39] Dieter Haban, Wolfgang Weigel.
Global Events and Global Breakpoints in Distributed Systems.
In: *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, Bd. II, S. 166–175, IEEE Computer Society, Januar 1988.

[40] Richard Koo, Sam Toueg.
Checkpointing and Rollback-Recovery for Distributed Systems.
In: *IEEE Transactions on Software Engineering*, SE 13(1), S. 23–31, Januar 1987.

[41] Hon F. Li, Gabriel Girard.
A Hierachy of View Consistencies and Exact Implementations.
In: *Proceedings of 1999 Workshop on Software Distributed Shared Memory*, Rhodos, Griechenland, S. 109–114, Juni 1999.

[42] Gabriel Girard, Hon F. Li.
Evaluation of Two Optimized Protocols for Sequential Consistency.
In: *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, Maui, Hawaii, USA, 1999.

[43] Anne E. Condon, Alan J. Hu.
Automatable Verification of Sequential Consistency.
In: *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, S. 113–121, 2001.

[44] Sarita V. Adve, Kourosh Gharachorloo.
Shared Memory Consistency Models: A Tutorial.
In: *IEEE Computer*, Bd. 29, Nr. 12, S. 66–76, 1996.

[45] P. Hutto, M. Ahamad.
Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories.
In: *Proceedings of the 10th International Conference on Distributed Computing Systems*, IEEE, S. 302–311, 1990.

[46] Michel Raynal.
Sequential Consistency as Lazy Linearizability.
In: *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, S. 151–152, 2002.

[47] R. Lipton, J. Sandberg.
PRAM: A Scalable Shared Memory.
In: *Technical Report CS-TR-180-88*, Princeton University, September 1988.

[48] Mark Burgin, Marc L. Smith.
Compositions of Concurrent Processes.
In: *Communicating Process Architectures*, IOS Press, Schottland, S. 281–296, September 2006.

[49] E. Schrödinger.
Die gegenwärtige Situation in der Quantenmechanik.
In: *Naturwissenschaften 23*, S. 807–812, 823–828, 844–849, 1935.

[50] J.D. Day.
The (Un)Revised OSI Reference Model.
In: *Computer Commun. Rev.*, Bd. 25, S. 39–55, Oktober 1995.

[51] Andrew S. Tanenbaum, Maarten van Steen.
Distributed Systems – Principles and Paradigms.
Second edition, Pearson Education Limited, 2007.

[52] George Coulouris, Jean Dollimore, Tim Kindberg.
Verteilte Systeme – Konzepte und Design.
3. überarbeitete Auflage, Pearson Education Limited, 2002.

[53] Jean Bacon.
Concurrent Systems – An Integrated Approach to Operating Systems, Database, and Distributed Systems.
Addison Wesley, 1993.

[54] Ben Shneiderman.
Designing the User Interface.
Third edition, Addison Wesley, 1997.

[55] Michael Himsolt.
GML: A Portable Graph File Format.
Technischer Bericht, Universität Passau, Deutschland. Verfügbar unter:
http://www.infosun.fim.uni-passau.de/Graphlet/GML

[56] Michael Himsolt.
GML: Graph Modelling Language.
Universität Passau, Deutschland, Dezember 1996. Verfügbar unter:
http://www.infosun.fim.uni-passau.de/Graphlet/GML

[57] http://www.yworks.com/en/products_yed_about.htm

[58] http://setiweb.ssl.berkeley.edu